Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

ALP: Adaptive Lossless floating-Point Compression

Author: Leonardo Xavier Kuffo Rivero (2722539)

1st supervisor:	Prof. Peter Boncz
daily supervisor:	Ph.D. (c) Azim Afroozeh
2nd reader:	Ph.D. Pedro Holanda

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

August 28, 2023

Abstract

IEEE 754 doubles do not exactly represent most real values, introducing rounding errors in computations and [de]serialization from/to text. These rounding errors inhibit the use of existing lightweight compression schemes such as Delta and Frame Of Reference (FOR), but recently new schemes were proposed: Gorilla, Chimp, Chimp128, PseudoDecimals (PDE), Elf and Patas. However, their compression ratios are not better than those of general-purpose compressors such as zstd; while [de]compression is much slower than Delta and FOR. We propose and evaluate ALP, that significantly improves these previous schemes in both speed and compression ratio. We created ALP after carefully studying the datasets used to evaluate the previous schemes. To obtain speed, ALP is designed to fit *vectorized execution*. This turned out to be key for also improving the compression ratio, as we found in-vector commonalities to create compression opportunities. ALP is an adaptive scheme that uses a strongly enhanced version of PseudoDecimals (1) to losslessly encode doubles as integers if they originated as decimals, and otherwise uses vectorized compression of the doubles' front bits. Its high speeds stem from our implementation in scalar code that auto-vectorizes using building blocks provided by the FastLanes library (2), and an efficient two-stage compression algorithm that first samples row-groups and then vectors. The evaluation shows that ALP can compress orders of magnitude faster and better than all current floating-point compressors in micro-benchmarks and in end-to-end queries inside a query processing engine.

To my friends (Galo, Mabe, Cristhian, Daniela, Zhaolin, Ruben, Luisfer, Lords, Pandis and many more), family (Mom, Dad, Tia, Tio Ric, Tio Leo (†), Manuel, Bolivar, Marli, and all) and Many (†) which always supported me during this 2-year adventure that is coming to an end.

This thesis, and my entire career, would have not been possible without all of you. Thanks.

I also want to thank Peter for the opportunity of being part of his team at CWI on this challenging project, and Azim for guiding me through the process with patience and care. This thesis marks the end of my master, but also the start of my PhD. under the supervision of Peter; which I am thoroughly excited for.

Acknowledgements

This work would not have been possible without the help of my supervisor Prof. Peter Boncz; whose supervision, feedback and corrections made this thesis cross the excellence bar and achieve outstanding results. Similarly, this work would not have been possible without the help of my daily supervisor Ph.D. (c) Azim Afroozeh.

ALP is a result of the joint work of Azim, Peter and me. ALP would not have been possible without any of our contributions.

This thesis was done as part of an internship at CWI (Centrum Wiskunde & Informatica).

A summarized version of this work has been accepted for publication in ACM SIGMOD.

Contents

List of Figures

List of Tables

1

Introduction

Data analytics pipelines manipulate floating-point numbers (64-bit *doubles*) more frequently than classical enterprise database workloads, which typically rely on fixed-point *decimals* (systems often store these as 64-bit integers). Floating-point data is also a natural fit in scientific and sensor data; and can have a temporal component, yielding *time series*.

Analytical data systems and big data formats have adopted columnar compressed storage (17, 18, 19, 20, 21, 22), where the compression in storage is either provided by general-purpose or lightweight compression. Lightweight methods, also called "encodings", exploit knowledge of the type and domain of a column. Examples are Frame Of Reference (FOR), Delta-, Dictionary-, and Run Length Encoding (RLE) (23, 24, 25). The first two are used on high-cardinality columns and encode values as the addition of a small integer with some fixed base value (FOR) or the previous value (Delta). These encodings also *bit-pack* the small integers into just the necessary bits. However, with IEEE 754 doubles (26), additions introduce rounding errors, making Delta and FOR unusable for raw floating-point data. General-purpose methods used in big data formats are gzip, zstd, snappy and LZ4 (27, 28, 29). LZ4 and snappy trade more compression ratio for speed, gzip the other way round, with zstd in the middle. The drawback of general-purpose methods is that they tend to be slower than lightweight encodings in [de]compression; also, they force decompression of large blocks for reading anything, preventing a scan from *pushing down* filters that could *skip* compressed data.

Recently though, a flurry of new floating-point encodings were proposed: Gorilla (30), Chimp and Chimp128 (9), PseudoDecimals (PDE) (1), Patas (31) and Elf (12). A common idea in these is to use the XOR operator with a previous value in a stream of data; as combining two floating-point values at the bit-pattern level using XOR provides somewhat similar functionality to additions, without the problem of rounding errors. Chimp does

an XOR with the immediate previous value, whereas Chimp128 XORs with one value that may be 128 places earlier in the stream – at the cost of storing a 7-bit offset to that value. After the XOR, most bits are 0, and the Chimp variants only store the bit sequence that is non-zero. Patas, introduced in DuckDB compression (31), is a version of Chimp128 that stores non-zero *byte*-sequences rather than bit-sequences. Whereas Patas trades compression ratio for faster decompression, Elf (12) does the opposite: it uses a mathematical formula to zero more XOR bits and improve the compression ratio, at the cost of lower [de]compression speed. PDE is very different as it does not rely on XOR: it observes that many values that get stored as floating-point were originally a *decimal* value and it endeavours to find that original decimal value, and compress that.

While these floating-point encodings avoid the need to always decompress largish blocks, as required by general-purpose compression, and thereby allow for predicate push-down in big data formats (32), their [de]compression speed (as well as compression ratio) is not much higher than that of general-purpose schemes (12); in other words, these encodings are not quite lightweight.

We introduce ALP, a lightweight floating-point encoding that is *vectorized* (33): it encodes and decodes arrays of 1024 values. It is implemented in dependency-free scalar code that C++ compilers can *auto-vectorize*, such that ALP benefits from the high SIMD performance of modern CPUs (34, 35). In addition, ALP achieves much higher compression ratios than the other encodings, thanks to the fact that vectorized compression does not work value-at-a-time but can take advantage of commonalities among all values in one vector. Its vectorized design also allows ALP to be *adaptive* without introducing space overhead: information to base adaptive decisions on is stored once per vector rather than per value, and thus amortized. While per-value adaptivity (e.g., Chimp[128] has four decoding modes) needs control instructions (if-then-else) for every value, and can run into CPU branch mispredictions, ALP's per-*vector* adaptivity only needs control-instructions once per vector, but vector [de]compression itself has very few data- or control dependencies, leading to higher speeds.

1.1 Contributions

Our main contributions are:

- a study of the datasets that were used to motivate and evaluate the previous floating-point encodings, leading to the new insights (e.g., many floating-point values actually were originally generated as a decimal).
- the design of ALP, an adaptive scheme that either encodes a vector of values as compressed decimals, or compresses only the front-part of the doubles, that holds the sign, exponent, and highest bits of the fraction part of the double.
- an efficient two-level sampling scheme (happening respectively per row-group, and per vector) to efficiently find the best method during compression.
- an open-source implementation of ALP in C++ that uses vectorized lightweight compression that can cascade (e.g., use Dictionary-compression, but then also compress the dictionary and the code columns, with Delta, RLE, FOR – such as provided by (1, 3, 36)).
- an evaluation (microbenchmarks and end-to-end queries) versus the other encodings on the datasets that were used when these were proposed, showing that ALP is faster *and* compresses better (as summarized in Figure 5.1).

1.2 Outline

First, we present a comprehensive literature review on lightweight compression in section 2, with special attention to encodings tailored for floating-point data. Next, in section 3, we present our in-depth analysis of real-world *double* datasets which uncovered unexploited opportunities for compression. In section 4 we present ALP and its design decisions. Next, we evaluate ALP compression ratios and [de]compression speed by performing micro-benchmarks and end-to-end query speed benchmarks on section 5. Finally, in sections 6 and 7 we discuss our results and present conclusions of our work.

2

Literature Review

We present a comprehensive literature review of lightweight compression methods. We start by introducing the most basic ideas of general purpose and lightweight compression schemes in section 2.1. Next, in section 2.2, 2.3, 2.4 and 2.5 we review lightweight compression schemes tailored for specific data types. In section 2.5 we focus on lightweight schemes for floating-point data. Next, in section 2.6 we review how lightweight compression schemes can be used in cascade. In section 2.7 we introduce storage layouts in order to expand on how lightweight compression has been leveraged in the context of different [big]-data formats and data systems on sections 2.8 and 2.9. Finally, we introduce white-box compression and compression by attributes decomposition in sections 2.10 and 2.11 respectively.

2.1 General Purpose vs. Lightweight Compression

2.1.1 General Purpose Compression

General Purpose Compression (GPC) methods are the most commonly used algorithms to compress data regardless of its context. They can be used to compress files in an Operating System or to compress records and attributes in a database system. Such versatility stems from the fact that they can be applied to any small or large stream of bits without needing context of the data nature. Thus, they are data-agnostic. GPC methods work like a black box that can be plugged into any data to compress it without loss of information. Most of these algorithms are based on variations of the Lempel-Ziv family of algorithms (37) that works by trying to build a dictionary based on patterns found on the bits streams; achieving good compression ratios on a variety of data [1]. However, such algorithms have long suffered

[1] `https://github.com/facebook/zstd#benchmarks`

shortcomings in terms of speed and random access of the compressed information.

Most GPC methods have been shown to perform relatively slow on both data compression and decompression (8, 9, 38). Hence, GPC methods are not ideal in the context of analytical databases (OLAP) in which *speed* of information retrieval is of critical importance. In addition to this, most GPC methods perform both compression and decompression on quite large blocks of data. The latter becomes a problem when random access to information is required. For instance, if one wants to access a single record compressed with a GPC algorithm, most probably a decompression of an entire block has to be done to do so. Moreover, GPC methods performance is hindered when small blocks of data are compressed since the algorithms are not able to find enough patterns to achieve compression. However, there have been recent advancements in GPC methods to improve their speed and to work as efficiently on smaller streams of data. As an example, we will analyze Zstandard – one of the most widely used GPC algorithms.

Zstandard[1], also known as Zstd, is a multithread lossless general purpose compression algorithm. Zstandard finds patterns in streams of bits by using the LZ77[2] algorithm. LZ77 is one variation of the Lempel-Ziv family of algorithms. This family of algorithms tries to find recurrent bit patterns to build a dictionary to map these –*ideally*– long bit patterns into smaller codes. As the algorithm advances through the data, this dictionary is continuously updated in such a way that it optimizes the patterns. Zstd also uses a variation of a Huffman (entropy) coder (FSE[3]) to further optimize the dictionary codes based on the frequency of the patterns (i.e. most repeated patterns are assigned the smallest codes). Since the algorithm depends on its own history of patterns, it struggles with small streams of data for which there is not a big enough window of bits to build such a dictionary. However, Zstd offers a special mode to be able to perform well on small data. In this mode, the algorithm passes through a sample of the data to build the dictionary a-priori. Afterwards, this dictionary is used to compress the data. This is a way to avoid the algorithm to start from *nothing* before compressing; which works especially well for small streams. The latest version of Zstd (i.e. 1.5.1) reports benchmark results of x2.8 improved compression ratios on average and 500 MB/s and 1660 MB/s of compression and decompression speeds which makes it one of the fastest and most performant GPC method. Similar to any GPC method, when data is compressed with Zstd it is not possible to perform random access.

[1] `https://github.com/facebook/zstd`
[2] `https://en.wikipedia.org/wiki/LZ77_and_LZ78`
[3] `https://github.com/Cyan4973/FiniteStateEntropy`

Other GPC methods such as Brotli[1] or LZ4[2] follow a similar architecture. They use a variation of the Lempel-Ziv family of algorithms and afterwards use an entropy coder to optimize the dictionary patterns codes based on the patterns entropy. The differences between most GPC methods usually are on small optimizations, architectural decisions and API design.

2.1.2 Lightweight Compression

In contrast to General Purpose compression, lightweight methods, also called "encodings", exploit knowledge of the type and domain of the data. For instance, if one knows that the data to compress are 32-bit integers, one could encode the difference between consecutive numbers in the hope of obtaining a smaller integer. Then, the encoding can further *bit-pack* the small integers into just the necessary bits. Lightweight encodings have been demonstrated to be orders of magnitude faster than most General Purpose compression algorithms in terms of [de]compression time (8, 12, 13, 38). This is thanks to the fact that they usually can fit in vectorized execution and can be fully (or partially) implemented using SIMD instructions (single-instruction-multiple-data). Furthermore, they usually enable random access into the data; in contrast to general purpose compression which is block-based (i.e. the entire block of data needs to be decompressed to access any value in it). For these reasons, LWC methods and databases, which store values of the same attribute close together (i.e. columnar storage) are a great fit. We will discuss more about how data formats and databases have leveraged LWC methods in sections 2.8 and 2.9.

In the next sections, we review a series of lightweight encodings that take advantage of different data types (i.e. integers, strings, floating-points) in addition to encodings that can be used regardless of the type of the data. A summary of these reviewed encodings is presented in Figure 2.1. It is important to note that we will give special attention to floating-point encodings since we are interested in the development of a new encoding for this data type. Furthermore, we will also analyze relevant variants of each encoding and the efforts made to SIMDize their implementation.

2.2 All-Type Encodings

Some lightweight compression schemes can be used to encode data regardless of their type. In other words, they are data-agnostic. These approaches take advantage of data

[1] `https://github.com/google/brotli`
[2] `https://github.com/lz4/lz4`

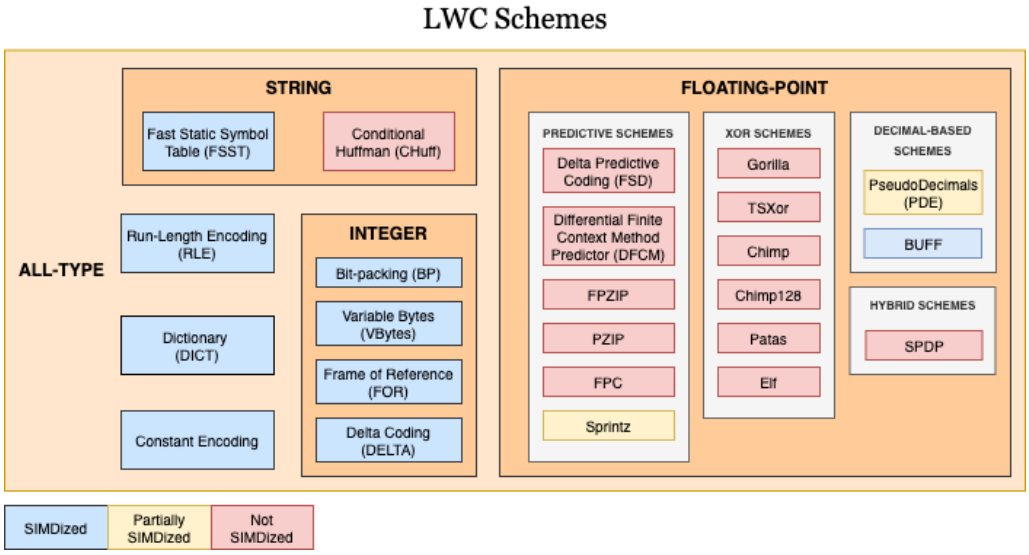

Figure 2.1: Overview of reviewed LWC methods

distributions and high percentages of repeated values. However, if the data lacks these properties, they can result in negative compression (i.e. compressed data is bigger than uncompressed). For this reason, it is usually encouraged to scan or sample data before applying these types of encodings; or have knowledge of the data a-priori. It is important to note that even these all-type encodings, in contrast to GPC, do not work over one continuous large stream of bits, but they have to be able to identify the bits corresponding to each tuple (i.e. value) inside the data.

2.2.1 Run-Length Encoding (RLE)

RLE takes advantage of consecutive repetition of values (runs) inside a dataset. For each value in a dataset, RLE encodes the value itself and the number of times it is consecutively repeated. An example of this can be seen in Figure 2.2.a. When used on data that lacks such runs of values, RLE results in negative compression (i.e. output is bigger than input) since the original value is stored alongside an additional *length* which would be near to 1 every time (Figure 2.2.b). If an entire block of data consists of the exact same value, a special case of RLE may be used in which all the block is encoded as one single value without explicitly specifying the length of such run. However, in such scenarios, the algorithm needs to know how many values a *block* is comprised of. The latter is called One Value or **Constant Encoding**.

In (36), Damme et al. presented a vectorized implementation of RLE based on parallel

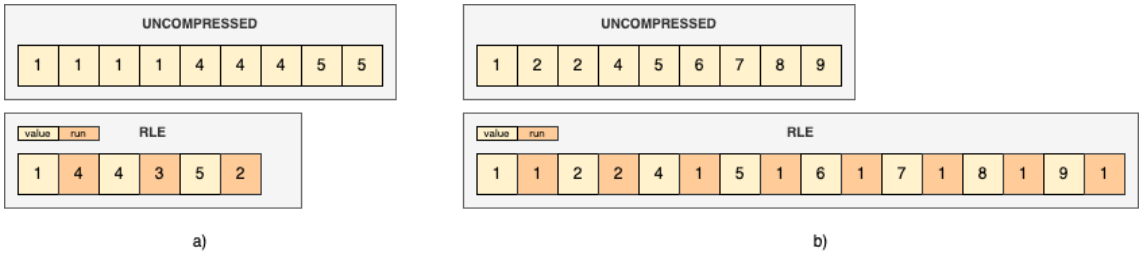

Figure 2.2: Overview of RLE applied in an ideal scenario (a), and in a not-ideal scenario (b).

comparisons; analyzing vectors of 4 integers at a time for each value to encode (i.e. 128 bits for 32-bit integers). In this approach, a 4-bit comparison mask is generated from the parallel comparisons performed with the value and each value of the vector. Afterwards, the trailing bits of this mask are analyzed to determine whether to store the value and its run length (i.e. run ended due to a non-equality found) or whether to advance to the next vector on the same run (i.e. the four bits of the mask deemed as equality). However, decoding requires two loops, one that iterates over the values, and an inner one that iterates over the run lengths. The latter is hard to fully SIMDize since SIMD instruction sets do not contain control instructions necessary for loops, and the scalar code required for this will incur branch mispredictions (especially frequently on low run-lengths). To tackle this problem Afroozeh & Boncz proposed FastLanes-RLE (3) which is part of a bigger project named FastLanes that we will review in subsection 2.3.5

2.2.2 Dictionary Encoding (DICT)

When data contains a high percentage of repeated values, but these values are not consecutively arranged, a dictionary of these values can be built. Inside such a dictionary, each value is mapped to a *smaller* value (usually an incremental integer starting from 0) which is used to replace the original value in the data (Figure 2.3.a). At decoding time, the original data is reconstructed using the dictionary mappings. Dictionary Encoding results in negative compression when data contains a small-to-zero percentage of repeated values. The latter results in a dictionary which size is equal to the raw data in addition to the mapped values. An example of this can be seen in Figure 2.3.b.

Variations of Dictionary Encoding have been developed in order to maintain a dictionary size small when the encoded data contains outliers. For instance, *Patched Dictionary*

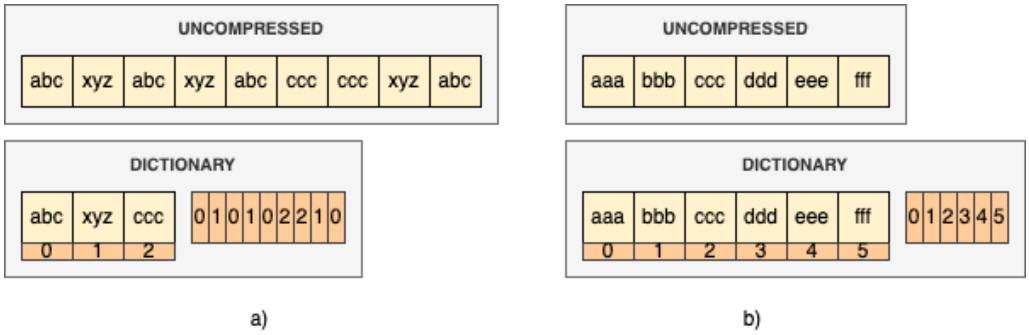

Figure 2.3: Overview of dictionary encoding applied in an ideal scenario (a), and in a not-ideal scenario (b).

Encoding (PDICT) (17) introduces the concept of *exception patching* in the dictionary. **Exceptions** are values in the original data which, when introduced to the dictionary, diminish the compression ratios substantially due to being outliers (e.g. infrequent values). These exceptions are detected and stored uncompressed in a separate storage segment alongside a reference to the position in which these occurred in the original data. This technique of exception handling was introduced in (17, 34) and it can potentially be exploited in other encodings that also expect certain properties in the data.

This concept of exceptions may also be used when one wants to build a *constant-size dictionary*. For instance, if one wants to build a dictionary of at most 16 entries, only the first 16th most frequent values in the data would be added to the dictionary and other values would be encoded as exceptions. This is useful when the data follows a skewed distribution or when the codes of a dictionary need to be *bitpacked* in a fixed bit-width. We will dive more into the latter when analyzing bit-packing encoding in subsection 2.3.2. Along the way, we will encounter more LWC methods that leverage the use of exceptions in order to achieve more speed or increased compression ratios.

2.3 Integer Encodings

Integers have received quite some attention regarding lightweight compression (3, 36, 38, 39). This is no surprise since in many applications such as search engines and database systems, data (and even indexes) are stored as arrays of integers. Hence, not only there are quite a few well-established algorithms to compress them in a lightweight fashion, but they have been optimized to be *fast* by using vectorized and SIMDized implementations up until the point of being able to decode hundreds of billions of compressed integers per second (3). Integers are usually represented in memory as 32-bits. In this representation,

every bit represents a power of 2, being the least significant bit (i.e. the right-most one) 2^0 and the most significant bit being 2^{31} (i.e. the left-most one). If the bit is 1 then the power of two at that position is added to a summation to compute the integer number; otherwise, it is ignored. From here we can make a difference between *signed* and *unsigned* integers. As their name implies, signed integers support negative numbers, while unsigned integers do not. In signed integers, the left-most bit is used to represent the sign of the number (i.e. 1 for negative, 0 for positive). Hence, unsigned 32-bit integers range from 0 to 4294967295 (i.e. $2^{32} - 1$) and signed integers from -2147483648 to 2147483647 (i.e. $2^{31} - 1$ on both negative and positive ranges). It is important to note how this representation based on summations of powers of 2 results in integers potentially having a lot of leading zeros. For instance, if numbers in a sequence are within the range from 0 to 7, they will always have 29 trailing zeros; in other words, we can represent them with at most 3 bits. The latter is a powerful observation leveraged by many integer encodings. There also exist other integer representations, for instance, `long` integers (represented by 64-bits) and `short` integers (represented by 16-bits); both maintaining the same definition based on a bit-wise summation of powers of 2. The key difference in these smaller/bigger representations relies on the range of the integers that they are able to represent. We now review the most relevant integer LWC methods and their variants.

2.3.1 Variable Bytes (VByte)

As we mentioned, the 32-bit representation of integers sometimes is not efficient. Such is the case of small numbers. Variable bytes, also called VByte, is an encoding that tries to only use the necessary *bytes* needed to represent integers (39). Although it may sound tempting to directly cut the leading bytes which are full of 0's, this will represent a problem at decoding time, since every number could have been encoded with a different number of bytes. Thus, VByte proposes to encode integers as sequences of individual bytes. In every byte, a control bit is located at the least significant bit (i.e. the right-most bit) to determine if one must continue reading the next byte to complete the integer representation or if that byte marks the end of the integer. For instance, let us say we want to encode the number 135 (`00000000 00000000 00000000 10000111`). It has 24 leading 0's. In order to use VByte we have to take its significant bits (i.e. `10000111`; the bits after the leading 0's) and for every 7 significant bits, add a control bit packed into a byte. Hence, the bytes encoded with VByte would be: `00000011 00001110`. In the end, our number can be recovered by removing the control bit (marked in red) and concatenating the remaining 14 bits. We have then effectively reduced the storage size

of our number in half. In practice, VByte has been demonstrated to be suboptimal in terms of compression ratios when compared to other encoding schemes (34). However, its simple algorithm makes it faster than most non-SIMDizable compression methods even with its scalar implementation (34). Nevertheless, a vectorized implementation using SIMD instructions called Masked VByte is presented in (40) which achieved twice as fast speed at decoding time than scalar VByte implementation. Furthermore, a more recent variant called StreamVByte presented by Lemire et al. (38) achieved even better performance (around x2 speed) than Masked Vbyte by optimizing the way in which control bits are stored and read.

2.3.2 Bit-[un]packing (BP)

Taking Variable Bytes to the level of *bits* results in Bit-packing (BP). Such an approach only uses the exactly necessary *bits* to store integers. For instance, to encode the number 135 (`00000000 00000000 00000000 10000111`) we only take its significant bits (i.e. `10000111`; the bits after the leading 0's). As a general rule, every integer in the range from $[0, 2^b)$ can be exactly represented with b bits. A logarithm in base 2 can also be used to determine how many bits one exactly needs to store an integer. For example, the number 135 needs $ceil(log_2(135)) = 8$ bits.

The reverse operation (i.e. to decode the compressed integers into 32-bit integers) is called bit-unpacking. Bit-unpacking can be implemented without control dependencies (i.e. if-else-then branches) with five instructions (LOAD, SHIFT, AND, OR, and STORE) (41).

As one may have noticed, bitpacking faces the same problem as VByte: How to decode bits of variable length without a control bit? In order to solve this, the community has adopted the implementation of BP in blocks of integers (e.g. 1024 integers) (3, 17, 34). Each block therefore has a range of integers. The maximum number of such range will then determine the number of bits b to encode every value in the block. In other words, every value in the block is said to be of a *fixed bit-width*. This b value is stored as a *header* or as *metadata* of the compressed data. Recently, Wang & Song presented *descending bit-unpacking* (42), in which a block of bitpacked integers support variable bit-width in a descending fashion; effectively never using more bits than needed for the integers. This type of bit-packing is effective for data which are known to have a skewed distribution and that can be sorted in descending order.

There have been efforts to make Bit-[un]packing fast by fully vectorizing the algorithm in blocks of different lengths (e.g. 128 or 32 integers at a time) and using SIMD-instructions (3, 34, 43) which have effectively achieved throughput of encoding and decod-

ing of billions of integers per second (e.g. SIMD-BP128 (34)) and even hundred billions of integers per second (i.e. FastLanes BP (3)). The latter is achieved by Afroozeh & Boncz in the FastLanes project (3) by introducing a novel 1024-bit interleaved data layout not tied to specific SIMD-widths, which accomplishes completely data-parallel decompression. In subsection 2.3.5 we will review FastLanes more in detail.

As we will see in the next subsection, bit-[un]packing is an essential part of other encodings to achieve impressive compression ratios. Thus, it is no surprise that such efforts have been made to improve its speed.

2.3.3 Frame of Reference (FOR)

In 1998, Goldstein et al. proposed Frame of Reference (FOR) (23). As its name suggests, FOR compresses a sequence of integers using a *referential value* to reduce the range of the entire sequence. For instance, let us assume we have an array of integers such as: [135, 136, 137, 138]. Such an array is within the range of $[0, 2^8)$, hence every value can be bit-packed with 8 bits. However, if we take the minimum value of this range as our *frame of reference*, namely 135; and we subtract this value from every element of the sequence we end up with the following array: [0, 1, 2, 3]. This array is within the range of $[0, 2^2)$ hence every value can be bit-packed with 2 bits. Thus, for a block of values with M and m being respectively the maximum and the minimum value of the block; FOR is able to bit-pack every number into $log_2(M - m + 1)$ bits (ceiled). In contrast to simply bit-packing; which can only bit-pack every number into $log_2(M)$ (ceiled). Goldstein et al. (23) reported compression ratios of x4 on real datasets and x88 on datasets identified with low cardinalities (i.e. small difference between their minimum and maximum values). From there, many variants of FOR have been developed to further improve its compression ratios and [de]compression speed.

FOR performance can be negatively impacted if there are values out of the distribution in the data (i.e. outliers). For instance, if we add the number 1023 into our first example array it will result in a bitpacking of 10 bits ($ceil(log_2(1023))$). Applying FOR will still keep our numbers within the same range of 10 bits ($ceil(log_2(1023 - 135 + 1))$). To tackle this problem, Zukowski et al. (17) proposed Patched Frame of Reference (PFOR)–a FOR which supports the encoding of *exceptions*. PFOR takes samples from a chunk of the data in search of an optimal bit-width b that should result from applying FOR on that sample. A chunk is defined here as 128 consecutive integers in the data. By taking samples, the algorithm aims to find a target b for the entire chunk which effectively ignores outliers. Hence, the used minimum and maximum values of a chunk are not necessarily the true

ones. Afterwards, every value in the chunk that needs more than 2^b bits to be encoded after FOR is treated as an exception. Exceptions are stored uncompressed (32-bits) in a separate location called the *exceptions section*. PFOR achieves better compression ratios than FOR and faster [de]compression (even faster than the fastest version of the Lempel-Ziv algorithm) thanks to its implementation without control dependencies.

Lemire et al. introduced FastPFOR and SimplePFOR (34). The main idea of both algorithms is to have a variant of PFOR such that it stores compressed blocks that are always aligned at a 32-bit level. They achieve the latter by not entirely skipping exceptions, rather they store the least b significant bits of the exception. This not only speeds up decompression but also creates opportunities to increase compression ratios. An extra location called the *byte array* is used to store in a byte-aligned way metadata such as the bit-width b, the number of exceptions and the location offset of the exceptions. The difference between SimplePFOR and FastPFOR is in how they store the truncated exceptions. SimplePFOR further compresses these using Simple-8b (44). On the other hand, FastPFOR, stores exceptions into 32 possible arrays, each array of a different bit-width. Each array is then bit-packed to its bit-width and padded to be aligned at a level of 32 bits. Finally, a variant of FastPFOR called SIMD-FastPFOR was presented, which works exactly the same but uses vectorized bit-packing. Both SimplePFOR and FastPFFOR achieved better compression ratios than PFOR but were slightly slower in compression and decompression. On the other hand, SIMD-FastPFOR was much faster than PFOR and both SimplePFor and FastPFOR. Achieving almost twice as fast decompression speed while only being in general slightly worse in compression ratios than PFOR.

Since bit-[un]packing is a key part of FOR [de]compression, FastLanes BP can be used as a building block for an even faster implementation of FOR (3). In fact, FastLanes proposes FFOR; a FOR implementation which fuses the kernels of both FOR and bit-[un]packing into a single kernel. We will discuss more regarding kernel fusion in subsection 2.6.1.

2.3.4 Delta Coding

In 1997, Ng & Ravishankar presented Tuple Differential Coding (24), also known as Delta Coding or just DELTA. The idea of Delta Coding is to compress integers by only storing the differences between consecutive values. For instance, our example of integers: [135, 136, 137, 138], will be encoded as: [135, 1, 1, 1]. Therefore, the i^{th} value is equal to the running sum from all the previous values before i. As one may expect, the resulting integers must be bit-packed to achieve compression. However, the resulting delta-coded array could still have one *big* number which would *ruin* the bit-packing of the other rather

small integers. PDelta (17) introduces a base number equal to the first number (in our case 135) stored in another location in such a way that the output array results as follows: [0, 1, 1, 1]. Finally, PDelta also introduces *exceptions* in Delta Coding, similar to PFOR. This same work introduces PFOR-Delta to further improve compression ratios. As its name suggests, PFOR is applied to the differences obtained from Delta.

Delta coding can achieve better compression ratios than other encodings if the data is appropriate. However, it introduces data dependencies at decompression since an integer value can only be decoded when all its previous values have been decoded. To tackle this data dependency problem, Afroozeh & Boncz introduced a Unified Transposed Layout in FastLanes (3) which we talk about in the next subsection.

2.3.5 FastLanes

FastLanes proposes a new layout optimized for a virtual SIMD register of 1024 bits (i.e. `FLMM1024`) which supports all common denominator instructions of most common ISAs. This layout called the Unified Transposed Layout, reorders tuples in such a way that they are (i) independent of the SIMD register width and (ii) they maximize independent work of instructions. At a high level, the key idea of the Unified Transposed Layout is to reorganize 1024 values (i.e. tuples) into eight 8x16 transposed blocks and to put these eight blocks in the order "04261537" (being these the original block indexes). This Unified Transposed Layout helps to tackle some of the problems we previously discussed regarding RLE and DELTA SIMDization while introducing significant speed improvements to other encodings and to Bit-[un]packing.

For instance, FastLanes-RLE leverages the Unified Transposed Layout and a special representation of RLE that uses the value array of standard RLE as a dictionary (in which contrary to normal DICT, values can repeat) and replaces the lengths with DELTA-encoded codes. In RLE, the next value is either equal to the previous, in which case the code is equal to the previous and hence its delta is 0; or the next value is different and stored in the next position of the value array (now, the dictionary), array and hence the code delta is 1. As such, the storage for the lengths becomes a bitmap and [de]compression can be handled with the efficient FastLanes primitives for DELTA decoding (that rely on the Unified Transposed Layout). Figure 2.4 depicts the differences between a traditional RLE compression (using DICT as internal representation) and FastLanes-RLE. Aside from FastLanes-RLE achieving better compression ratios than conventional RLE (up to average run-lengths of 12); it also achieves increased speed when the average runs in a 1024-value vector are less or equal to 15. The latter stems from the lack of branch-mispredictions that

FastLanes-RLE can achieve thanks to the Unified Transposed Layout tuples reordering (depicted in Figure 2.5) and the profit of using the full-width of a SIMD register.

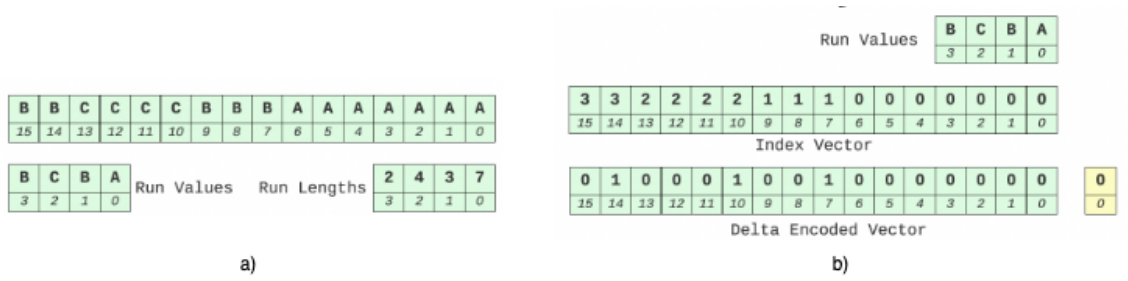

Figure 2.4: Classic RLE representation as a dictionary encoding (a) side by side with FastLanes-RLE representation (b) (Figure borrowed from Afroozeh & Boncz work (3))

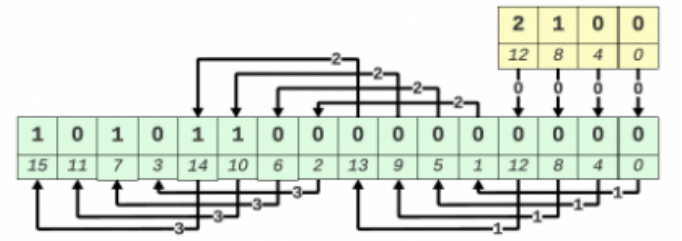

Figure 2.5: FastLanes-RLE index vector representation in the Unified Transposed Layout (Figure borrowed from Afroozeh & Boncz work (3))

This reordering that happens at the Unified Transposed Layout effectively breaks sequential dependencies which happen for instance in the DELTA encoding. FastLanes reports a performance higher than 40 tuples processed per CPU cycle on the faster platforms for 8-bit integers (i.e. 8-bit lane). In addition to this, the performance of Bit-unpacking can reach up to 60 tuples decompressed per CPU cycle for 8-bit integers (x40-x60 times against Scalar implementations); and 3x-4x times faster against Scalar implementations for wider types such as 64-bits. Furthermore, the performance of FOR in FastLanes is one order of magnitude over naive sequential bit-packed layouts.

Another advantage of FastLanes is the portability of the code in terms of heterogeneous ISAs and the capability of modern compilers to auto-vectorize the scalar code written within the FastLanes framework; thus avoiding the need for explicitly writing SIMD intrinsics for each different SIMD ISA (e.g. SSE, AVX, NEON).

2.4 String Encodings

Strings can be compressed by using Dictionary or RLE encoding. However, as we previously mentioned, such encodings require fully repeating strings to be effective. The latter is not useful for human-generated text such as descriptions, reviews or comments. Neither for common user-related database attributes such as usernames, emails or URLs. Hence, general purpose compression such as LZ4 has been the *easy* way out for compressing strings inside databases. However, we have already discussed the shortcomings of such approaches. Nevertheless, recently Boncz et al. (4) introduced Fast Static Symbol Table (FSST)–a true fast encoding for strings when Dictionary or RLE are not suitable.

2.4.1 Fast Static Symbol Table (FSST).

FSST key idea relies on the observation that in a sequence of string values (e.g. a database column), there are data commonalities amongst the *substrings* of such values. For instance, if a column in a database is comprised of URLs, the prefix `http://www` and the suffix `.com` would repeat several times amongst every value. Hence, if one could represent these frequently occurring substrings as smaller codes while maintaining a symbol table (i.e. a mapping of the codes to their substrings), compression can be achieved by storing the concatenation of such codes to build the original strings. Figure 2.6 shows a small example of this procedure in practice. This enables random access to compressed strings since each string is compressed to a (typically) shorted consecutive sequence of bytes, independent of the surrounding strings. Decompression is reasonably fast on platforms that allow unaligned memory access, and can often be postponed or avoided (by delaying decompression and operating on the compressed string).

Even though simple at first glance, a couple of challenges are presented: (i) How to build an efficient symbol table in terms of size and compression ratios? and (ii) How to prevent this symbol table from growing infinitely with data?

To build an efficient symbol table and prevent it from infinitely growing, symbols are bounded to a maximum of 8 bytes and codes are bounded to 1 byte. Hence, only a maximum of 256 symbols are used per table. It is important to note that the symbol table that is built at compression time, is immutable (i.e. static). In other words, it will not change if new data is compressed – this is the "static" in Fast Static Symbol Table (FSST). Finally, a special code is defined as an *escape code*. This escape code is used to encode raw substrings after it if they are not found in the symbol table. Hence, giving the algorithm flexibility to compress new data without having to re-build the symbol table. The symbol

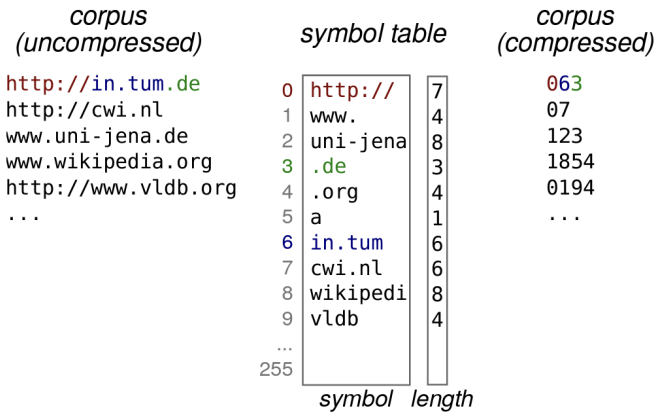

Figure 2.6: Overview of FSST in action with a small example (Figure borrowed from Boncz et al. work (4))

table itself is constructed in a bottom-up approach by iteratively scanning samples of the data, using a mechanism similar to a genetic algorithm.

FSST was evaluated against LZ4 using 23 string columns from real-world data which ranged from human-readable names and human-generated texts. FSST achieved on average x2 times more compression ratios than LZ4 and a slightly faster compression and decompression. Moreover, FSST achieved significant improvements over LZ4 when tested on selective queries. The lower the percentage of information that was queried (i.e. selective queries), the more orders of magnitude FSST performed faster than LZ4. This is because FSST can just skip over strings that belong to tuples that are not selected, but LZ4 needs to decompress entire blocks of values to perform random access. On top of that, FSST decompression is fast as it requires few instructions, does not have control dependencies, and the symbol table fitting into the L1 CPU Cache.

2.4.2 Conditional Huffman (CHuff)

CHuff (45) trades-off [de]compression speed for higher compression ratios than FSST; while still maintaining the capabilities of random access into the data. As its name suggests, CHuff uses a set of Huffman Encoding trees optimized for read operations built from random samples of the data. Huffman Encoding is a type of entropy encoding that maps symbols to smaller codes that do not share a prefix equal to another code. Furthermore, more frequent symbols are mapped to the smallest codes. Huffman Encoding mappings can be represented in a binary tree data structure in which leaves closer to the root are more frequent symbols on the data. The key idea to CHuff is to build a set of Huffman trees

instead of only one; based on the prefix of common strings. At decompression time, the tree used to compress the original substrings is determined based on its code K-length prefix. Hence the algorithm name, *Conditional* Huffman. Since the trees are built from a sample of the data, and each tree has a limited size, an escape code is used as an exception-like mechanism.

CHuff achieves 24% higher compression ratios than FSST, also beating GPC methods such as LZ4 and Zstd; while still supporting random access into the data. In terms of decompression speed, CHuff is faster than FSST only in settings in which there is a restrictive disk throughput bound. In other words, CHuff works better than FSST in the context of low read-throughput. However, we have to note that this is mostly thanks to the higher compression ratios. On absolute performance, FSST is still faster.

2.5 Floating-Point Encodings

IEEE 754 (26) represents 64-bit doubles in 3 segments of bits: 1 bit for sign (0 for negative, 1 for positive), 11 bits for an exponent (represents an unsigned integer from 0 to 2047), and 52 bits for the fraction (also known as mantissa) –*each bit representing an inverse power of 2 inside a summation.* Figure 2.7 shows a depiction of this representation and Formula 2.1 shows how this representation is decoded bit-by-bit into a double.

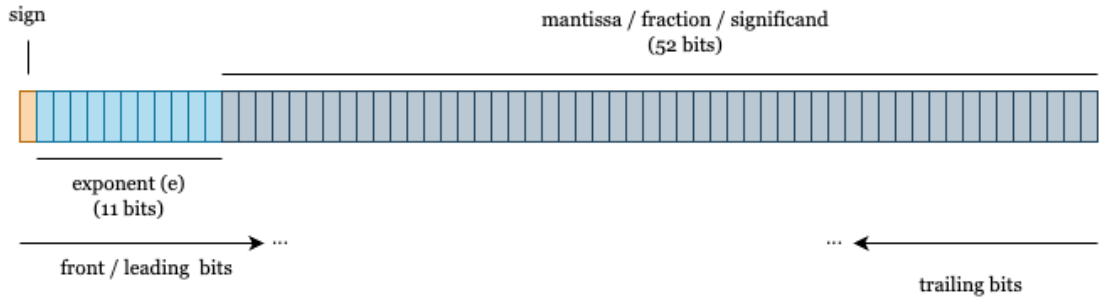

Figure 2.7: IEEE 754 double precision floating-point bitwise representation. One bit for sign, 11 bits for exponent and 52 bits for mantissa / fraction.

$$(-1)^{sign}\left(1+\sum_{i=1}^{52} b_{52-i}2^{-i}\right)\times 2^{e-1023} \tag{2.1}$$

The higher the precision of a double, the more entropy is present in the bits of the mantissa. The same standard also defines 32-bit floats (with 8 bits for exponent and 23 for mantissa) and half-precision (float16) as well as quadruple and octuple floating points. These representations cannot accurately represent all real numbers. For instance, the real number 0.1 cannot be represented either in the 32-bit or 64-bit floating-point definitions. The closest one can get to this value in a double is 0.10000000000000005551. This creates precision errors when performing arithmetic (e.g. additions). Hence, using integer encodings such as FOR or DELTA on floating-points would result in lossy compression if used without care. Also, these representations make floats not suitable for bit-packing because *small* or low-precision numbers do not necessarily have a representation with many leading zero bits.

We now review the relevant work done in lossless lightweight floating-point compression; the main focus of this thesis. The techniques developed can be categorized mainly into three groups, which at the same time, tell the progression of how the encodings have evolved through time: (i) Predictive schemes, (ii) *purely* XOR schemes and (iii) Decimal-based schemes. There is an additional branch of research focused on lossy algorithms. Lossy compression trades off the precision of the data for higher compression ratios. In other words, the exact original data is lost. In this literature review, we will only focus on lossless compression methods; in which the output of the decompression process is the exact original data (bit by bit).

2.5.1 Predictive Schemes

Predictive Schemes were the first novel approaches designed to compress floating-point data (5, 18, 46, 47, 48). The core idea of these approaches is to use a *function* to generate a *predicted value* based on patterns found within the data prior to the value to encode. Predictive algorithms are as good as their prediction can be. If the predicted value and the value to encode are similar enough, one can yield a compressible chain of bits by applying an *operation* between them. Such operation, the predictor function, and the ways to encode the result from this operation (also called *residual*) have greatly changed through time. The used predictor function can be context-specific. For instance, if one is trying to compress points in space of a 3D mesh, this predictor function can be a Lorenzo Predictor (49) which estimates a point value based on its n-previous neighbours in every dimension. However, we are more interested in approaches tailored to work regardless of the nature of the floating-point data.

2.5.1.1 Delta Predictive Coding (FSD)

Back in 2000, Engelson et al. (5) introduced Delta Predictive Coding –the first true encoding for any floating-point data. Prior to this research, floating-point compression was focused on domains such as audio, geometry meshes and images. The algorithm is based on the well-known Delta encoding we already discussed in subsection 2.3.4. It converts every float into its 64-bit integer representation. In other words, the 64 bits allocated for the floating-point number, are used to define an integer. This is a clever way to enable the use of arithmetic on floating-point data without losing precision, since any operation between integers is reversible to the original integer that, in its bitwise representation, represents the original double. Afterwards, the algorithm computes the m^{th} previous differences for every value in a position further than m. To understand this let us define an array of 64-bit integers b. The first order difference ($m = 1$) for a b_i element for $i = 1...n$ is then defined as: $\Delta^1 b_i = b_i - b_{i-1}$. Then, the m^{th} order of difference is $\Delta^m b_i = \Delta^{m-1} b_i - \Delta^{m-1} b_{i-1}$. Hence, this computation of the m^{th} differences results in an array as follows: $(b_1, b_2, ...b_m, \Delta^m b_{i+1}, \Delta^m b_{i+2}, ..., \Delta^m b_n)$. An image of how this progressive differences calculation works for $m = 1$ (i.e. lookup into one previous value) can be seen in Figure 2.8. From this sequence (which occupies the same space as the original doubles sequence), we can restore the original array b losslessly since we are using integer arithmetic. Based on the assumption that such numbers in the array of differences are small, these 64-bit integers are then compressed with a type of variable-length bit-packing in which all 0's front-bits are removed and the number of stored significant bits (i.e. the bits after the 0's front-bits) are stored in 6 additional bits. Six bits are needed since at most, the significant bits would be 64 ($log_2(64) = 6$).

Compression ratio benchmarks were performed on artificially generated data, achieving a maximum of x3.68 compression ratio with differences of 10th order on blocks of 2^{16} values. As expected, the higher the m^{th} orders the higher the compression performance since the *predictions* are closer to the real value. Further testing showed that the compression algorithm was greatly affected by the *smoothness* of the data. Engelson et al. (5) defined a sequence of values as *smooth of order m* if a value at position i can be well approximated by a polynomial extrapolation of m-consecutive previous values. For instance, a sequence of values close to 0^{th} order (e.g. data generated from a function that has small and slow changes) achieved the best performance in terms of compression ratios.

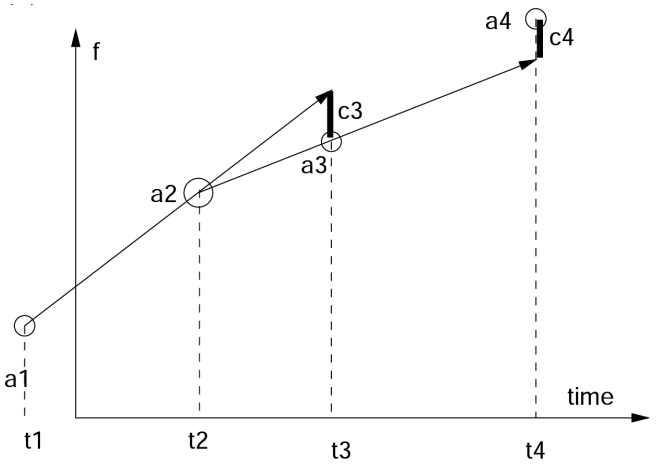

Figure 2.8: Depiction of calculating the first order differences ($m = 1$) for 64-bit integer values $a3$ and $a4$; namely $c3$ and $c4$. (Figure borrowed from Engelson et al. work (5).)

2.5.1.2 Differential Finite Context Method Predictor (DFCM)

FSD performance greatly depended on the smoothness of the data. In 2006, Ratanaworabhan et al. (6) proposed a new predictive encoding for 64-bit doubles whose novelty was to use a bitwise **XOR** as the operator between the predicted value and the value to encode instead of conventional arithmetics operators (while still being a fast operator with low latency in most ISAs (50)). The XOR operator compares two bit streams position-by-position. If the bits are the same, the result is a 0-bit, otherwise, the result is a 1-bit. This is illustrated in an example in Figure 2.9. The XOR definition is completely independent of the datatype on which the operator is applied since it works at a bit level. When two floats are close to each other in the numeric range, their sign, exponent and –possibly– the first bits of the fraction part are equal. This results in **leading zeros** when using the XOR operator. Furthermore, if their precision is similar (i.e. their radix point position) then the last bits of the mantissa could be similar. This results in **trailing zeros** when using the XOR operator.

$$0.2 \oplus 0.4$$

```
0.2: 00111111 11001001 10011001 10011001 10011001 10011001 10011001 10011010
0.4: 00111111 11011001 10011001 10011001 10011001 10011001 10011001 10011010
XOR: 00000000 00010000 00000000 00000000 00000000 00000000 00000000 00000000
```

Figure 2.9: Depiction of the bitwise XOR operator between 0.2 and 0.4 represented as 64-bit doubles. Equal bits between both numbers are highlighted in green.

An overview of the DFCM algorithm is presented in Figure 2.10. In addition to using the XOR operator, DFCM somewhat removes the dependency on the smoothness of the data since it does not rely on consecutive values being similar, rather it tries to record and identify in a hash lookup table all the predictions that have previously occurred. If a new prediction wants to be made, a hash is computed using the previous m-differences of the current value to predict. Using this hash as an index, a lookup is performed to retrieve the last predictions that followed the last m-times that the same hash was found. In other words, it tries to find predictions that happened in a similar context. Similarly to FSD, this lookup also happens up to a predefined m-order, which was fixed to 3 after experimentation (i.e. the two last predictions for the two hashed last differences are looked up). This is called a differential-finite-context-method predictor (DFCM) (51)– *the name by which the algorithm will be referred to later in the literature.* Contrary to FSD, the differences are computed with the raw doubles. For this reason, the hash index does not store all the precision of the differences, rather it only uses the n-most significant bits of the doubles and ignores the rest. By doing so, it is more likely to find recurrent patterns (e.g. a difference sequence of (0.5001 and 0.6001) are stored as an index in the lookup table as (0.5000 and 0.6000)). After experimentation, only storing the first 14 bits is enough to achieve good enough predictions. Once the algorithm has found in the lookup table the last two predictions of the same previous two differences, it uses them to calculate the current predicted value in the following way: if both previously predicted values ($dpred'$ and $dpred''$ in Figure 2.10) are similar (i.e. their first 14 bits are the same), then the new predicted value is calculated as $dpred' + (dpred' - dpred'')$. Otherwise, the predicted value is equal to $dpred'$. In such a formula, the $dpred' - dpred''$ term takes into account the drift in the difference of values, which improved the prediction accuracy and compression ratios.

With a predicted value at hand, the XOR operation is applied between it and the true value (left-most part of Figure 2.10). The yielded chain of bits is compressed with a variable length bit-pack of the first front 0's bits based on the assumptions that similar values will have the same sign, exponent and first front bits of mantissa (resulting in an XOR filled of leading 0's). However, instead of storing the exact count of 0's in 6 bits (as DPC), they encode the count of 0's divided by 4 in only 4 bits (a floor to the nearest common divisor of 4 is done to avoid decimal results). This increases the algorithm speed since the offset is at a half-byte granularity instead of a bit-granularity. The rest of the bits are stored bitpacked to only the needed amount of bits.

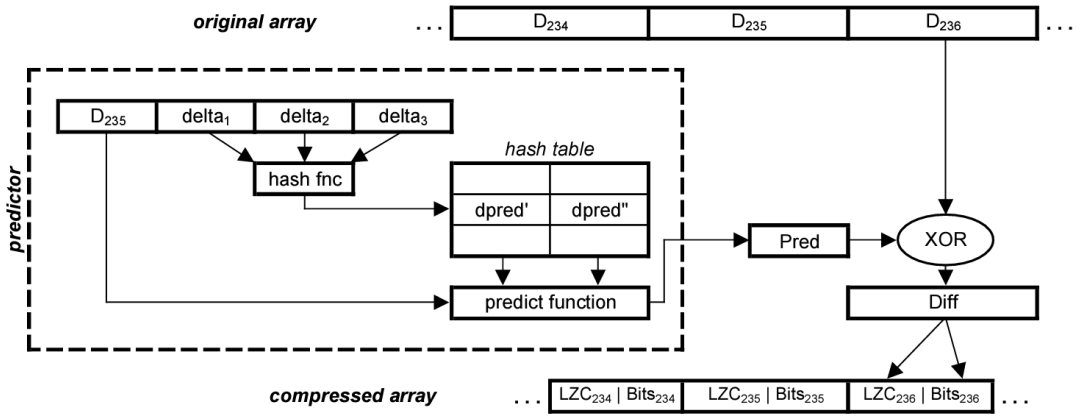

Figure 2.10: Overview of the DFCM algorithm. (Figure borrowed from Ratanaworabhan et al. work (6).)

DFCM was tested on 7 datasets from different sources of scientific data against 6 general purpose compression algorithms (bzip2, gzip, lzpx, p7zip, rar and zzip) and against FSD. In terms of compression speed DFCM achieved the fastest performance of all, being 15% faster than FSD and 257% faster than gzip. In decompression, however, it resulted to be 2.5% and 31% slower than FSD and gzip. In terms of compression ratios, it achieved the highest compression ratios in 5 out of 7 datasets, only beaten by p7zip which was orders of magnitude slower in both compression and decompression. On the other hand, it outperformed FSD in every dataset in terms of compression ratios.

2.5.1.3 Fpzip and Pzip

Later on in the same year in which DFCM was developed, Lindstrom & Isenburg developed fpzip (46) (also referred to as PLMI). Fpzip diverges from the DFCM idea of XORing. It first predicts subsequent values using already encoded values with a 1D generalization of the Lorenzo predictor (49). Then, both the predicted and true values are transformed to their 64-bit integer representation. Finally, residuals are computed between both values using an arithmetic subtraction. The novelty of fpzip is that these residuals are then further compressed using a combination of entropy coders and raw bit storage. Fpzip was benchmarked on 2D and 3D grids data (despite of this, a generalization for 1D arrays of floating-point data is given). Fpzip achieved better compression ratios than the ones of DFCM on these types of datasets. However, in terms of speed, it is slightly slower than DFCM. In 2019, Cayoglu et al. (7) improved on fpzip to develop pzip. Pzip introduces the

XORing idea into fpzip to calculate the residual between the predicted and true values. However, before the XOR, the predicted value is adjusted with an operation they defined as *shift*. A *shift* is a sequence of bit flips that tries to maximize the count of 1's which follows the initial count of 0's. Afterwards, these counts of 0's and 1's are rearranged and encoded with an entropy encoder. Pzip demonstrated to be on average 10% better in compression ratios and 6 times faster than fpzip. Unfortunately, the benchmarks were only done against fpzip despite the fact that many more floating-point schemes surfaced in years prior to this work as we will see in the next subsections. Both fpzip and pzip algorithms are depicted in Figure 2.11.

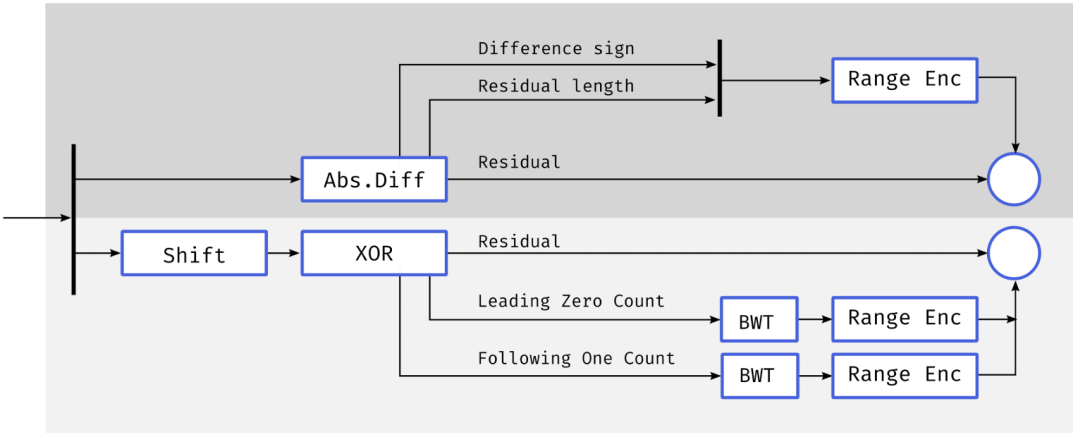

Figure 2.11: Overview of Fpzip (top) and Pzip (bottom) algorithms. (Figure borrowed from Cayoglu et al. work (7).)

2.5.1.4 FPC

Shortly after DFCM, Burtscher & Ratanaworabhan developed FPC (8), which achieved orders of magnitude better compression and decompression speeds while still improving on compression ratios compared to both DFCM and FSD. FPC is depicted in Figure 2.12. FPC followed the same idea of DFCM: sequentially predicting each value to encode, XOR-ing the predicted value with the value to encode and compressing the XOR result with compression of the front 0's bits. The main architecture-wise difference is that FPC uses *two* different predictors instead of one to improve prediction accuracy and therefore improve compression ratios. Then, a *selector* chooses the best prediction among both predictors (i.e. the one that shares the most front bits with the value to encode). The first predictor is a *fcm* (52) while the second is an *dfcm* (51). Similarly to the DFCM algorithm, both

predictors are hash tables indexed at the m-previous values. The main difference between fcm and dfcm predictors is that the *dfcm* table is indexed on the differences of the m-previous values (similar to the DFCM algorithm) and the *fcm* table is indexed on the raw m-previous values. On FPC, doubles are converted to their 64-bit integer representations (similar to FSD) to use integer arithmetics. Similarly to the DFCM algorithm, the hashing function in FPC also cut the numbers to their first 14 bits to eliminate the random mantissa bits.

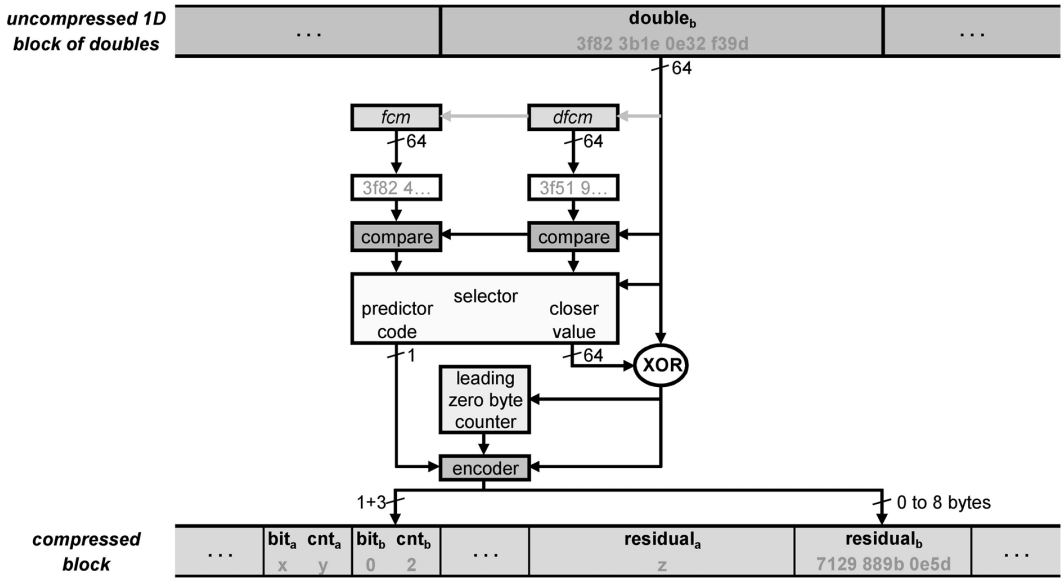

Figure 2.12: Overview of FPC. (Figure borrowed from Burtscher & Ratanaworabhan work (8).)

Since two predictors are used at compression time, the algorithm has to be able to detect which one was used to perform the decompression. Hence, when encoding the XOR result, an additional control bit is added to the compressed value. This additional bit is compensated by only using 3 bits to store the XORed value front 0's count (contrary to the 4 bits used in DFCM). Although 3 bits only allow us to store a maximum count of 32 leading zeros (a count from 1-8 times 4), 4 bits are deemed to be unnecessary since in practice, leading 0's count bigger than 32 rarely happens. If it does, these extra 0's are stored alongside the residual (i.e. trailing bits; see bottom of Figure 2.12). Finally, as shown in Figure 2.12, FPC stores data in two separate segments: metadata and residuals. The metadata segment contains blocks of half-bytes comprised of 1 control bit (i.e. predictor used) and 3 bits for the count of leading zeros / 4. The residuals segment contains the raw trailing bits of the compressed number. Each block is stored with a header that

determines how many values are compressed in a block. This separation of the compressed data into two segments instead of interleaving them substantially improves decompression speed since all metadata can be sequentially read without any bit-level offset corrections if read as a pair (effectively 1 byte).

FPC was evaluated on 4 different computer architectures (i.e. Alpha, Athlon, Itanium and Pentium) using 13 different datasets of 3 different natures (i.e. observational scientific data, numeric simulations and parallel messages). FPC was tested against two general-purpose compressors (i.e. BZIP2 and GZIP) and three compressors for floating-point data (i.e. FSD, DFCM and Fpzip (referred to here as PLMI)). In terms of compression speed measured in throughput, FPC is orders of magnitude better than any other compressor in every architecture (8 to 300 times faster in compression and 9 to 100 times faster in decompression). Furthermore, by using predictor tables that fit into L1 Cache, FPC achieved a throughput of 84 million doubles per second on the 1.6GHz Itanium architecture.

In terms of compression ratio, the performance of FPC widely depends on the maximum size set for the predictors' hash tables. For instance, with a size of 1MB, FPC achieved the highest compression ratio than any other algorithm in 4 datasets and individually it achieved compression ratios between x1.08 and x15.05. Although competitive, FPC fails to be on par with Gzip and Bzip2 on some datasets with a high entropy in its values. In general, FPC performed better than DFCM. This is due to DFCM focusing on specializing one complex predictor. On the other hand, FPC has the advantage that it uses two simpler predictors, and when one predictor performs poorly, the other is able to help obtain a better-predicted value. Increasing the table size in FPC helps to achieve more compression ratios, however, it affects the speed of [de]compression.

2.5.1.5 Sprintz

Striving for a predictive scheme that can run in low-memory environments such as IoT devices while still achieving improved compression ratios and decompression speed Blalock et al. (53) developed Sprintz. In contrast to previous approaches, Sprintz gets its speed from processing small blocks at-a-time. Furthermore, its improved compression ratios stem from the use of FIRE (Fast Integer REgression) as a predictor, instead of Delta Encoding; and the use of other LWC methods to further compress the residuals.

In a high-level overview, Sprintz compresses data in 4 steps:

- Floating-point values are converted to their integer representation.
- The FIRE predictor is used to forecast a value based on previous values.

- The residuals (subtraction) between the predicted value and the true value are bit-packed to only the necessary amount of bits (SIMDized).
- RLE is used to further compress such residuals (SIMDized).
- Finally, metadata (i.e. bits used on bit-packing and run lengths) and residuals are encoded with Huffman codes.

FIRE is a novel predictor introduced in the same work and despite being more computationally expensive than Delta, it achieved more accurate predictions, hence smaller residuals. FIRE is a linear regression model in the form of: $x_i = x_{i-1} + \alpha x_{i-1} + \alpha x_{i-2} + \epsilon_i$; being x a true value of the data to encode and ϵ a term that accounts for noise. Essentially, a model is built for every block of data.

Sprintz was tested on around 100 datasets from 5 time series repositories of floating-point data, and compared against some LWC (e.g. SIMD-BP128 and FastPFOR) and some GPC (Snappy, Zstd and LZ4). As one may have noticed, these LWCs used for comparison are for the integer type. Sprintz was compared against such methods since it only works by converting floats to integers in the first step of compression. Sprintz achieved on average higher compression ratios than all of the algorithms (LWCs and GPCs) it was compared against, ranging from 2x - 6x compression ratios. However, in terms of decompression speed, it was orders of magnitude slower than SIMD-BP128 and FastPFOR. In addition to that, Sprintz also showed to be slower than some GPCs such as LZ4.

2.5.2 XOR Schemes

Motivated to achieve even higher throughput in the compression of floating-point data within a streaming context, Pelkomen et al. (30) re-evaluated the need for a predictor function to obtain a similar value to the value to encode. Their key idea was that in certain contexts such as time series, using the immediate previous value in a stream of data to operate with the current value to encode works as well as using a predictor function. This assumption motivated the development of Gorilla which at the same time, has recently motivated countless schemes to compress floating-point data which we explore in the next subsections. All of these schemes value the simplicity and power of the XOR operator when trying to make floating-points compressible, thus making it the core of their algorithms. This effectively allowed researchers to focus on clever ways to optimize the way in which the XORed chain of bits is compressed, instead of focusing on developing better predictive functions. Despite their implementation being relatively recent, none of these encodings

implements a SIMD-friendly variation or counterpart, and neither is tailored for vectorized execution. The latter is due to most of these algorithms having a strong use of control structures and data dependencies which inhibit SIMD implementations.

2.5.2.1 Gorilla

The motivation behind Gorilla was to create an algorithm able to compress floating-point data incoming in a streaming fashion at a high throughput rate. In such a setting a predictor function introduces significant overhead and the need to look at previous values. The novelty of Gorilla lies in the fact that no predictor function is needed to generate a value that yields a nice compressible chain after the XORing process. In Gorilla, the predictor function is replaced in favour of using the *immediate previous value*. This idea was based on the observation that on time series data, similar values (i.e. same sign, exponent and first bits of the mantissa) are stored close-by given the temporal commonalities of consecutive values. On time series, each floating-point value v_{i+1} is recorded further in time than value v_i. Hence, a **time series** can be defined as a sequence of observations ordered increasingly by time. On the other hand, in non-time series data, this temporal property is not present. For instance, this can result in a higher entropy between consecutive values. Gorilla's original compression algorithm encodes a pair of timestamp and floating-point values. In the context of our LWC study, we focused on the part of the algorithm that compresses floating-points rather than timestamps since both of these may exist as standalone algorithms.

Each block encoded by Gorilla leaves the first value uncompressed, and consequent values are encoded by applying a bit-wise XOR between its immediate previous value and using the following variable length encoding scheme for each XORed chain of bits:

- If the XORed chain of bits is 0 (i.e. perfect XOR; equal values)
  - Write "0" bit [case #1 in Figure 2.13]
- Otherwise,
  - Write "1" bit
  - If the number of leading zeros and trailing zeros is equal or higher than the previous XORed chain leading zeros and trailing zeros: (i) write "0" bit and (ii) write the significant bits (i.e. bits after the leading zeros) [case #2 in Figure 2.13]

– Otherwise: (i) write "1" bit, (ii) write the number of leading zeros (in 5 bits, due to $log_2(32)$ being the maximum amount of leading zeros supported), (iii) write the number of significant bits (in 6 bits, due to $log_2(64)$ being the maximum amount of possible significant bits) and (iv) write the significant bits themselves. [case #3 in Figure 2.13]

These cases are visually depicted in Figure 2.13. There are a couple of remarks we would like to highlight from Gorilla: 1) The significant bits of an XOR result are the bits that are not deemed as leading or trailing zeros. 2) The previous number of leading and trailing zeros are stored as an algorithm state every time we reach the right-most branch (Figure 2.13). As one may have noticed, this means that every time the current XOR leading or trailing zeros are smaller than the previous one, these state values become progressively smaller; hindering the performance if the analyzed blocks are big. For instance, if at one point the previous leading zeros or trailing zeros are equal to 0, there is no point of return until the next block starts. The following values to encode will fall into the worst-case scenario (case #2 in Figure 2.13) in which all bits would be stored as significant. The latter can be seen as being stuck in a local optimum.

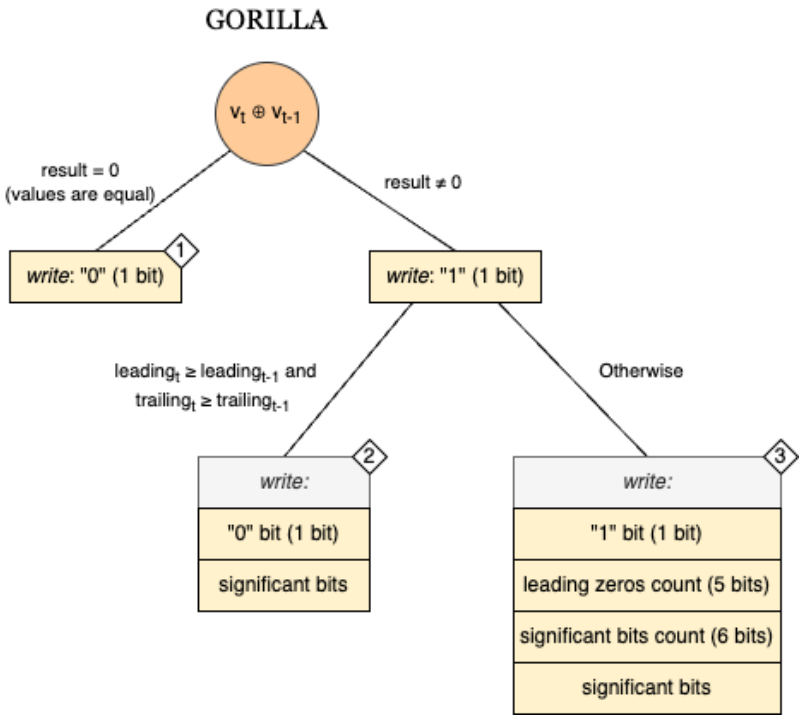

Figure 2.13: Overview of Gorilla algorithm. Case identifiers are enclosed in diamonds. (Figure format inspired by Liakos et al. work (9).)

Unfortunately, the Gorilla paper does not perform benchmarking against previous schemes. However, Gorilla was deemed to be faster than any predictive scheme on both compression and decompression, since encoding and decoding are achieved using a simple XOR with the immediate previous value instead of tuning and running a predictive function.

Recently, Wang et al. (10) proposed a small optimization to Gorilla to improve compression ratios in such cases when Gorilla get stuck in a local optimum. In order to do so, the block of compressed values is scanned and case #3 is preferred in favour of case #2 when the state of previous leading and trailing zeros is already low but a further XOR results in more zeros that could potentially save space in the further next cases. Essentially, this process serves as a recovery for Gorilla when its state of leading and trailing zeros is close to 0 and compression is not effective anymore since case #3 overwrites the state of the algorithm. Figure 2.14 shows an example in which favouring case #3 instead of case #2 despite the amount of leading zeros being higher than the previous one, results in higher saves for the latter values.

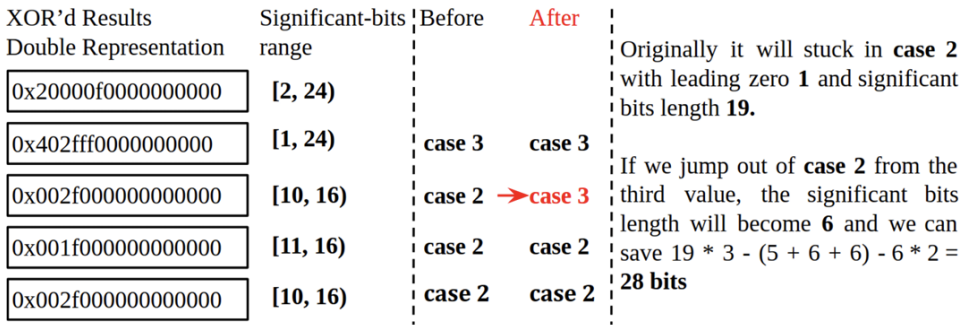

Figure 2.14: Depiction of local optimum escape in gorilla (Figure borrowed from Wang et al. work (10).)

2.5.2.2 TSXor

In 2021, Bruno et al. (11) followed the line of XORing floating-points and developed TSXor. TSXor authors observed that floating-point values which look similar in their decimal representation (i.e. their human-readable representation), may not have a similar bitwise representation. An example of this is seen in Figure 2.15.

Hence, the key idea of TSXor is to look for the best value to XOR within a window of previous values. The best value to XOR is defined as the value that yields the most amount of leading and trailing zero bits after XOR. This lookup window is set to the previous 128

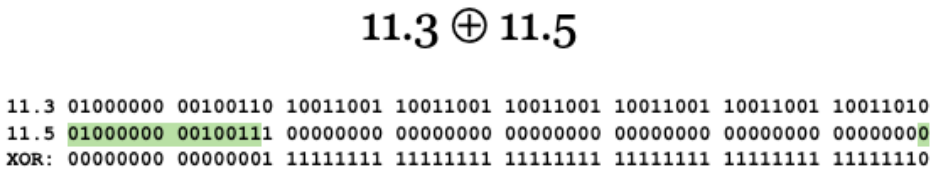

Figure 2.15: Depiction of the bitwise XOR operator between 11.3 and 11.5 doubles. Equal bits between both numbers are highlighted in green.

values to comfortably fit the reference index into one byte (we can store an index from 0 to 127 in 7 bits). From here, 3 different compression cases are defined:

- **Reference:** If a perfect XOR is found (i.e. equal values), write the previous index position in 1 byte. [case #1 in Figure 2.16]
- **XOR:** If the sum of the number of leading zeros and trailing zeros is bigger or equal to 2; write (i) the previous index position + 128 (1 byte), (ii) the number of leading bits (in 4 bits) and the number of trailing bits (in 4 bits) packed in 1 byte and (iii) write the significant *bytes*. [case #2 in Figure 2.16]
- **Exception:** Otherwise, write (i) 255 as an exception code in 1 byte and (ii) the uncompressed double (8 bytes). [case #3 in Figure 2.16]

TSXor algorithm cases are depicted in Figure 2.16. It is important to note that TSXor works in a byte-aligned way. This efficiently increases the algorithm speed since there are no bit-level offsets to manage in compression or decompression procedures. It is also important to highlight that the three cases are differentiated at decompression time by the first byte. In the case of *Reference*, this first byte ranges from 0 to 127. In the case of *XOR*, the addition with 128 makes this control byte always have a '1' as the first bit. Finally, the special value 255 is used to encode exceptions. Note that in the case of a high number of exceptions, TSXor may lead to negative compression (i.e. compressed data is bigger than uncompressed) due to the extra control byte to represent the special value 255 alongside the uncompressed value.

TSXor was tested in 7 real-life floating point datasets. In terms of compression ratios, TSXor beats both FPC and Gorilla in most of the datasets with compression ratios ranging from x1.30 and x6.4. In such datasets, on average 7.8% of the values were encoded as exceptions, 54.3% as reference and 37.9% as XOR. However, in two of the seven datasets, the exceptions percentage goes as high as 23%. In terms of decompression speed, TSXor is

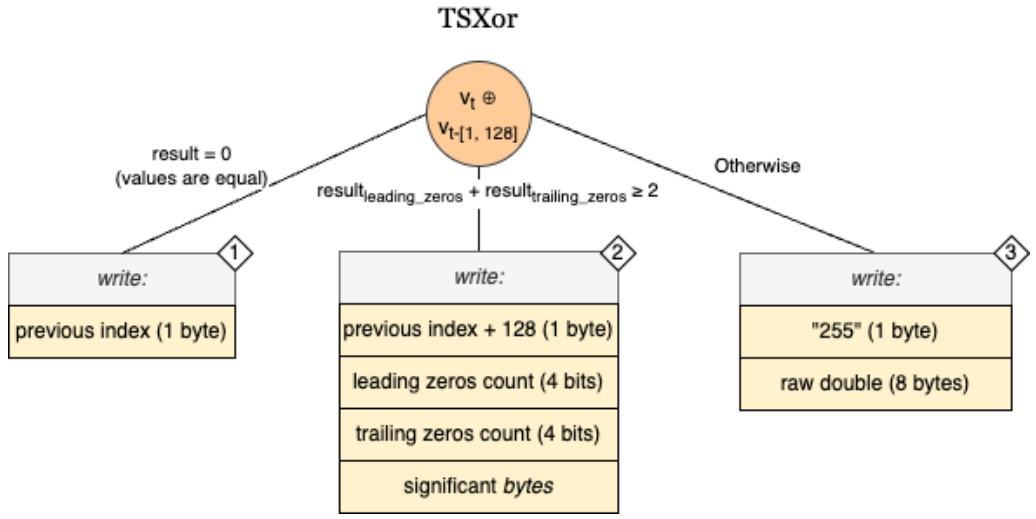

Figure 2.16: Overview of TSXor algorithm. Case identifiers are enclosed in diamonds. (Figure format inspired by Liakos et al. work (9).)

1.9 and 4.2 times faster than Gorilla and FPC respectively mainly due to its byte-aligned way of storing information. In compression speed, however, TSXor falls behind both FPC and Gorilla mainly due to the window lookup for the proper value to XOR. In Figure 2.17 we can see how different window size creates a tradeoff between slower compression speed (due to lookup) and higher compression ratios while decompression speed remains the same.

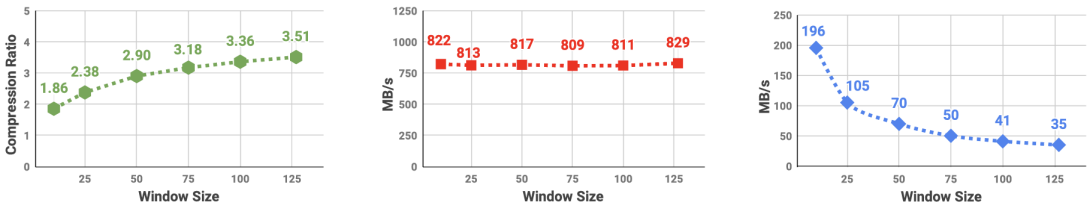

Figure 2.17: TSXor trade-offs when using increasingly window sizes for the reference value on compression ratio (left), decompression speed (middle) and compression speed (right) (Figure borrowed from Bruno et al. work (11).)

We can notice how TSXor combines ideas from both predictive schemes (e.g. maintaining a lookup table to search for the best value to operate with) and purely XOR schemes (e.g. packing both leading and trailing zeros).

2.5.2.3 Chimp

In contrast to Gorilla, which only leverages Facebook's time series data, Liakos et al. (9) analyzed 19 real datasets of floating point data of different natures and decimal precision to develop Chimp. 14 of these were data in a time series fashion and 5 of these were data from non-time series. For instance, these last 5 datasets are more representative of doubles stored in classical database workloads. These datasets contain data from temperature measurements, monetary data (i.e. stocks, prices), coordinates and scientific measurements. Furthermore, each dataset had different decimal precisions, ranging from 1 up to 17. This variety of datasets is remarkable since it is the first research work on floating-point compression which did not focus on a specific data domain to test its compression algorithm. Let us remember that up until now, predictive schemes have focused on scientific/simulations data, and Gorilla focused on very specific data from Facebook's systems.

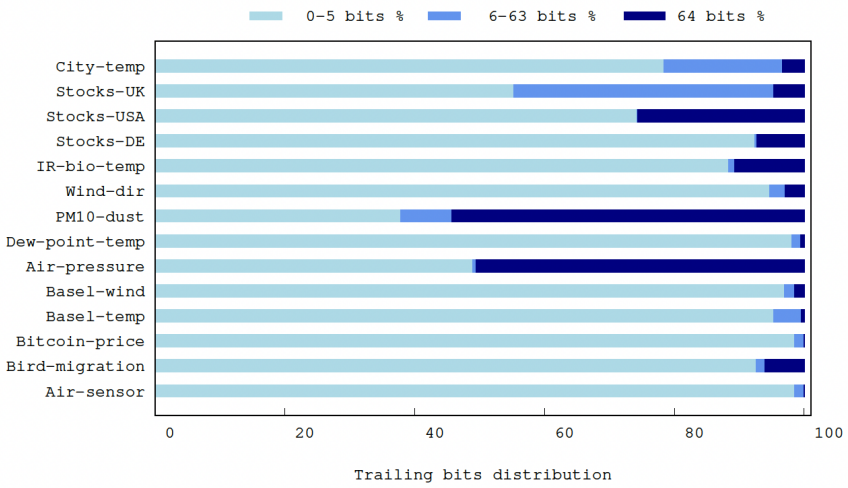

Figure 2.18: Distribution of trailing zeros from XORing with previous value on real-life datasets analized to develop Chimp. (Figure borrowed from Liakos et al. work (9).)

When analyzing these datasets output when XORing every value with its previous value, Liakos et al. (9) observed that in the case in which two values are not identical, their XORed chain of bits is not likely to have a large number of trailing zeros. In fact, the distribution shows that there is a high probability that the resulting XORed values have less than six trailing zeros and only a few cases of 6 to 63 trailing 0's bits and lesser of 64 bits (equal XOR). This distribution is depicted in Figure 2.18. This means that reserving 6 bits to denote the number of trailing zeros (as Gorilla) is usually worse than storing the bits itself. On the other hand, they observed that such an XORed chain of bits exhibits a considerable

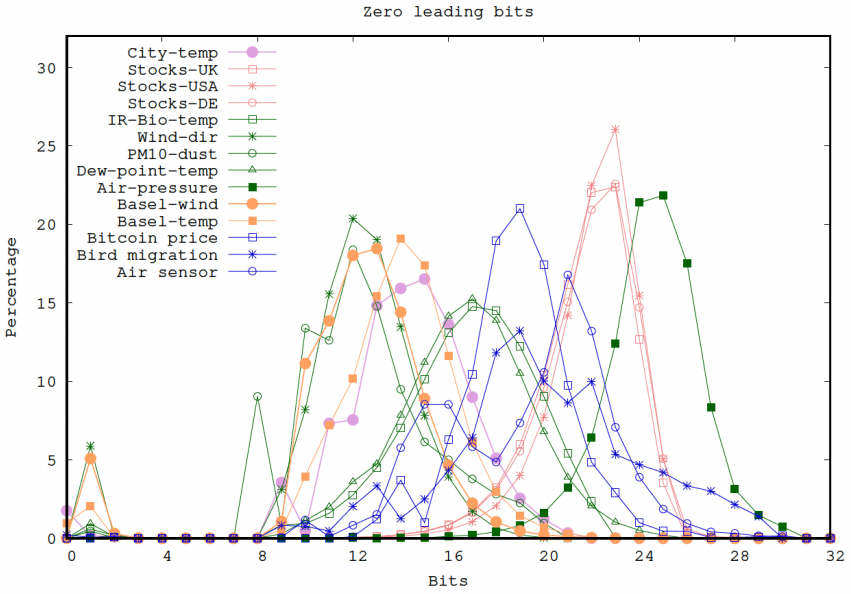

Figure 2.19: Distribution of leading zeros from XORing with previous value on real-life datasets analized to develop Chimp. (Figure borrowed from Liakos et al. work (9).)

number of leading zeros instead. For instance, there are very limited cases in which the leading zeros range is between 1-7 bits. Most of the leading zeros distributions are located between 9 and 24 bits (Figure 2.19).

These observed trailing and leading zeros properties make most real-world scenarios quickly fall in a local optimum which leads to the *worst* case scenario of Gorilla, in which almost no compression is achieved (case #2 of Figure 2.13). Leveraging these properties, Liakos et al. developed Chimp. Chimp (9) refined Gorilla by exploiting previously mentioned properties of the bit-chains yielded by the XORing process in time series data. Chimp uses a 2-bit control flag to distinguish four compression cases mainly distinguished by the amount of trailing zeros in the XORed chain. The first value of the block to compress is stored uncompressed. Consequent values are stored based on the following rules:

- If XOR with previous value has more than 6 trailing zeros ($> log_2(64)$)
  - Write "0" bit.
  - If values are identical (i.e. perfect XOR), write "0" bit. [case #1 in Figure 2.20]
  - Otherwise, write (i) "1" bit, (ii) write the number of leading zeros (in 3 bits; we explain why only 3 bits are used later), (iii) write the length of the significant bits (in 6 bits) and (iv) write the significant bits themselves. The significant

bits are the bits of an XOR bit-chain which are neither trailing zeros nor leading zeros. Hence, the number of significant bits can be calculated as follows: $64 - leading\_zeros\_count - trailing\_zeros\_count$. [case #2 in Figure 2.20]

- If XOR with previous value has less or equal to 6 trailing zeros ($\leq log_2(64)$)
  - Write "1" bit.
  - If the number of leading zeros is equal to the previous XORed value leading zeros: (i) write "0" bit and (ii) write the non-leading XORed bits (i.e. significant bits). [case #3 in Figure 2.20]
  - Otherwise, (i) write "1" bit, (ii) write the number of leading zeros (in 3 bits) and (iii) write the non-leading XORed bits (i.e. significant bits). [case #4 in Figure 2.20]

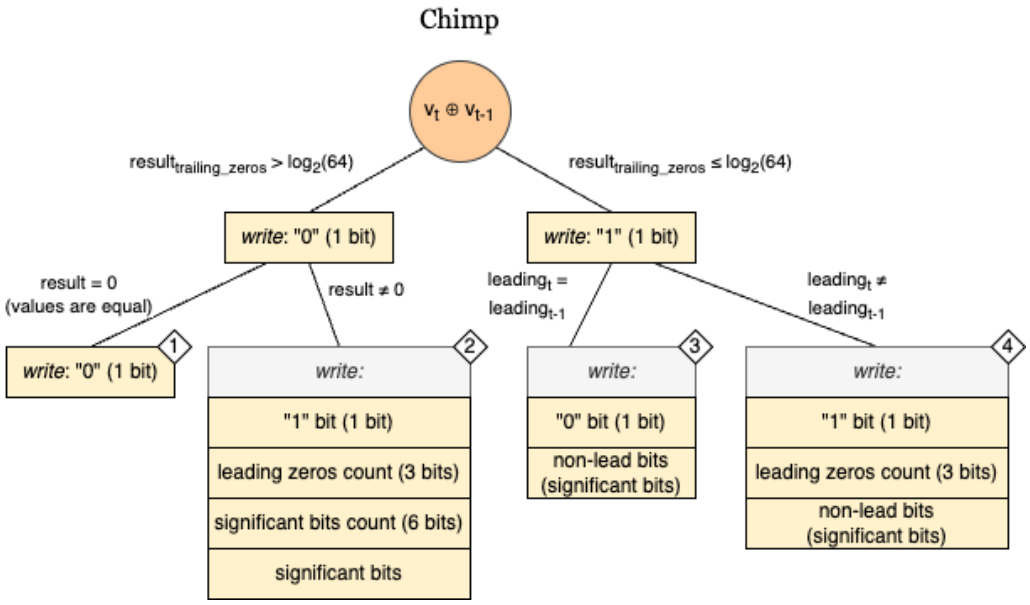

Figure 2.20: Overview of Chimp algorithm. Case identifiers are enclosed in diamonds. (Figure format inspired by Liakos et al. work (9).)

Chimp encoding cases can be seen in detail in (Figure 2.20). There are a couple of remarks regarding the Chimp algorithm: 1) The significant bits of an XOR result are the bits that are not deemed as leading or trailing zeros. 2) In contrast to Gorilla, Chimp stores the leading zeros count in 3 bits instead of 5. This is an observation based on the observed skew of the distribution of leading bits (Figure 2.19). For instance, values of leading zeros between 1 and 8, or 24 and 32 rarely happen. Hence, these 3 bits are used

to map 8 exponentially decaying steps of possible leading zero counts: 0, 8, 12, 16, 18, 20, 22 and 24. In other words, it serves as an optimal floored rounding of the number of leading zeros. 3) In contrast to Gorilla, the previous number of leading zeros stored in the algorithm state is not always decreasing; it is updated every time the algorithm lands in case #4 and it is reset on cases #1 and #2.

2.5.2.4 Chimp128

Chimp was jointly developed with a variant called Chimp128 in which, very similar to TSXor, the algorithm looks into the previous 128 values in order to find the *most suitable* value to XOR at the expense of 7 additional bits to store the position of this value (7 bits due to $log_2(128)$). Note that it is remarkable that the TSXor paper was not cited by the Chimp paper. The use of a buffer to look into the previous 128 values effectively skew the distribution of trailing bits towards more than 6 bits, and even towards perfect XOR (i.e. 64 trailing zeros–equal values). However, by maintaining Chimp's 4-cases, this previous index is only used if it is useful to achieve compression. That is, only if the number of trailing zeros resulting from the XOR with any of the previous 128 values is higher than the number of bits needed to store the index itself (7 bits for a buffer of 128 values), plus the number of bits to specify the number of significant bits. The latter slightly changes both left-most cases of Chimp compression as follows (Figure 2.21):

- If XORed value trailing zeros surpass the space needed to store the previous value index plus the number of significant bytes ($trailing\_zero\_count > log_2(128) + log_2(64)$)
  - Write "0" bit
  - If values are identical (i) write "0" bit and (ii) write the previous index (in 7 bits)
  - Otherwise, (i) write "1" bit, (ii) write the previous index (in 7 bits), (iii) write the number of leading zeros (in 3 bits), (iv) write the number of significant bits (in 6 bits) and (v) write the significant bits themselves.
- Otherwise ($trailing\_zero\_count \leq log_2(128) + log_2(64)$)
  - Write "1" bit
  - If the number of leading zeros is equal to the previous XORed value leading zeros: (i) write "0" bit and (ii) write the non-leading XOR bits

– Otherwise: (i) write "1" bit, (ii) write the number of leading zeros (in 3 bits) and (iii) write the non-leading XORed bits.

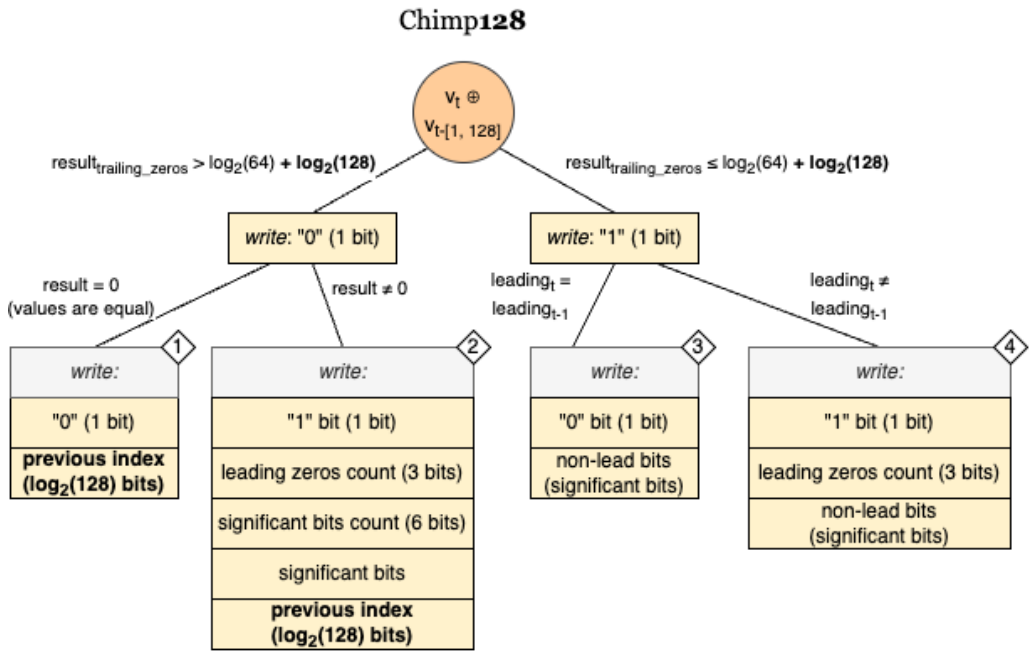

Figure 2.21: Overview of Chimp128 algorithm. Case identifiers are enclosed in diamonds. Text highlighted in bold are the main differences with Chimp. (Figure format inspired by Liakos et al. work (9).)

In terms of evaluation, Chimp128 proved to be substantially better than FPC and Gorilla with 39% and 40% higher compression ratios respectively (9). Moreover, Chimp128 achieved higher compression ratios than GPC methods as Snappy and LZ4; while still underperforming against Zstd. In terms of speed, Chimp128 is on par with Gorilla, faster than FPC and faster than every general purpose approach in both compression and decompression (9). Chimp128 buffer lookup was tested also with 16 values instead of 128. Such configuration did not achieve substantial improvements over only looking at the previous value in some datasets. Interestingly enough, Chimp128 showed that previous-value XORing approaches also work well for non-time series data. Based on these results, Chimp128 was deemed to be the de facto alternative for floating-point compression.

2.5.2.5 Patas

In order to improve Chimp *decompression speed*, DuckDB Labs developed Patas (31). The goal was to get a variant of Chimp128 faster at [de]compression. Patas achieves this by

re-engineering Chimp128 into an algorithm with a single encoding mode (i.e. no if-else statements) that stores bits in a byte-aligned way (i.e. less CPU work). Patas is depicted in Figure 2.22. Due to its single encoding mode (instead of four), Patas does not encode control bits. Furthermore, compressed data is divided into two close-by segments: data and metadata (red and yellow in Figure 2.22). For every value, Patas encodes a metadata block of exactly 2 bytes (i.e. 16 bits) containing: (i) the previous value index (in 7 bits), (ii) the XORed value number of significant bytes (in 3 bits due to $log_2(8)$) and (iii) the number of trailing zeros (in 6 bits). These metadata blocks are stored continuously. At decompression time, the number of leading zeros can be known from the number of significant bytes and the number of trailing zeros available in the metadata. On the other hand, the data block contains the significant *bytes* of the XORed value. Note that these are bytes, rather than bits. Hence the unpacking of data is byte-aligned; which makes it faster since there are no bit-level offsets that require shift operations. In addition to this, the uniformly packed metadata (also byte-aligned) makes decompression faster since there is no need to maintain and update multiple pointers to access each array of metadata; unlike Chimp128. Figure 2.23 depicts how Patas compressed data is physically stored in DuckDB.

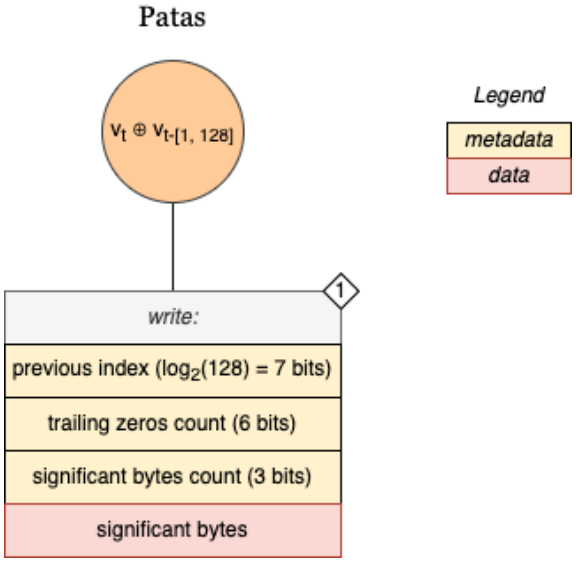

Figure 2.22: Overview of Patas algorithm. (Figure format inspired by Liakos et al. work (9))

Patas trades compression ratio for an x3-4 speed improvement at decompression time compared to Chimp128. In the context of analytical databases such as DuckDB decompression speed is important for obtaining fast query results. Note that Patas may easily lead to negative compression (i.e. compressed data bigger than uncompressed) since, no

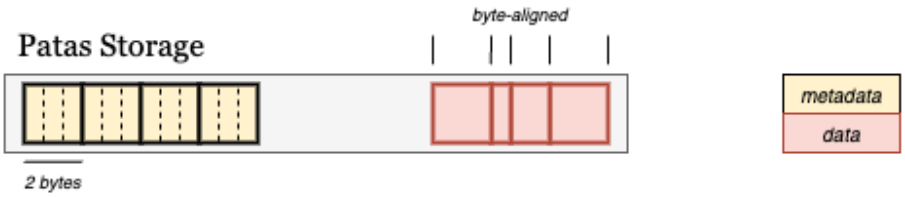

Figure 2.23: Overview of how DuckDB stores Patas compressed data.

matter what, every encoded value significant bytes (which could be 8 bytes) comes with its block of 2 bytes of metadata. For this reason, Patas is always going to be at a disadvantage against Chimp, TSXor or Gorilla if perfect XORs are found.

2.5.2.6 Elf

Most recently, Li et al. (12) proposed a new scheme called Elf (12); which trades both compression and decompression speed for increased compression ratios. Elf is a LWC method for streaming floating-point compression which serves mainly as a pre-processor of the doubles. Elf erases bits from the mantissa at encoding time to make the XOR result more compressible. After erasure, any floating-point compression method can be plugged in to encode the *less random* doubles. After decoding, Elf losslessly reconstructs the original double. For instance, in Figure 2.24 we present an example of how Elf is able to erase 44 bits of the mantissa of the double 3.17. Afterwards, this double XORed to its previous value resulting in a chain of bits filled with trailing and leading zeros (Δ' in Figure 2.24). The latter is a chain from which Gorilla and Chimp can achieve higher compression ratios. As Elf architecture depicts in Figure 2.25, these erasure and restoring procedures are agnostic of the used LWC method. In other words, any state-of-the-art method can be plugged into Elf's architecture. For example, in Figure 2.25, one can replace XOR_{cmp} and XOR_{dcmp} with Gorilla_{cmp} and Gorilla_{dcmp}.

Elf challenges are: How to erase bits from the mantissa, and more importantly, how to recover them at decoding time without information loss. And second, how to do it *fast*. The idea behind erasing bits from a double n is to find a number δ between 0 and an inverse power of 10 ($0 < \delta < 10^l$), being l the decimal place count of n, such that the number obtained from $\delta - n$ result in a *good* XORed result with a previous value n'. We define a *good* XORed result as a chain of bits with a high number of leading 0's bits. At decompression, the original n can be recovered from the XOR result, and both δ and n'. Such pair (n', δ) is satisfied by many combinations. For instance, for $n = 3.17$, such pairs could be

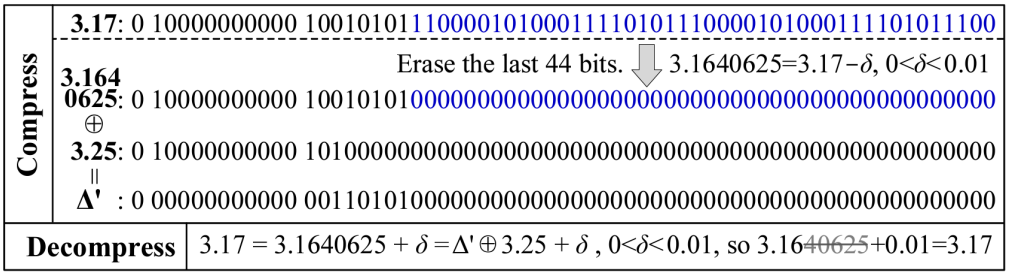

Figure 2.24: Elf eraser and restorer applied to the double 3.17 (Figure borrowed from Li et al. work (12))

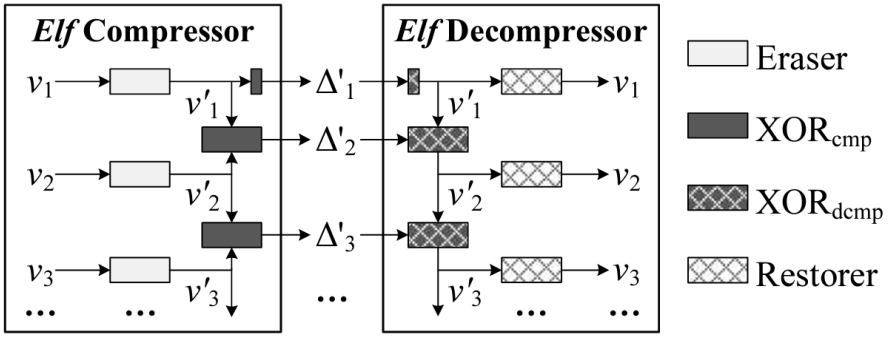

Figure 2.25: Overview of Elf architecture (Figure borrowed from Li et al. work (12))

$(3.17, 0)$, $(3.169999837875366, 0.000000162124634)$ or $(3.1640625, 0.0059375)$. From these three possible pairs, the latter has the n' with the most amount of trailing zeros. This search can be done with brute force (at most 52 trials; that is, one trial per each position of the mantissa). However, by knowing the decimal place count of n, namely α; Elf is able to find the best n' by simply erasing the mantissa bits after the bit at position α of the mantissa. The latter is formally demonstrated in (12). This effectively removes the need to compute and store δ, since n can be recovered only by knowing α and n'. If we construct a procedure called *LeaveOut* which removes digits from a double decimal representation, we can recover n at decompression time with Equation 2.2:

$$n = LeaveOut(n', \alpha) + 10^{\alpha} \tag{2.2}$$

Additionally, Elf uses a 1-bit flag to mark if a number n has gone through the erasing procedure or not. The latter is to handle the edge cases of 0, Inf and NaN.

Elf experiments were carried out on 22 datasets. Of which 14 represent time series and 8 represent non-time series. 19 of these were the same ones already tested in the development

of Chimp[128]. The 3 new datasets represent vehicle charges with a maximum precision of 3 decimals and latitude and longitude coordinates with 6 and 7 decimal precision respectively. Elf achieves significant improvements in compression ratio against previous LWC methods for floating points. Beating FPC, Gorilla and Chimp with an average relative improvement of ≈51% more compression ratios. Against Chimp128, this improvement is ≈10%. Elf achieves an average relative improvement of 30.2%, 7.5% and 27.5% against LZ4, Zstd and Snappy respectively for time series datasets. On the other hand, it achieves an average relative improvement of 18%, 3.5% and 16.7% on non-time series datasets. Regarding speed, Elf is the slowest of all the LWC methods on both compression and decompression time, demonstrating a very strong tradeoff between compression ratios and speed. Being around x4 slower than Gorilla and 3x slower than Chimp at compression time, and x2 slower in decompression when compared to both.

It is important to note here that the used Zstd implementation is from Apache Hadoop, rather than the more optimized Facebook's version. This was also the case when evaluating Chimp and Chimp128.

2.5.3 Decimal-based Schemes

Fast and well-known encodings such as Delta or FOR cannot be applied to floating-point data since they are based on arithmetic which is prone to rounding errors when operating with floating-point values. However, there is a branch of research that has focused on trying to convert floating-point numbers into integers to make them compressible with these encodings. In fact, we have already previously introduced Delta Predictive Coding, whose first step is to transform floating-point numbers into an integer representation. This transformation is based on the bitwise representation of the doubles, which can also effectively represent a 64-bit integer. Afterwards, a delta encoding is applied between this integer and a predicted value. However, the randomness of the mantissa makes the 64-bit integer representation of *similar looking* doubles very different. Thus, without a predictor, trying to achieve compression using Delta or FOR is not effective. For instance, the doubles 2.5 and 3.5 have a raw decimal integer value of 461**2811918334230528** and 461**5063718147915776**. In this section, we review efforts to compress floating-point data based on their visible decimal representation rather than their bitwise representation.

2.5.3.1 BUFF (BoUnded Fast Floats compression)

Liu et al. (13) observed that in many real-world data stores, floating-point data is bounded within a specific range and precision. If such precision is known, one can just ignore bits from the mantissa at compression time. For instance, we can take the single precision float 3.14: 10000000 10010001111010111000011. Given a precision of two decimals, we can eliminate mantissa bits: 10000000 10010001100000000000000. This would be equal to 3.13671875. However, since we know that the original number had a precision of 2 decimal places, we can recover it by rounding without losing the original representation. From empirical analysis, the number of bits needed for a specific precision was computed (Figure 2.26). Such trailing zeros create opportunities for compression; especially if one knows that such precision is bounded.

Precision	1	2	3	4	5	6	7	8	9	10
Bits needed	5	8	11	15	18	21	25	28	31	35

Figure 2.26: Bits needed to achieve a decimal precision in a float (Figure borrowed from Liu et al. work (13))

Based on this observation, BUFF compress floating-points by splitting their integer and fractional *decimal* parts (Note that here we are *not* talking about exponent and mantissa but integer and fractional parts of the decimal representation of the number). BUFF works in the following way:

- The *exponent* of the float (namely E) is obtained as an integer (marked in green in Figure 2.27). In the example presented in in Figure 2.27, the exponent 4 is obtained from $E - 127; E = (10000011)_2 = (131)_{10}$. Based on this exponent value, and using the single precision floating-point definition, one can determine the bit offset to the decimal point of the real number. Hence, we are able to determine which bits of the floating-point bitwise representation, are used to represent the integer part of the real number. An important observation to understand this is that the integer value of a floating-point (i.e. the number at the left of the decimal point) is defined by the first n bits of the mantissa. Being $n = E - 127$ for floats and $n = E - 1023$ for doubles. Hence, we are able to only retain that integer part of the number (marked in blue in Figure 2.27).

- The rest of the bits are truncated to the number of bits needed to maintain a given precision (Figure 2.26). In the example provided in Figure 2.27, we maintain 15 bits (marked in bold orange) since the precision is of 4 decimal places.
- If the range of the values to compress is known, one can apply a FOR to the extracted integer part of the number; further reducing its size (step 2 in Figure 2.27).
- Both parts (orange and blue) are stored byte-aligned to improve [de]compression speed.
- The doubles precision and the value used as a Frame Of Reference are stored as metadata for decompression.

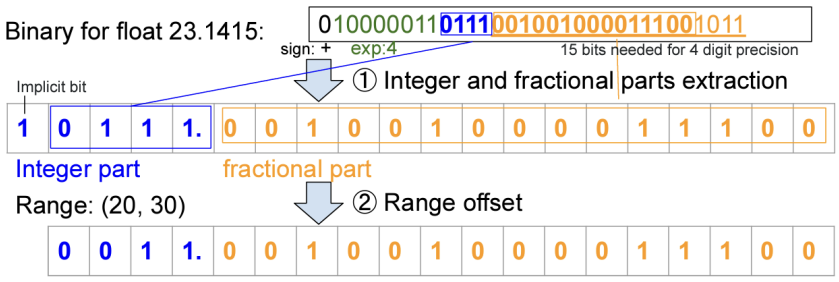

Figure 2.27: BUFF key idea overview (Figure borrowed from Liu et al. work (13))

It is important to highlight that BUFF needs to know a-priori the precision of the floating points in order to be used. Otherwise, BUFF is not able to perform compression.

BUFF was tested in 8 datasets; all within ranges of bounded floating-point data. Including stocks, temperature measurements, CPU usage and GPS coordinates. BUFF was tested against some LWC and GPC methods such as Gorilla, Sprintz, GZIP and Snappy. Furthermore, end-to-end query benchmarks were presented. BUFF outperforms Gorilla, Sprintz, Snappy and GZip by a considerable margin. Achieving the best compression ratios for all datasets. In terms of speed, BUFF outperforms almost every algorithm in every dataset, only falling behind Snappy in one dataset. On end-to-end query performance, BUFF shines and outperforms every other method by orders of magnitude in queries with highly selective equal filters and MAX aggregation. The latter is due to being able to decompress on a per-value basis and the ability to keep statistics of compressed data thanks to the splitting of the integer part of the numbers. Furthermore, a faster version of BUFF

called Fast-Buff leverages the availability of the ranges of values in a block as metadata. This eliminates the search for the FOR value in the third step of the process.

2.5.3.2 PseudoDecimals (PDE)

One way to achieve compressibility is to consider floating-point numbers that appear to be decimals and convert those to integers in order to apply integer encodings. For instance, one can take advantage of their visible decimal representation based on the fact that some doubles were generated as decimals (54). For example, if one could encode the number 8.0605 as the number 80605; integer encoding schemes could be applied and therefore take advantage of the commonalities found within the data. This can be effectively achieved by multiplying the double with a power of 10. Afterwards, one could recover the double by doing the inverse operation (i.e. multiplying the integer with an inverse power of 10). However, this process is lossy due to the error introduced by the inverse power of 10 in the multiplication at decoding (9). However, based on this same idea, Kuschewski et al. (1) recently introduced PseudoDecimals (PDE) – a lossless approach to encode floating-point values as integers based on their visible decimal representation.

PDE tries to encode doubles as a division between an integer and an inverse power of 10 under the assumption that the doubles were generated from a `DECIMAL`. Thus, we could also refer to this type of encoding as Decimal-based encoding. As we mentioned, this operation may result in errors if the chosen inverse power of 10 lacks the necessary precision. Hence, PDE performs a brute-force search among the entire solution space to find the inverse power of 10 which leads to the original double. In the PDE implementation, this solution space is set to the first 22 inverse powers of 10. Furthermore, PDE allows the encoding of exceptions in the case that a double is not able to be recovered from an integer representation with any of the possible powers of 10.

In detail, PDE works as follows for every double that is encoded:

- For every `exponent` e between 0 and 22:
  - Encode a candidate integer by **dividing** the double by the current inverse power of 10 (10^{-e}) and rounding the result.
  - Try to recover the exact original double from the candidate integer by **multiplying** it with the current inverse power of 10 (10^{-e}).
  - If the original double was recovered, store the candidate integer and the exponent e and stop the search.

  - Otherwise, continue to the next exponent.

- If no candidate integer to encode the double losslessly was found, store the double uncompressed as an *exception*

There are a couple of remarks regarding the PDE algorithm that are important to note: 1) Candidate integers are bounded to a maximum size of 32 bits. Thus, floating-points of a significant precision of more than 11 digits will always be encoded as exceptions. 2) As one may have noticed, by itself PDE does not compress data. PDE stored a 32-bit integer and a 32-bit exponent. However, PDE has the advantage that its output is further compressible using integer lightweight encoding schemes such as `FOR`, `BP` or `DELTA` (1, 3, 36). This is called **cascading compression**; which we will dive in-depth in section 2.6. To benchmark its compression ratios, PDE used a SIMD bitpacking implementation (SIMD-BP128) to compress the encoded exponent and integers (34).

PDE was tested against FPC, Gorilla, Chimp and Chimp128 using 12 non-time series datasets from the Public BI Benchmark (55)–a collection of the biggest Tableau Public workbooks (56). PDE achieved the best compression ratios in 7 out of the 12 datasets. In two datasets it achieved no compression since it encoded all values as exceptions. In some datasets, it achieved significant compression ratios of x75 and x54. On further inspection of these datasets, we discovered that they are comprised mostly of 0's. Despite not reporting speed benchmarks for PDE; we believe that it is slower than any other previous approach at compression due to the brute-force search and the use of division (an expensive operation in most ISAs (50)). However, we think decompression is the fastest of all since it is basically comprised of only one multiplication plus the overhead to handle exceptions.

Since decompression is made up of only one multiplication, PDE was easily SIMDized by using the `_mm256_cvtepi32_pd`[1] intrinsic (in case of the AVX2 ISA) to convert 32-bit integers to doubles and the `_mm256_mul_pd`[2] intrinsic to multiply doubles. However, if there are any exceptions in the block, the scalar implementation is used. PDE compression is hard to SIMDize since it is comprised of many control dependencies and a `FOR` loop with data dependencies (i.e. the continuity of the loop depends on a previous result).

[1] `https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/mm256-cvtepi32-pd.html`

[2] `https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/mm256-mul-pd.html`

2.5.4 Other Schemes

In this section, we discuss a scheme which greatly varies from the three main floating-point LWC research branches we have previously discussed.

2.5.4.1 SPDP

Claggett et al. (57) experimented with algorithms synthesizing to generate a new floating-point compression scheme called SPDP. The idea behind such synthetization is to automatically build all the possible permutations from a set of algorithms (referred to as *components*). Each permutation is then used as a pipeline to compress data. A permutation performance is ranked based on the compression ratio achieved on the data used for testing. Using this method, Claggett et al. generated 9,400,320 permutations of at most 4 stages from 48 components. These components were composed of LWC methods such as bit-packing and RLE, GPC methods (LZ77) and transformations such as bitwise negation, bitwise equidistant cuts, previous n-value XOR and previous n-value arithmetic subtraction. The best performant combination was: 1) Arithmetic subtraction with the 2nd previous value, 2) bytes cutting and reordering, 3) Previous value arithmetic subtraction and 4) LZ77. We would like to clarify step 2. This step cuts the result from step 1 into bytes (i.e. every double is cut into 8 segments). Then, the bytes of all the numbers in a block are reordered in such a way that the bytes of the same position of each tuple are next to each other. For instance, if result from step 1 is x, y, z values, the cutting will result in $x_1, x_2, ..., x_8, y_1, y_2, ..., y_8, z_1, z_2, ..., z_8$, and the tuples reordering will result in $x_1, y_1, z_1, x_2, y_2, z_2, ..., x_8, y_8, z_8$.

SPDP was evaluated on the same datasets that were used to evaluate FPC, and the benchmarks were performed against standalone GPC schemes. It demonstrated to be on average, 30% better in terms of compression ratio than the next most performant (bzip2); leaving behind other GPC schemes such as LZ4, LZO and Snappy. However, SPDP still fell behind Zstd in compression ratios. In terms of speed, SPDP while not being the best (beaten by Snappy), was still on par with methods such as Zstd while being faster than LZO and LZ4 modes optimized for compression ratios. Although not referred to as such, SPDP was a hint that cascading compression schemes worked also for floating-point data. This idea of synthezising algorithms to achieve better compression ratios has been further used to achieve better methods to compress 3D and 2D geometry grids (58).

2.6 Cascading Lightweight Compression

Some LWC compression methods output is just a smaller representation of a certain data type. For instance, if we apply DELTA compression to integers, the encoded values are just smaller integers that are bit-packed into just the necessary bits. This property of some LWC methods can be advantageous since it enables the possibility of applying them in a cascading fashion. In other words, we are able to apply different encodings one after another in order to achieve even higher compression ratios by trading off the speed of both [de]compression. For instance, if data exhibits a high percentage of consecutively repeated string values one can apply a RLE encoding, and then encode the runs lengths (integers) with a FOR encoding and the values with FSST encoding. It is important to note that such cascading cannot be used with GPC methods, since neither the input nor output are considered a sequence of values but rather continuous streams of bits.

This idea of cascading compression has been only recently explored by a few studies (1, 36) and proposed as future work on others (3). For instance, Damme et al. (36) were the first to introduce this idea with such a name (cascade). They tested the compression ratios that could be achieved by cascading schemas in different combinations such as DELTA + RLE and FOR + DICT. From these experiments, they observed that not every cascade works to improve compression ratios. For instance, DELTA + RLE usually negatively affects compression ratios. On the other hand, FOR + DICT proved to be a combination that always yields positive results. However, they only explored cascades of one level.

Kuschewski et al. (1) natively implemented multi-level cascading compression of LWC methods in their proposed big-data format based on Parquet (i.e. BtrBlocks) to decrease data size. In fact, the entire format is based upon cascading compression applied recursively to attributes inside datasets. In order to do so, BtrBlocks scans a sample of the data and calculates certain statistics such as the maximum, minimum and ratio of repeated values. Based on these statistics and the data type, BtrBlocks chooses an encoding from a pool of available encodings and applies it to the data. Afterwards, if possible, the output of the encoding is subject to the same process to apply a scheme on cascade. This is done recursively until the output is deemed not further compressible or a configurable maximum level of depth is reached.

2.6.1 Kernel Fusing

There are some encodings which by design must be used in combination with another encoding for it to compress data. For instance, a FOR encoding cannot achieve any com-

pression at all if it is not used with BP to store only the necessary bits. In such scenarios, one needs the implementation of two different kernels (i.e. functions)–one for FOR and one for BP. In such a setting, a "data transmission" must happen between the FOR kernel and the BP kernel once the former finishes its encoding. This translates into one STORE and one LOAD instruction. However, Afroozeh & Boncz (3) propose the idea of kernel fusing for these cases of encodings that only work when used together. The key idea is to implement more than one encoding into only one kernel (i.e. one function), which effectively saves both STORE and LOAD instructions between every encoding that is fused. Testing the fusing idea in a FOR + BP kernel resulted in an x2 performance increase at decoding speed. Furthermore, the big-data format BtrBlocks (1) implements fusing on DICTIONARY + RLE encodings since they found it is a recurring cascade that works well to achieve increased compression ratios.

Kernel fusing opens the opportunity to *freely* speedup virtually any combination of LWC schemas that can be plugged in cascade. The tradeoff mainly falls on implementation work and an increase in code size.

2.7 Storage Layouts: NSM, DSM & PAX

Data schemas are usually comprised of records and their properties (attributes). In relational schemas, these are namely rows and columns depicted as tabular data. However, the underlying layout of how the data is stored on disk can greatly vary. Such a design decision has an effect on the performance in terms of the speed of certain types of operations within the data. Furthermore, it has an effect on how the data can be compressed.

For instance, in the **N-ary Storage Model (NSM)** data is stored row-by-row. In other words, all attributes of a record are stored contiguously. A depiction of NSM can be seen in Figure 2.28. In this kind of storage random access of an entire record is fast since there is a locality by records. However, analytical queries are usually not interested in accessing all attributes of a record, but few. This causes an I/O bottleneck when querying since we would be forcefully reading unnecessary attributes due to how the data is stored. Similarly, the CPU cache is rapidly filled with attributes of records that probably are never need to be read. This translates into an increased number of cache misses. Moreover, this storage layout challenges LWC methods capabilities since the contiguous data in the disk is of variable size, nature and type. Thus, GPC is the preferred way to save storage in such a setting.

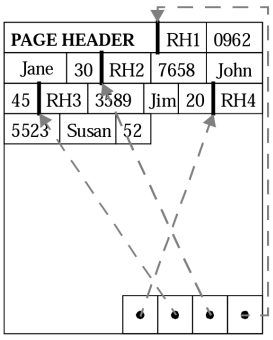

Figure 2.28: Overview of the N-ary Storage Model (Figure borrowed from Ailamaki et al. work (14))

The **Decomposition Storage Model (DSM)** proposed by Copeland et al. (59) tries to overcome these performance issues of NSM by storing together attributes (columns) instead of rows. Although more performant for OLAP workloads, if the entire record attributes are needed to be retrieved, DSM falls short with performance overhead, since different attributes of the same record are not stored close-by. To improve this Ailamaki et al. (14) proposed **Partition Attributes Across (PAX)**. In this storage layout, both attributes and records are stored close-by in *pages*. A depiction of PAX is presented in Figure 2.29. Each *page* is comprised of a header with information regarding the number of attributes, number of records, size of each attribute and free space of the page. Each page is then divided into *mini-pages*. Each mini-page is a partition that stores each attribute of all the records stored on the page. PAX defines two types of mini-pages: fixed-length (F-minipages) and variable-length (V-minipages). On F-minipages the attribute size is fixed, hence running through the minipage is done seamlessly. This kind of pages implement a footer composed of presence flags (0 or 1 bit) to detect null values in the minipage. On the other hand, each attribute value on V-minipages is of variable size. Hence, these pages implement a footer composed of the offset of bits each value in the minipage occupies. PAX and its many variations in modern data formats are known as columnar-storage.

Both DSM and PAX opened opportunities for more efficient compression since data stored close-by share the same type, nature and size (excluding V-minipages). This enables the use of LWC algorithms and motivated research on more data-type-specific encodings.

2.8 Compression and Data Formats

Compression is able to substantially reduce the size of data. This translates into more optimized storage usage and cost savings. Furthermore, it reduces the data transmission

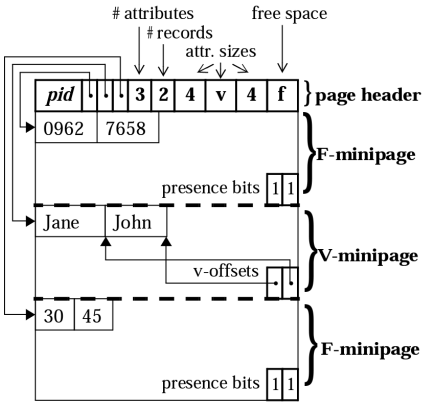

Figure 2.29: Overview of the Partition Attributes Across storage layout (Figure borrowed from Ailamaki et al. work (14))

overhead over a network since less data has to be moved. Thus, it is no surprise that [big]-data formats have compression mechanisms intrinsically implemented into their core. In addition to optimizing storage size, data formats usually provide efficient ways of traversing through large chunks of data and optimizing analytical SQL-like queries to query, filter or aggregate data. Sometimes these formats are a black box to end-users. However, most of these are optimized versions of *open* formats such as Parquet or ORC. In this section we analyze 3 different open-data formats for big data which leverage compression into their implementation: (i) Parquet, (ii) ORC and (iii) BtrBlocks.

2.8.1 Parquet

Parquet[1] is an open data format for [big]-data developed by Apache which was built to efficiently support data encoding, efficient data querying and *nested* data structures in the values of their columns. An example of nested data is a multi-level object. Parquet stores data using a PAX-based storage. In other words, it tries to store data in a columnar format while maintaining attributes from the same records close-by (though close-by in this context can be interpreted as "within a few megabytes" rather than within a traditional 4KB disk block). A high-level abstraction of a parquet file is depicted in Figure 2.30.

Each Parquet file is comprised of a 4 bytes *magic number* (which contains the file format), a series of *row groups* and a *footer* (which contains metadata of the file and each row group). **Rowgroups** are a logical horizontal partitioning of data into rows. This ensures that all the attributes (columns) of the same records are stored close-by. Parquet allows to configure a variable row group size. Each rowgroup is comprised of one *column chunk* per

[1] `https://github.com/apache/parquet-format`

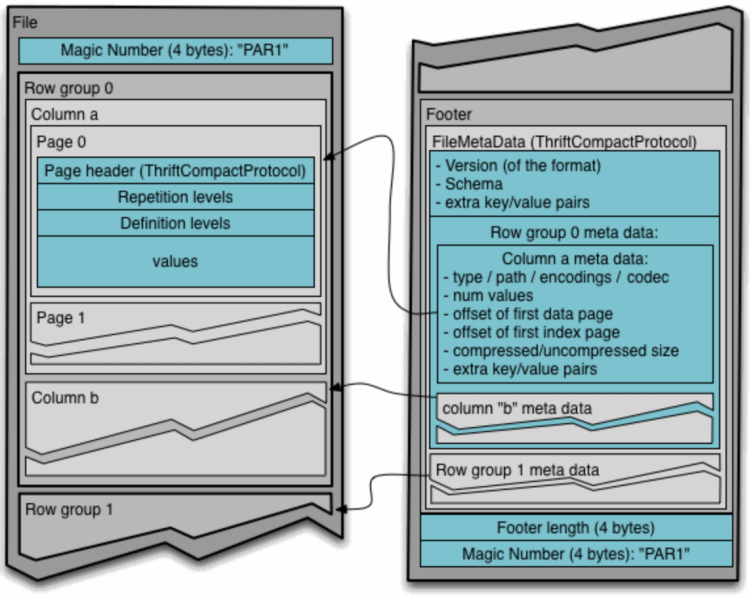

Figure 2.30: Overview of the Apache Parquet file format (Figure borrowed from Apache Parquet documentation [1])

each attribute of the rowgroup. Each column chunk is then divided into sequential *pages*. Aside from containing the actual column data, the pages also contain a *page header* and two values which are used when the values are part of a nested column: repetition levels and definition levels. We will dive further into these parameters further on this section.

The page header contains metadata regarding the page, such as it's [un]compressed size, the number of values, and the *encoding* used for the values. **The footer** of a Parquet file (right part of Figure 2.30) contains metadata of the file and of each column inside a rowgroup. Having this information inside a footer effectively separates the metadata from the data. The metadata of each rowgroup contains the column type (i.e. boolean, int32, int64, int96, float, double, bytearray, fixedlengthbytearray), the number of values, an offset to the first page, its [un]compressed size, and the encodings or compression used on the data. At a column level, GPC methods can be used to further encode pages. GPC[1] and LWC[2] algorithms supported in Parquet are presented in Table 2.1. Parquet automatically analyzes the data to determine which encoding is appropriate for the values inside a page. As the table shows, Parquet implements a few of the LWC methods we discussed in previous sections. On the contrary, they implement a lot of modern GPC methods. As one may have noticed, Parquet does not really focus its compression on LWC methods.

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

As we previously mentioned, Parquet supports the storage of **nested data structures**. Parquet implements a novel approach introduced by Melnik et al. (60) by storing two additional values per page which capture the nested structure of a record: definition and repetition levels. Definition levels specify at which level of the nested structure that value falls within. For instance, if the maximum nested level of a column is 3, then every value in the definition level will range between 0 and 3 depending up to which level the nesting is defined or not. On the other hand, repetition levels serve as an indicator of at which levels repeated structures (i.e. arrays) can happen. In other words, it determines the structure of the nested type. Needless to say, both repetition and definition values are only present if the column is nested.

Parquet is currently the de-facto standard when building data lakes or storing data to be used by analytical workflows (e.g. Apache Spark in Databricks). Being an open-source format with an accessible API, any individual can use Parquet as their file format to store and query their data. For instance, Pandas, the popular data analysis Python library, allows reading and saving DataFrames from and to Parquet files. Another example is the analytical database DuckDB (20), which allows the importing of tables directly from the Parquet format.

2.8.2 ORC

Even though Parquet is currently the de-facto standard for [big]-data formats, there exist other open data formats that are worth studying. For instance, Optimized Row Columnar (ORC) (15) is another open [big]-data format created as an optimized version of the RC file format (61) to be data-type aware in order to use encodings to compress data more efficiently. Contrary to Parquet, ORC implements more LWC encodings into their format. A high-level abstraction of an ORC file is depicted in Figure 2.31.

Similarly to Parquet, ORC is also a PAX-based storage that stores data in a columnar format while maintaining attributes from the same records close-by to easily traverse through records. ORC files start with a sequence of *stripes* and end with a *footer* and a *postscript*. The *footer* contains a sequence of pointers to each stripe, the number of rows per stripe, each column datatype and each column aggregated statistics (min, max, count and sum). The postscript contains information about the used compression parameters and the size of the compressed footer.

Stripes are a logical partition of data (tables) horizontally. They are similar to row-groups in Parquet. Stripes have a fixed configurable size. Each stripe is comprised of three segments: (i) row index data, (ii) row data and (iii) stripe footer. The row index data

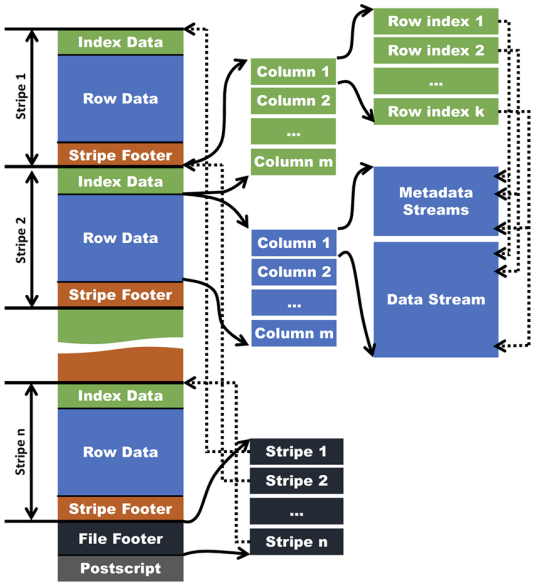

Figure 2.31: Overview of the ORC file format (Figure borrowed from Huau et al. work (15))

contains, for each column, statistics (min/max) and pointers to where the data streams of that column are located inside the row data. Such pointers, alongside the file footer stripe pointers and statistics, enable fast seeking to the right data stream. As Figure 2.31 shows, these row indexes are stored for each attribute inside the data. This allows for fast reads despite a high stripe size since data can be easily skipped.

One of the main reasons to develop ORC was the implementation of encodings before plugging in any GPC. ORC leverages data statistics, data type and ratio of distinct values to decide which encoding to use to compress the data streams. Encodings supported by ORC are presented in Table 2.1. Afterwards, the result of the encoded values can be further optionally compressed with any of the GPC methods listed in Table 2.1.

2.8.3 BtrBlocks

Parquet can be deemed costly in terms of scan performance since they do not leverage most of the current LWC encodings to further optimize the file sizes; rather Parquet prefers GPC schemas. This results in elevated costs for scanning data that in the end is going to be discarded by a query filter. BtrBlocks (1) is a recently proposed open-data format for Data Lakes that tries to overcome this problem with a new file format that implements fast [de]compression through a great variety of LWC methods that can be applied recursively in cascade to achieve even higher compression ratios (e.g. A column initially compressed

with PDE can be further compressed with DELTA, and then with BP). By optimizing storage, cloud provider costs of data transmission and processing can be reduced. LWC encodings implemented in BtrBlocks are presented in Table 2.1. BtrBlocks also introduces a new LWC encoding called PseudoDecimals for floating-point data previously reviewed in subsection 2.5.3.2. Furthermore, they support recursive cascade encoding of LWC methods.

BtrBlocks also introduces a simple sampling mechanism to further optimize the selection of the most suitable encoding algorithm for the data. The sampling scans through 1% of the data in equidistant runs of values. From this sampling, BtrBlocks leverages data types, data repetition, and data statistics (e.g. min, max) to choose from the pool of encodings.

2.9 Compression and Database Engines

In the context of OLAP and OLTP databases the *speed* in which information is stored and queried is of critical importance. Implementing any compression schema to reduce storage size usually comes with a speed overhead to execute the compression and decompression procedures. However, modern databases value size reductions at the expense of this speed overhead. General purpose compression methods are not efficient in such setups since random access is required to be able to efficiently query segments of the data. Thus, in the context of databases, lightweight compression is preferred in most cases –if possible.

In this section, we analyze use cases of how some modern database engines implement compression.

2.9.1 Compression in DuckDB

DuckDB(20) is an in-process SQL OLAP database management system that uses a columnar storage layout. DuckDB specializes in the performance of complex analytical queries on large tables. Its speed stems from its vectorized engine and parallel query processing. Tables are split into row-groups of 120K rows, which store each attribute in columnar chunks called Segments (analogous to Parquet rowgroups and columns). However, in DuckDB, these segments are of fixed size. At query time, these segments are loaded and processed as vectors, also of fixed size (2048 values); so that they can fit in CPU cache and achieve vectorized execution.

DuckDB implements LWC into its tables' schemas (62) by first scanning the columnar segment and deciding which encoding will achieve the best gains in terms of compression ratios. After a decision has been made, compression is executed and compressed data is written on disk. The engine also allows users to manually set different encodings into

their columns instead of automatically choosing an encoding. The available encodings in DuckDB are presented in Table 2.1, from which we can highlight the implementation of Chimp[128] and Patas and the lack of GPC methods. Thanks to its storage layout comprised of many columnar chunks, DuckDB allows the use of different encodings for different chunks of the same column.

2.9.2 Compression in Amazon Redshift

Redshift[1] is a columnar storage relational database created by Amazon Web Services whose engine is built upon PostgreSQL. Redshift is specialized in analytical queries of big data while also efficiently supporting transactional queries (e.g. INSERT, UPDATE).

By default, Redshift applies different compression methods depending on the column type[2]. Furthermore, they allow users to manually set different encodings into their columns. The available encodings are presented in Table 2.1, from which we can see a few of the LWC methods we have already discussed: RLE, Delta, Dictionary, and some GPC methods such as Zstd and LZO. However, as seen in Table 2.1, there is one encoding which we have not discussed: **AZ64**. This encoding is an encoding designed by Amazon from which there is little to no information on how it works[3][4]. AZ64 is used by default on columns of any of these types: INTEGER, DECIMAL, DATE, TIME and TIMESTAMP. Amazon reports a 5-10% improvement in compression ratio and a 70% improvement in speed against Zstd. However, recent empirical testing has contradicted these benchmarks; reporting a similar performance between both algorithms[5]. In addition to this, recent reverse engineering efforts[6] have concluded that it is an adaptive scheme with four different modes depending on properties of the data. All of these modes are based on enhanced versions of the Facebook's Gorilla algorithm for timestamps compression in combination with RLE.

However, these efforts have found that there are four cases in which the algorithm does not perform well: (i) Tables sorted with interleaved keys[7], (ii) Unsorted tables, (iii) Unsorted columns, (iv) Data which have a high number of repeated values (on this cases RLE is much better).

[1] https://aws.amazon.com/redshift/
[2] https://docs.aws.amazon.com/redshift/latest/dg/r_CREATE_TABLE_NEW.html
[3] https://aws.amazon.com/about-aws/whats-new/2019/10/amazon-redshift-introduces-az64-a-new-compression-
[4] https://docs.aws.amazon.com/redshift/latest/dg/az64-encoding.html
[5] http://www.hydrogen18.com/blog/redshift-az64-performance-vs-zstd.html
[6] https://www.amazonredshiftresearchproject.org/white_papers/downloads/az64_encoding.pdf
[7] https://docs.aws.amazon.com/redshift/latest/dg/t_Sorting_data.html

2.9.3 Compression in MySQL: InnoDB

In MySQL (InnoDB) Data is stored on a row-by-row basis. They use an optimized version of the NSM format which we will not explain in detail in this literature review. In MySQL (InnoDB), only general purpose compression is used. Specifically, they implement LZ77[1] – another algorithm of the Lempel-Ziv family. It is important to highlight how the way in which data is stored in MySQL (InnoDB) inhibits the use of LWC methods.

2.9.4 Compression in CodecDB

CodecDB (63) is a recently presented columnar database which is encoding-aware. In other words, the database design and the data encodings are tighly coupled. As seen in Table 2.1, CodecDB natively implements RLE, DICTIONARY, Delta, FOR, PFOR, Bitpacking and even encodings in cascade such as DICTIONARY + RLE. CodecDB implements an encoding selection scheme in which a series of features are extracted from the data in order to choose the encoding that best fits these features. Some of these features are the ratio of distinct values, the values statistics (i.e. mean, max, min), the sparsity ratio (non-empty records vs. total number of records) and the level of sortedness in the data (how "in order" the data is). Experiments carried out to test query speed differences between query engines that are oblivious to the encoding schemes and CodecDB were carried out, demonstrating that an encoding-aware query execution engine is faster in several queries (e.g. aggregation, joins, comparisons).

2.10 Whitebox Compression

While data is compressed, query engines are not able to fully exploit query optimization techniques such as predicate push-downs. Blocks of data must be first decompressed entirely for query operators to operate on data. Ghita et al. (16) refer to all conventional LWC and GPC methods as black-box compression since their [de]compression logic is unknown and inaccessible to the query execution engine. Thus, they propose a compression model called whitebox compression. Whitebox compression differentiates two types of columns: logical and physical columns. Logical columns are the columns with the "raw" data that the end-user can visibly understand (i.e. data in its tabular representation). However, logical columns are not stored on disk, rather they are generated at query execution time from the physical columns using a series of composite functions. Such functions consist

[1] `https://dev.mysql.com/doc/refman/8.0/en/innodb-compression-internals.html`

of simple scalar operators. Hence, the physical columns are the only ones that are stored on disk and these columns are (ideally) "compressed" representations of the data. In Figure 2.32, an example of Logical vs. Physical columns is presented, alongside the functions to recover the logical columns.

$$\begin{aligned} A &= concat(map(P, dict_{AP}), const(\texttt{"\_"}), format(Q, \texttt{"\%d"})) \\ B &= map(P, dict_{BP}) \end{aligned}$$

A	B
"GSA_8350"	"GENERAL SERVICES ADMINISTRATION"
"GSA_8351"	"GENERAL SERVICES ADMINISTRATION"
"HHS_2072"	"HEALTH AND HUMAN SERVICES"
"TREAS_4791"	"TREASURY"
"TREAS_4792"	"TREASURY"
"HHS_2073"	"HEALTH AND HUMAN SERVICES"
"GSA_8352"	"GENERAL SERVICES ADMINISTRATION"

Logical

P	Q
0	8350
0	8351
1	2072
2	4791
2	4792
1	2073
0	8352

Physical

Figure 2.32: An example of Logical vs Physical data in the whitebox compression model (Figure borrowed from Ghita et al. work (16))

This representation allows the fusing of decompression and query execution allowing query optimizations such as predicate push-down and physical column pruning. Furthermore, it demonstrated to be able to achieve on average two times more compression ratio than black-box methods (i.e. LWC and GPC) since the functions are defined on a per-column basis instead of having a fixed algorithm for all cases.

Although promising, whitebox compression is still in an early stage of research. Some questions arise regarding how to find the best functions to represent the logical columns, and how to address dirty data (e.g. exceptions/outliers in columns which make a simple function non-viable for an optimal physical representation).

2.11 Pattern Inference Decomposed Storage (PIDS)

In 2020, Jiang et al. presented PIDS (64). PIDS main idea is to use unsupervised learning on string columns to detect recurring patterns in the strings and outliers to these patterns. Here, a pattern is not only limited to recurring substrings but also patterns such as strings of equal-length or equal-structure which can be exploited by splitting the string and storing each segment physically separately and applying further encoding schemes to each segment. PIDS allows for predicate pushdowns at query execution time, hence resulting in faster queries to obtain data. In some way, PIDS could be considered a white-box compression mechanism.

Type	Name	GPC	LWC	Cascade Support
File Format	**Parquet**	Snappy, Brotli, Gzip, Zstd, LZ4, LZO	Dictionary, RLE, BP, Delta	Dictionary + RLE / BP
	ORC	Zlib, Snappy, LZO	Dictionary, RLE, Delta	No
	BtrBlocks	*None*	Dictionary, RLE, Constant, BP (SIMD-FastBP128), FOR (SIMDFastPFOR), FSST, PDE	[LWC] + [LWC] + ... + [LWC]
Database Engine	**DuckDB**	*None*	Dictionary, RLE, Constant, BP, FOR, FSST, Chimp, Chimp128, Patas	No
	Amazon Redshift	Zstd, LZO	Dictionary, RLE, BP (by the name of Mostlyn), Delta, AZ64	No
	MySQL (InnoDB)	LZ77	*None*	No
	CodecDB	*None*	Dictionary, RLE, Delta, FOR, BP	Dictionary + RLE / BP

Table 2.1: Lightweight and General Purpose Compression schemes supported by a variety of file formats and database engines.

The capabilities of PIDS for string compression were compared against GPC schemes such as Snappy and GZIP. PIDS achieved the best compression ratio amongst all of the methods it was compared against by a comfortable margin. For instance, achieving 20% more compression than Gzip. In terms of compression throughput, it stands faster than both GZIP and Snappy (around 2x faster) at almost 50MB/s. As expected, on end-to-end query performance on compressed data PIDS is superior by orders of magnitude (2x - 30x) to GPC methods.

3

Datasets Analysis

The main objective of LWC algorithms is to find exploitable properties in values within a block of data. The more of these properties that they may find, the more compression they can potentially achieve. On the other hand, if such properties are not present in data, a compression method can fail to achieve compression and even result in negative compression (i.e. compressed data is bigger than uncompressed). Hence, data properties are a conditioning factor for performance in terms of compression ratio and [de]compression speed. For instance, XOR-based approaches perform the best when the data entropy is low and there are data similarities within their bit-wise representation. On the other hand, Decimal-based approaches perform the best when the data consists of floating-point numbers that stem from a decimal representation.

In this section, we meticulously analyze the properties of a wide variety of real floating-point datasets. By doing so, we aim to discover properties that have not been previously exploited by other algorithms. Furthermore, we are interested in analyzing these datasets from the point of view of *vectorized* query processing, since big data format readers and scan subsystems of database systems by now standardize on this methodology (20, 65): they deliver vector-sized chunks of data, and use decompression kernels that decompress one vector (e.g., 1024 values) at-a-time.

In subsection 3.1 we present the chosen datasets and their semantics. Next, in subsections 3.2 and 3.3 we analyze properties from the datasets from two different points of view: (i) their bitwise properties (IEEE 754) and (ii) their human-readable properties. This segmentation aligns with the two active branches of research in floating-point compression (i.e. XOR-based and Decimal-based). Finally, in subsection 3.4 we summarize and discuss the unexploited opportunities that cemented the road for the development of ALP.

Table 3.1: Floating-Point Datasets

	Name ↓	Semantics	Source	N° of Values
Time series	Air-Pressure(66)	Barometric Pressure (kPa)	NEON	137,721,453
	Basel-temp[1]	Temperature (C°)	meteoblue	123,480
	Basel-wind[1]	Wind Speed (Km/h)	meteoblue	123,480
	Bird-migration[2]	Coordinates (lat, lon)	InfluxDB	17,964
	Bitcoin-price[2]	Exchange Rate (BTC-USD)	InfluxDB	2,686
	City-Temp[3]	Temperature (F°)	Udayton	2,905,887
	Dew-Point-Temp(67)	Temperature (C°)	NEON	5,413,914
	IR-bio-temp(68)	Temperature (C°)	NEON	380,817,839
	PM10-dust(69)	Dust content in air (mg/m3)	NEON	221,568
	Stocks-DE[4]	Monetary (Stocks)	INFORE	43,565,658
	Stocks-UK[4]	Monetary (Stocks)	INFORE	59,305,326
	Stocks-USA[4]	Monetary (Stocks)	INFORE	282,076,179
	Wind-dir(70)	Angle Degree (0°-360°)	NEON	198,898,762
Non Time series	Arade/4[5]	Energy	PBI Bench.	9,888,775
	Blockchain-tr[6]	Monetary (BTC)	Blockchain	231,031
	CMS/1[5]	Monetary Avg. (USD)	PBI Bench.	18,575,752
	CMS/25[5]	Monetary Std. Dev. (USD)	PBI Bench.	18,575,752
	CMS/9[5]	Discrete Count	PBI Bench.	18,575,752
	Food-prices[7]	Monetary (USD)	WFP	2,050,638
	Gov/10[5]	Monetary (USD)	PBI Bench.	141,123,827
	Gov/26[5]	Monetary (USD)	PBI Bench.	141,123,827
	Gov/30[5]	Monetary (USD)	PBI Bench.	141,123,827
	Gov/31[5]	Monetary (USD)	PBI Bench.	141,123,827
	Gov/40[5]	Monetary (USD)	PBI Bench.	141,123,827
	Medicare/1[5]	Monetary Avg. (USD)	PBI Bench.	9,287,876
	Medicare/9[5]	Discrete Count	PBI Bench.	9,287,876
	NYC/29[5]	Coordinates (lon)	PBI Bench.	17,446,346
	POI-lat[8]	Coordinates (lat, in radians)	Kaggle	424,205
	POI-lon[8]	Coordinates (lon, in radians)	Kaggle	424,205
	SD-bench[9]	Storage Capacity (GB)	Kaggle	8,927

3.1 The Datasets

Table 3.1 presents an overview of the 30 datasets that we choose to analyze in detail in order to design ALP: 18 of these datasets were previously analyzed and evaluated to develop Elf (12) and Chimp[128] (9). Similarly, the other 12 were used to evaluate PDE (1) and are part of the Public BI Benchmark (55). The Public BI Benchmark is a collection of the biggest Tableau Public workbooks (56). We consider these 30 datasets to be relevant since they consist of a great variety of double precision floating point data from different nature and distribution. Furthermore, they played a key role in the design and evaluation of the other state-of-the-art floating-point encodings; which at the same time enables us to focus on properties that have not been analyzed yet. Finally, by using these datasets we are able to perform a fair comparison between these methods and our new ALP compression.

The first 13 datasets presented in Table 3.1 are comprised of sequences of doubles ordered increasingly by the time in which they were recorded. The latter is known as **time series** data. On these datasets, each double value v_{i+1} is recorded further in time than value v_i. The next 17 datasets contain doubles which are not stored in a time series fashion. These are more representative of doubles stored in classical database workloads. We segregate the analysis of time series and non-time series data since the underlying temporal property of such datasets can create exploitable advantages for compression methods. It is important to highlight that all these datasets are non-synthetic, since generating datasets synthetically could produce predictable results for a compression scheme given the synthetic generation method chosen.

Our datasets have further variety regarding the nature of the data itself. As presented in Table 3.1, 14 datasets contain doubles that represent monetary values (i.e. Exchange rates, public funds, product prices, stocks and crypto-currencies). 4 of our datasets represent coordinates (i.e. latitude and longitude), 2 contains discrete counts stored as doubles and 1 contains computer storage capacities. Finally, the other 9 datasets depict a variety of scientific measures (i.e. temperature, pressure, concentration, speed, degrees and energy) with different decimal precision. Some datasets share a common prefix in their name

[1] https://www.meteoblue.com/en/weather/archive/export/basel_switzerland
[2] https://github.com/influxdata/influxdb2-sample-data
[3] https://www.kaggle.com/sudalairajkumar/daily-temperature-of-major-cities
[4] https://zenodo.org/record/3886895#.ZDBBKuxBz0r
[5] https://github.com/cwida/public_bi_benchmark
[6] https://gz.blockchair.com/bitcoin/transactions/
[7] https://data.humdata.org/dataset/wfp-food-prices
[8] https://www.kaggle.com/datasets/ehallmar/points-of-interest-poi-database
[9] https://www.kaggle.com/datasets/alanjo/ssd-and-hdd-benchmarks

followed by a number. This means that a dataset, from the point of view of a database table, had more than one `double` column. Thus, this number represents the *index* of the analyzed *column*.

The format in which these datasets are found online greatly varies. For instance, the data of the stock datasets are scattered amongst thousands of files which contain metadata for each registered double. Thus, a pre-processing was made to strip from these files only the doubles themselves. Furthermore, it is important to mention that such doubles are found as `string` literals that are then read and cast into a 64-bit double. Hence, each double in its raw format contains a fixed decimal point that determines what we will refer to as a double *visible precision*. Note that this is usually how user-contributed data is generated (e.g. a field in a web form to enter an individual height and is afterwards stored as a double).

Table 3.2: Detailed metrics computed on the Datasets

Name ↓	**Decimal Precision Max | Min | Avg. | Std. Dev.**				**Values per Vector Non-Unique % | Avg. | Std. Dev.**			**IEEE 754 Exponent per Vector Avg. | Std. Dev.**		**Success of P_{enc} and P_{enc} using one exponent e per: Value | Dataset | Vector**			**Previous Value XOR 0's Bits Front | Trail.**	
C1	*C2*	*C3*	*C4*	*C5*	*C6*	*C7*	*C8*	*C9*	*C10*	*C11*	*C12*	*C13*	*C14*	*C15*
Air-Pressure	5	0	4.9	0.3	74.7%	93.4	0.1	1021.5	0.0	63.2%	14 (99.4%)	99.4%	44.5	32.9
Basel-temp	11	5	6.3	0.4	26.2%	11.4	4.6	1025.5	1.0	64.3%	14 (99.7%)	99.7%	14.0	2.6
Basel-wind	8	0	6.1	1.2	61.8%	7.1	4.1	1024.7	12.8	65.8%	14 (98.6%)	98.6%	14.2	3.1
Bird-migration	5	1	4.5	0.8	55.9%	26.6	6.0	1026.4	0.6	61.7%	14 (93.8%)	96.4%	26.4	7.8
Bitcoin-price	4	1	3.9	0.4	0.0%	19187.5	790.6	1037.0	0.0	84.2%	14 (99.9%)	99.9%	20.6	1.0
City-Temp	1	0	0.9	0.3	60.3%	56.0	21.3	1028.3	1.6	67.3%	14 (97.4%)	97.4%	15.8	11.0
Dew-Point-Temp	3	0	2.8	0.3	19.3%	14.4	1.4	1026.0	1.1	80.2%	14 (99.3%)	99.3%	16.8	1.5
IR-bio-temp	2	0	1.9	0.3	49.1%	12.7	4.2	1025.6	4.8	83.5%	14 (99.3%)	99.3%	22.0	7.8
PM10-dust	3	0	2.8	0.2	93.7%	1.5	0.8	1016.1	1.2	88.8%	14 (99.9%)	99.9%	40.5	38.3
Stocks-DE	3	0	2.4	0.5	89.2%	63.8	9.1	1027.8	0.3	84.2%	14 (98.9%)	99.1%	24.9	5.8
Stocks-UK	2	0	1.2	0.6	88.1%	1593.7	317.1	1032.2	0.4	84.5%	14 (99.9%)	100.0%	23.7	19.4
Stocks-USA	2	0	1.9	0.4	91.5%	146.1	11.7	1029.1	0.1	87.5%	14 (98.6%)	99.2%	32.6	16.8
Wind-dir	2	0	1.9	0.3	3.9%	192.4	81.1	1029.8	1.2	90.0%	14 (99.5%)	99.5%	13.8	2.6
TS AVG.	**3.9**	**0.5**	**3.2**	**0.5**	**54.9%**	**1646.7**	**96.3**	**1026.9**	**1.9**	**77.3%**	**94,8%**	**99.0%**	**23.8**	**11.6**
Arade/4	4	0	3.5	0.6	0.2%	738.4	389.8	1031.6	0.9	80.1%	14 (99.5%)	99.5%	13.1	1.1
Blockchain-tr	4	0	3.8	0.6	0.6%	638646.4	$1.3E^7$	1031.8	12.5	76.3%	14 (92.1%)	92.3%	9.8	1.7
CMS/1	10	0	4.0	2.8	54.7%	97.0	110.0	1028.0	1.3	83.2%	14 (98.5%)	98.6%	32.9	24.8
CMS/25	10	0	9.1	1.9	5.7%	12.6	19.2	984.1	179.1	68.0%	14 (98.7%)	98.7%	9.5	1.5
CMS/9	1	0	0.0	0.0	71.5%	235.7	908.5	1028.3	1.6	100.0%	14 (99.9%)	100.0%	11.8	47.3
Food-prices	4	0	1.1	1.1	52.5%	6415.8	14656.8	1030.4	1.8	92.4%	14 (99.2%)	99.2%	27.1	33.5
Gov/10	2	0	1.0	0.8	26.1%	240153.6	$1.6E^7$	873.5	298.8	90.5%	14 (89.9%)	95.9%	13.8	18.8
Gov/26	2	0	0.0	0.0	99.5%	442.3	8036.8	4.6	11.9	100.0%	14 (99.9%)	100.0%	63.7	63.8
Gov/30	2	0	0.1	0.3	89.7%	10998.7	102748.6	115.6	170.6	98.6%	14 (98.5%)	99.4%	56.6	57.1
Gov/31	2	0	0.1	0.1	96.0%	893.2	6288.2	69.9	57.4	99.1%	14 (99.8%)	99.9%	60.6	60.9
Gov/40	2	0	0.0	0.0	99.1%	791.4	6650.9	12.1	18.7	99.9%	14 (99.8%)	99.9%	63.4	63.5
Medicare/1	10	0	4.0	2.9	41.3%	97.0	146.2	1028.0	1.6	83.2%	14 (98.5%)	98.6%	25.2	16.6
Medicare/9	1	0	0.0	0.0	70.6%	235.7	1006.2	1028.3	1.7	100.0%	14 (99.9%)	100.0%	11.3	47.1
NYC/29	13	0	12.9	0.3	51.0%	-73.9	0.0	1029.0	0.0	93.7%	14 (99.9%)	100.0%	38.9	23.2
POI-lat	20	0	15.9	0.4	1.4%	0.6	0.4	1021.7	1.4	73.4%	16 (74.1%)	76.4%	10.6	1.0
POI-lon	20	0	15.7	0.5	0.8%	-0.1	1.2	1022.0	4.0	64.6%	16 (61.5%)	70.5%	5.1	1.0
SD-bench	1	0	0.9	0.2	92.4%	446.0	521.5	1030.3	1.2	65.8%	14 (99.9%)	100.0%	17.4	15.8
NON-TS AVG.	**6.4**	**0.0**	**4.2**	**0.7**	**50.2%**	**52948.9**	**1745162.6**	**786.4**	**45.0**	**86.4%**	**95.1%**	**95.8%**	**27.7**	**28.2**
ALL AVG.	**5.3**	**0.2**	**3.8**	**0.6**	**52.2%**	**30717.9**	**988967.2**	**890.6**	**26.3**	**82.5%**	**95.0%**	**97.2%**	**26.0**	**21.0**

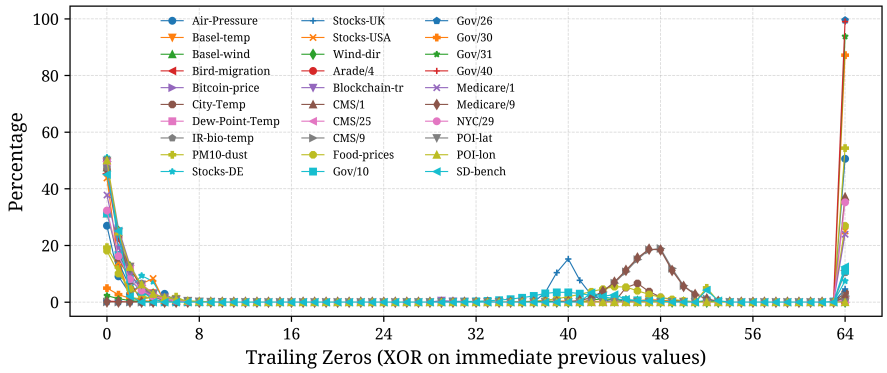

Figure 3.1: Distribution of Trailing Zeros resulting from XORing each value with its previous immediate value.

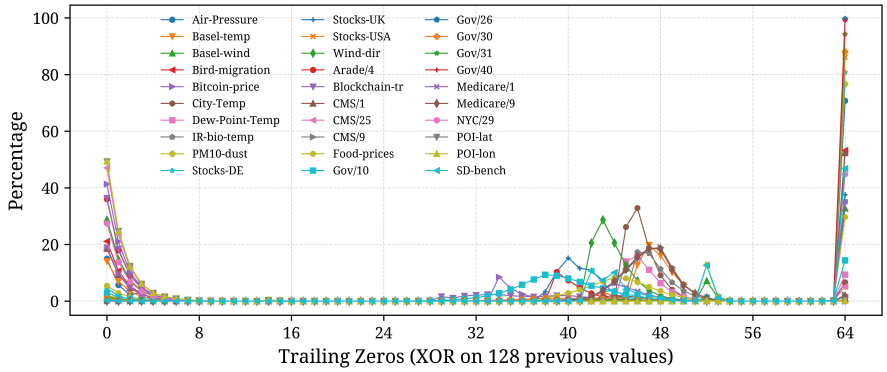

Figure 3.2: Distribution of Trailing Zeros resulting from XORing each value with one of its 128 previous values.

3.2 XOR-based Analysis

From a bitwise perspective, two double floating point values are considered *similar* if their sign, exponent and fraction parts are similar (IEEE 754). Table 3.2:C9 and C10 show the doubles exponent average and deviation per vector. We define a `vector` as 1024 consecutive values. Also, note that here the 11 bits of the exponent are used to represent an unsigned integer. In most of the datasets, this exponent deviation is very small. Especially in time-series data. These small deviations on the exponent, which comprises most of the front bits, are consequently reflected in the amount of leading 0's bits resulting when XORing the doubles with their previous values.

3.2.1 Leading and Trailing Zeros

When similar values are XORed, a compressible-friendly chain of bits is yielded. This XORed chain of bits usually contains a high amount of leading and trailing zero bits. As we can see in Table 3.2:C14 and C15, the average number of leading and trailing zero bits after XORing is similar between time series and non-time series data. Hence, this *similarity* of values stored close-by is also present on non-time series data. This is further shown by how well Chimp and Chimp128 perform on both of these types of datasets (9). Note that this has an explanation: Data usually follows normal or skewed distributions with few outliers. Furthermore, the nature of the data usually determines its decimal precision. For instance, data measured from one sensor will always have the same precision. Knowing that by the IEEE 754 definition, the front bits in a certain way determine the magnitude of numbers; and the trailing bits determine their precision; such finding that close-by values are similar in their bitwise representation is expected.

Regardless of the data nature, leading and trailing zero bits seem to be compromised by lower percentages of non-unique values (Table 3.2:C6) and higher decimal precisions (Table 3.2:C2). For instance, in both datasets in which the visible decimal precision reaches up to 20 (i.e. POI-lat and POI-lon), the leading and trailing zero bits average of XORed values is the lowest. It is important to note that these values of decimal precision stem from the doubles being originally found as `string` literals. Only then it is possible to have a number with a significant precision of 20 decimals; since the IEEE 754 definition of doubles would not be able to achieve such precision accurately (maximum of 17).

The distribution of the number of *trailing zeros* for each dataset can be seen in Figure 3.1. Here we can see that the distributions of zeros are mainly located in three ranges of bits (i.e. 0 - 8; 32 - 48; and 64). When plotting the same distribution but looking for the best XOR result amongst the previous 128 values (Figure 3.2) the differences are subtle. The distributions of zeros are still within the same 3 ranges of bits. However, when looking into the previous 128 values, the distribution is less cluttered in the first range (1-8 bits) and much more cluttered on a perfect XOR (64 zeros). At the same time, some datasets gain zeros in the middle range (from 32 to 48 bits). Even though the gains on this range are remarkable for a few datasets (e.g. Wind-dir, City-Temp, Dew-Point-Temp, Basel-Temp); we believe the biggest gains achieved by Chimp128 are in its ability to achieve a perfect XOR. The two datasets with the highest amount of zero trailing zeros are both datasets in which the decimal precision is truly high (i.e. POI-lat and POI-lon) and there are almost no repeated values.

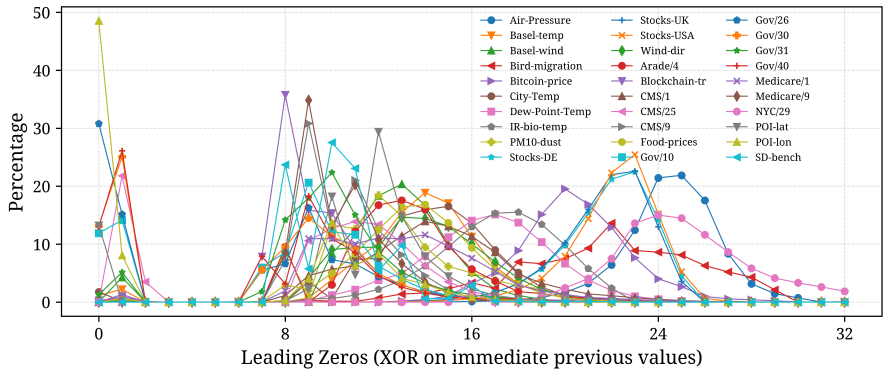

Figure 3.3: Distribution of Leading Zeros resulting from XORing each value with its previous immediate value.

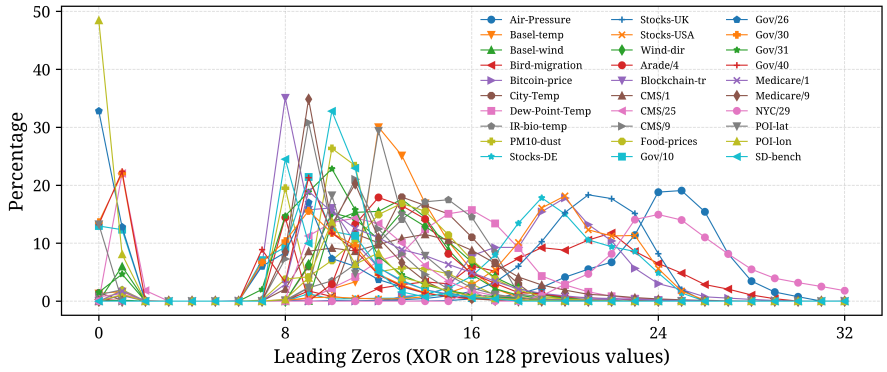

Figure 3.4: Distribution of Leading Zeros resulting from XORing each value with one of its 128 previous values.

When analyzing the distribution of *leading zeros* for each dataset we found that these are more consistently exploitable than trailing zeros (Figure 3.3). This is due to most of the doubles inside a vector having a low variance on their IEEE 754 exponent. In fact, only a few datasets have a relevant part of their leading zero distribution between 0 and 8 bits (e.g. POI-lon, Gov/26, Gov/40, Gov/10). Most of the datasets have their distribution between 8 and 24 bits in an unimodal fashion. When doing the same analysis but XORing with one of the previous 128 values (Figure 3.4) we found that the changes are very subtle. There are no significant gains in terms of leading zero bits. In fact, some distributions even become worse by being more skewed to the right (e.g. Air-pressure, Stocks-USA, Bitcoin-Price, Basel-Wind). This shows how looking into the previous 128 values is more beneficial in terms of trailing zeros than leading zeros and it even diminishes the performance on some datasets regarding leading zeros.

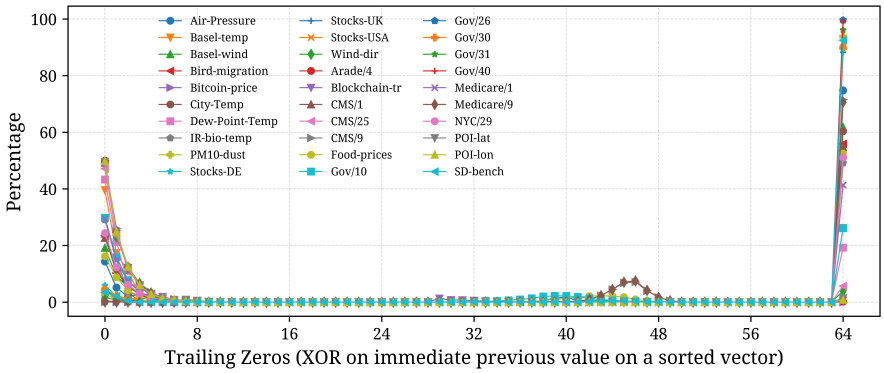

Figure 3.5: Distribution of Trailing Zeros resulting from XORing each value with its previous immediate value on a block of values sorted in ascending order.

3.2.2 Towards a SIMD XOR-based encoding

Looking into a ring buffer of N-previous values creates data dependencies since a value [de]compression always depends on a previous value that could be far behind the stream. Such a buffer would not be necessary to achieve higher compression ratios if an optimal value to XOR was found immediately prior to the current value. One could try to partially *sort* the block of doubles to achieve this (as similar doubles in magnitude are more likely to share their front bits). Figures 3.5 and 3.6 show the resulting distributions of trailing and leading zeros when XORing with the immediate previous value inside a vector of doubles sorted in ascending order. In contrast to 1 and 128-previous value XORing (Figures 3.3 and 3.4) the distributions of leading zeros are mostly skewed to the left, with a mean of around 16 and 32 and a smaller standard deviation. On the other hand, the trailing zero distributions depict a higher percentage of datasets values falling in 64 zeros and a much lower percentage of the distributions between 0 and 8 bits; while the middle range (32-48 bits) is now not present for most datasets. These distributions hint to us that having sorted tuples a-priori would improve compression ratios of XORing (we estimate improvements of ≈12% in Patas). In FastLanes (3); tuples in a SIMD lane are reordered in a certain way to process them more efficiently. However, in our experiment, most of the tuples inside a vector were moved at a much larger distance than what FastLanes can reorder in its virtual register. Furthermore, the need for an additional index to store the original position of the tuples would decrease compression ratios.

Another factor that inhibits SIMD decoding is the variable bit-width of the encoded data resulting from Gorilla and its variants. Despite Patas always using a fixed size for

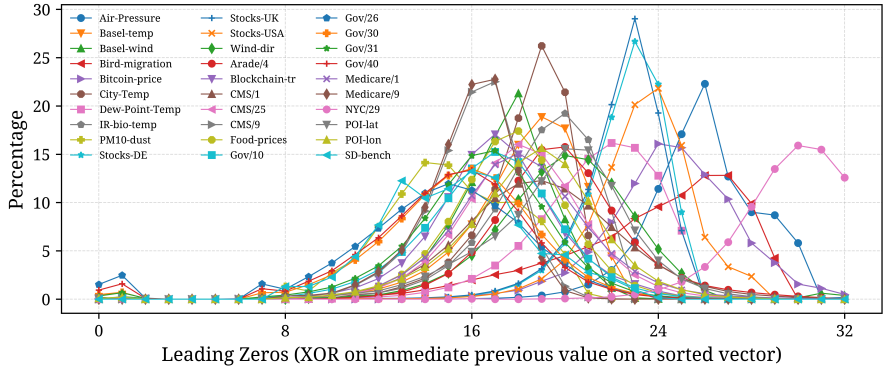

Figure 3.6: Distribution of Leading Zeros resulting from XORing each value with its previous immediate value on a block of values sorted in ascending order.

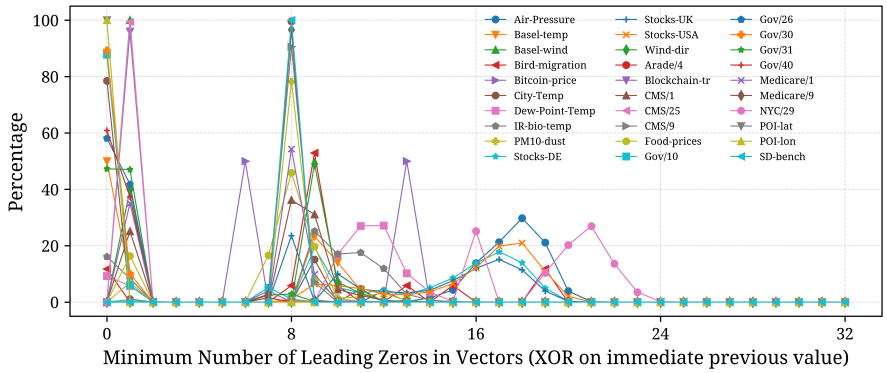

Figure 3.7: Distribution of the minimum number of leading zeros in vectors resulting from XORing each value with its previous immediate value.

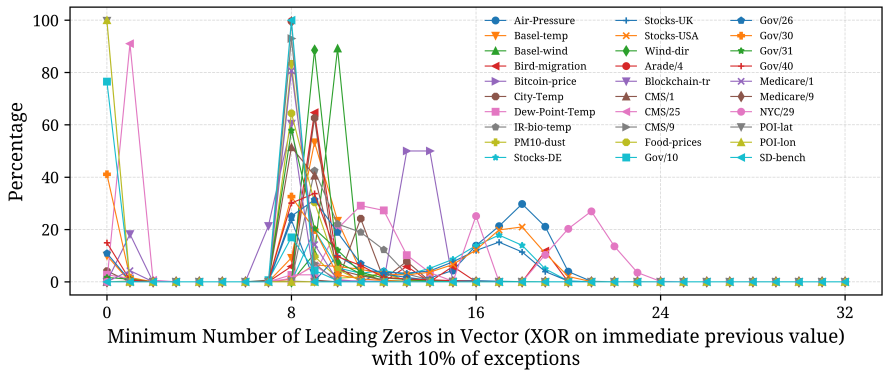

Figure 3.8: Distribution of the minimum number of leading zeros in vectors resulting from XORing each value with its previous immediate value by patching 10% of exceptions. Exceptions are XORed values inside a vector whose number of leading zeros is less than 8.

metadata, each value in the encoded data itself (i.e. significant bytes of the XOR) is still of variable bit-width.

A fixed bit-width of the encoded data would be achieved if, similar to BP, one packs all the values with the minimum bit-width needed to fit the biggest encoded value inside a vector. In such a scenario, an encoded value of 64 bits would ruin the bit-width of the entire vector and lead to negative compression (as the 64 bits would still have their associated metadata). Thus, at least one of the minimum number of leading and trailing zeros resulting from the XOR of all values inside a vector must be higher than 0 to not fall into negative compression.

We saw that in almost 100% of the vectors of every dataset, the minimum amount of trailing zeros for a vector is 0. Hence, trailing zeros are not reliable enough to achieve gains with a fixed bit-width. Such is not the case for leading zeros. Figure 3.7 shows the distribution of the minimum number of leading zeros present in every vector. Here we can see that in some datasets we could have a fixed bit-width of 52 bits in a great percentage of vectors (i.e. 64 - 8 bits), and even more in some datasets such as NYC/29, Air-pressure and the Stocks datasets. However, most of the datasets still have a relevant percentage of their vectors at around 0 and 2 minimum leading zeros. By tolerating around 10% of exceptions (XORed values inside a vector whose number of leading zeros is less than 8) the left part of the distributions resides in the range of 8 - 11 bits instead of 0 - 2 bits (Figure 3.8). However, the compression ratios that one could achieve by using this idea of fixed bit-width for the encoded data would be sub-optimal in comparison to Chimp128 and even Patas.

3.3 Decimal-based Analysis

From a human-readable perspective, two doubles are similar when their orders of magnitude and their visible decimal precision (as a string literal) are similar. On our time-series datasets, the standard deviation of the values magnitude (Table 3.2:C8) is relatively small (e.g. Stocks-USA, Dew-Point-Temp, Air-Pressure). In contrast, on non-time series data, this measure is elevated for some datasets (e.g. Food-Prices, Gov/40, CMS/9), without being orders of magnitude higher than the values magnitude average (Table 3.2:C7). The latter means that close values are similar in their orders of magnitude.

Decimal precision varies between datasets (Table 3.2:C2 and C3). For instance, datasets that contain coordinates such as POI-lat and POI-lon can vary between 0 and 20 decimals of precision. On the other hand, datasets such as Medicare/9, SD-bench and City-Temp

contain at most values with 1 decimal precision. Despite these differences in decimal precision inside a dataset, the deviation of this property is usually small from a vector perspective (Table 3.2:C5). In fact, for 25 out of 30 datasets, the decimal precision deviation inside vectors is smaller than 1. The latter means that most of the values inside a vector share the same decimal precision.

As we reviewed in section 2.5.3, decimal-based encoding approaches exploit these human-readable similarities of doubles by trying to represent them as integers. The more similar the decimal precision and the orders of magnitude of doubles inside a block of values, the more performance in terms of compression ratio can be achieved. In contrast to XOR-based approaches, these outputs of integers can be further compressed with integer LWC techniques. Additionally, having metadata regarding the precision bounds of the doubles is beneficial for algorithms such as BUFF (13) in order to perform arithmetic to get rid of unnecessary information in the bitwise representation of such numbers. However, we are not interested in designing an encoding limited by the availability of metadata of a block of doubles.

3.3.1 Representing Doubles as Integers

Representing double precision floating-point values as integers is non-trivial. Take for instance the number $\boldsymbol{n = 8.0605}$. At first glance, to encode $\boldsymbol{n}$ as an integer we could be tempted to move the decimal point $\boldsymbol{e}$ spaces to the right until there are no decimals left (i.e., 4 spaces). The latter can be achieved with the following procedure: P_{enc}:

$$P_{enc} = round(n \times 10^{e}) \tag{3.1}$$

Since one of the multiplication operands of P_{enc} is a double, we need to round the result to obtain an integer. Then, we could conclude that we have reduced our double precision floating point number into a 32-bit integer $\boldsymbol{d}$ = 80605 (i.e. result of P_{enc}) and another 32-bit integer representing the number of spaces $\boldsymbol{e}$ we moved the decimal point (i.e. a factor of 10).

Hence, from the encoded integer $\boldsymbol{d}$ result of P_{enc}, and the number of spaces $\boldsymbol{e}$ we moved the decimal point, we should be able to recover the original double by performing the following procedure P_{dec}:

$$P_{dec} = d \times 10^{-s} \tag{3.2}$$

Executing this in a programming language will *visually* yield on screen the original number 8.0605. However, the exact bitwise representation of the original double has been lost in the process. The correctness of the procedures fails to hold due to our number 8.0605 not being a **real double** (71). The real representation of the number 8.0605 as a double based on the IEEE 754 definition is: 8.060499999999993320 9 . To achieve lossless compression, this has to be the exact result of our procedure P_{dec}. However, in our example P_{dec} yields 8.0605000000000011084 . This is a consequence of the error introduced in the multiplication by the inverse factor of 10 in P_{dec}. The latter turns out to be a double that does not have an exact decimal representation either. Hence, 10^{-4} is not 0.0001 but more something like 0.000100000000000000002082. This error is introduced in the multiplication, and reflected in the end result of the procedure P_{dec}. The P_{enc} procedure does not suffer this problem since 10^e has an exact double representation for $\boldsymbol{e} \leq 21$.

Table 3.2:C11 depicts the percentage of doubles in each dataset that can be losslessly represented by an integer $\boldsymbol{d}$ and an exponent $\boldsymbol{e}$ using the P_{enc} and P_{dec} procedures. But, *always* using the *visible* precision of the doubles as the exponent $\boldsymbol{e}$ (e.g., for 0.**0001**, the visible precision is 4; for 1.**4297546**, the visible precision is 7). This results in only 82.5% of the values successfully encoded and decoded on average for all the datasets. However, in some datasets, the success probability gets as low as 61.7%. We found the success of the procedures P_{enc} and P_{dec} to encode and decode the exact original doubles to depend on two factors: (i) the *real precision* of the exponent $\boldsymbol{e}$ and (ii) the *visible precision* of the double $\boldsymbol{n}$.

3.3.2 High exponents work for all values

Table 3.2:C12 shows the exponent $\boldsymbol{e}$ which leads to the highest success-rate of P_{enc} and P_{dec} on each dataset. It is evident that higher exponents $\boldsymbol{e}$ such as 14 and 16 are predominant, with an average of 95% successfully encoded values in all of the datasets; and up to a rate of 99.9% in datasets such as SD-bench, Stocks-UK, Medicare/9, Gov/31 and PM10-dust. The effectiveness of higher exponents stems from the fact that the more we *increase the exponent* $\boldsymbol{e}$ the closer we can get to obtaining the real double with the procedures. This is due to higher exponents $\boldsymbol{e}$ resulting in a more precise inverse factor of 10 on P_{dec}. For instance, 10^{-14} represented as a double is equal to $1.00000000000000007771E^{-15}$. As a consequence, the result of P_{dec} to recover the real double is more accurate. Furthermore, higher exponents are powerful because they are able to cover a wider range of decimal precision. Moreover, as shown in Table 3.2:C13, when optimizing to use a different exponent $\boldsymbol{e}$ per vector, we reach an average of 97.2% of successfully encoded values in all the datasets.

Based on these results, *we question* whether a different exponent $\boldsymbol{e}$ for each value is needed – which is what PDE does.

Nevertheless, by using higher exponents $\boldsymbol{e}$ the integers resulting from the procedure P_{enc} transform into big integers (i.e. 64 bits). These high exponents that lead to big integers are not taken into account by PDE since they lead to a worse compression ratio than leaving the data uncompressed. The latter is due to the number of bits needed to store big integers (i.e. 64 bits) and the number of bits needed to store the exponent itself (i.e. 32 bits) being bigger than the uncompressed value (i.e. 64 bits for a double). However, some doubles found in datasets such as the ones in NYC/29, POI-lat and POI-lon are *only* representable as big integers.

3.3.3 The 52-bit limit for integers

Exponent $\boldsymbol{e} = 14$ is the most successful in most of the datasets to represent doubles as integers using P_{enc} and P_{dec}. We have shown that this is due to the difference between the exact value and the real value of 10^{-14} being too small to have an effect in P_{dec} result. However, there are two datasets in which even higher exponents $\boldsymbol{e}$ are needed (i.e. POI-lat, POI-lon). This is due to the visible precision of the double values inside those datasets being on average higher than 14 (Table 3.2:C4). As we explain subsequently, when the order of magnitude of a double $\boldsymbol{n}$ plus its visible decimal precision reaches 16, P_{enc} is prone to fail due to a limitation of the IEEE 754 doubles.

The multiplication inside P_{enc} yields a double due to having a double operand. Hence, before rounding, our resulting integer $\boldsymbol{d}$ is a double. However, there is a known limitation to the accuracy of the integer part of a double. Only the integers ranging from -2^{53} to 2^{53} (i.e. -9,007,199,254,740,992.0 to 9,007,199,254,740,992.0) can be exactly represented in the integer part of a double number. Going beyond this threshold is problematic. Between 2^{53} and 2^{54} (i.e. 18,014,398,509,481,984.0), only *even* integer numbers can be represented as doubles. Similarly, between 2^{54} and 2^{55} only *multiples of 4* can exist. Furthermore, doubles stop having a decimal part after 2^{53}. Hence, if a double multiplication yields a double higher than 2^{53}, results will be automatically rounded to the nearest existing double number. The latter happens in P_{enc} when the order of magnitude of the double plus the visible decimal precision reaches 16. Hence, representing a number as an integer could be impossible in these cases using P_{enc} and P_{dec}. Thus, the reason why Air-sensor, POI-lat and POI-lon achieve a relatively low successful encoding rate of 76.4%, 70.5% and 91.7% respectively. This is also the reason why we stated earlier that 10^e only has an exact double representation for $\boldsymbol{e} \leq 21$.

3.3.4 Division vs Multiplication

We can define a different procedure P_{enc_aux} as a variation of P_{enc} as follows:

$$P_{enc\_aux} = round(n/10^{-e}) \tag{3.3}$$

P_{enc_aux} performs a division with an inverse factor of 10 instead of multiplication with a factor of 10. The latter can help to slightly increase the success rate on some of these special cases. This happens since the error introduced in the inverse power of 10 helps to propagate the invisible decimals of the original double into the integer representation. For example, trying to encode the double $\boldsymbol{n} = 1.42975460000000009764$ with $\boldsymbol{e} = 16$ using P_{enc} and P_{enc_aux} will go as follows: Before rounding, P_{enc} would have yielded $d_1 = 14297546000000000.9764$ and P_{enc_aux} would have yielded $d_2 = 14297546000000001.27525$. Due to the IEEE 754 limitation, both d_1 and d_2 are automatically converted to the nearest representable double, which are only the even numbers on these orders of magnitude. Therefore, $d_1 = 14297546000000000$ and $d_2 = 14297546000000002$ (note here that the *round* instruction of both procedures is now redundant). However, only d_2 is able to yield the original double using P_{dec}. From our investigations, in these cases obtaining the right $\boldsymbol{d}$ is a mathematical coincidence, rather than a consistent phenomenon. In addition to this, division is an expensive operation in most computer architectures (50). Hence, avoiding it is key to achieving desirable encoding and decoding speeds.

3.3.5 Towards a SIMD Decimal-based encoding

In contrast to our XOR-based analysis, there are opportunities to implement a truly SIMD approach tailored for vectorized execution based upon decimal-based encodings. For instance, the in-vector commonalities that we have found can be exploited to store metadata per-vector instead of per-value. In the following section, we dive more in-depth into these unexploited opportunities.

3.4 Unexploited Opportunities

Both current state-of-the-art approaches to compress doubles (i.e. Chimp128 and PseudoDecimals) already exploit some of the properties analyzed in the previous subsections. However, we believe there is room for substantial improvement both in terms of compression ratio and (de)compression speed; especially in decimal-based encodings.

3.4.1 Vectorizing Decimal Encoding

In subsection 3.3.2 we demonstrated that it is possible to achieve near 100% success rate of our procedures P_{enc} and P_{dec} to encode doubles as two integers by using only one exponent $\boldsymbol{e}$ on a per-vector basis. The current state-of-the-art approach (i.e. PseudoDecimals) uses one individual exponent $\boldsymbol{e}$ per value. This not only increases the computational cost of finding such exponent per value but also decreases the compression ratios that can be achieved. Hence, if we exploit this in-vector commonality of exponents, we believe compression ratios can be greatly improved.

3.4.2 Use of Long Integers (Cutting trailing 0s with an extra multiplication)

In subsection 3.3.1 we demonstrated that high exponents $\boldsymbol{e}$ achieve the highest success rate on our procedures P_{enc} and P_{dec} to store doubles as integers. However, we also mentioned that using exponents such as 14 results in 64-bit integers being encoded. Despite this, we believe that using a unique exponent $\boldsymbol{e}$ per vector opens the opportunity to encode big integers without instantly falling *behind* in compression ratio against uncompressed values. This is due to the unique 32-bit exponent being amortized by all the values inside the vector. More interestingly, we have also observed that in some cases, these long integers yielded from P_{enc} can be exploited further due to their tails of 0's digits.

High exponents $\boldsymbol{e}$ in combination with low-precision decimals datasets (e.g. SD-bench, City-Temp, Stocks-UK, Wind-Dir, PM10-Dust) results in long integers which contains *tails* of repeated digits (e.g. $\boldsymbol{n} \approx 37.3$ and $\boldsymbol{e} = 14$, yields $P_{enc} = 3730000000000000$; $\boldsymbol{n} \approx 100.8333$ and $\boldsymbol{e} = 14$, yields $P_{enc} = 10083330000000000$). These tails of repeated digits will have the same length in datasets with low values magnitude variance and low decimal precision variance (e.g. SD-bench, City-Temp, PM10-Dust, Air-pressure). Cutting these tails with an extra multiplication with an inverse factor of 10, namely $\boldsymbol{f}$, results in a smaller integer that can be used to recover the big integer with the inverse operation (i.e. a multiplication with a factor $\boldsymbol{f}$ of 10). Hence, we can redefine P_{enc} and P_{dec} as follows:

$$ALP_{enc} = round(n \times 10^{e} \times 10^{-f}) \tag{3.4}$$

$$ALP_{dec} = d \times 10^{f} \times 10^{-e} \tag{3.5}$$

Based on our previous analysis done in subsection 3.3.1 we could suspect that this new multiplication with another inverse factor of 10 in ALP_{enc} would result in a new error

introduced in the procedure. However, in this scenario, the introduced error due to the non-exact representations of inverse factors of 10 produces zero side effects on both procedures. For example, with $\boldsymbol{n} \approx 8.0605$, $\boldsymbol{e} = 14$ and $\boldsymbol{f} = 10$, ALP_{enc} and ALP_{dec} will execute as follows:

$$ALP_{enc} = round(8.06049999999999933209 \times 10^{14} \times 10^{-10})$$
$$ALP_{enc} = round(806049999999999.875 \times 10^{-10})$$
$$ALP_{enc} = round(80604.999999999985448)$$
$$ALP_{enc} = d = 80605$$
$$ALP_{dec} = 80605 * \times 10^{10} \times 10^{-14}$$
$$ALP_{dec} = 806050000000000 \times 10^{-14}$$
$$ALP_{dec} = n = 8.06049999999999933209$$

As we can see in the third step of ALP_{enc}, the error introduced by 10^{-10} is negligible to the resulting integer $\boldsymbol{d}$. Using this reducing factor $\boldsymbol{f}$ in the procedures is a way of taking advantage of the high exponents coverage and success rates without having to encode big integers $\boldsymbol{d}$. Note that this example is the same $\boldsymbol{n}$ we used at the beginning of subsection 3.3.1, which could not be encoded by simply using $\boldsymbol{e} = 4$. Also, note how a tail composed of 9's digits can also be reduced without any side-effect. Despite the effectiveness of using a reducing factor, this comes at the expense of doing an additional operation at encoding and decoding. Plus, an additional 32-bit to store the factor amortized by all the values inside a vector.

3.4.3 Limited Search Space

As of now, we have ignored the process of *finding* the exponent $\boldsymbol{e}$ for our decimal-based encoding procedures ALP_{enc} and ALP_{dec}. The current state-of-art on decimal-based encoding (i.e. PseudoDecimals) performs a brute-force search for each value in a dataset in order to find the exponent $\boldsymbol{e}$. For our *ALP* procedures, an additional nested brute-force search needs to be performed in order to find the **best combination** of exponent $\boldsymbol{e}$ and factor $\boldsymbol{f}$. We define the *best combination* as the one in which ALP_{enc} yields the smallest integer $\boldsymbol{d}$ with which ALP_{dec} succeed in recovering the original double $\boldsymbol{n}$. This translates into a search space of 253 possible exponent $\boldsymbol{e}$ and factor $\boldsymbol{f}$ combinations (given that $\boldsymbol{f} \leq \boldsymbol{e}$ and $0 \geq \boldsymbol{e} \leq 21$). This is not SIMD-friendly or efficient. However, we have already demonstrated that most values inside a vector can be encoded by using one single exponent.

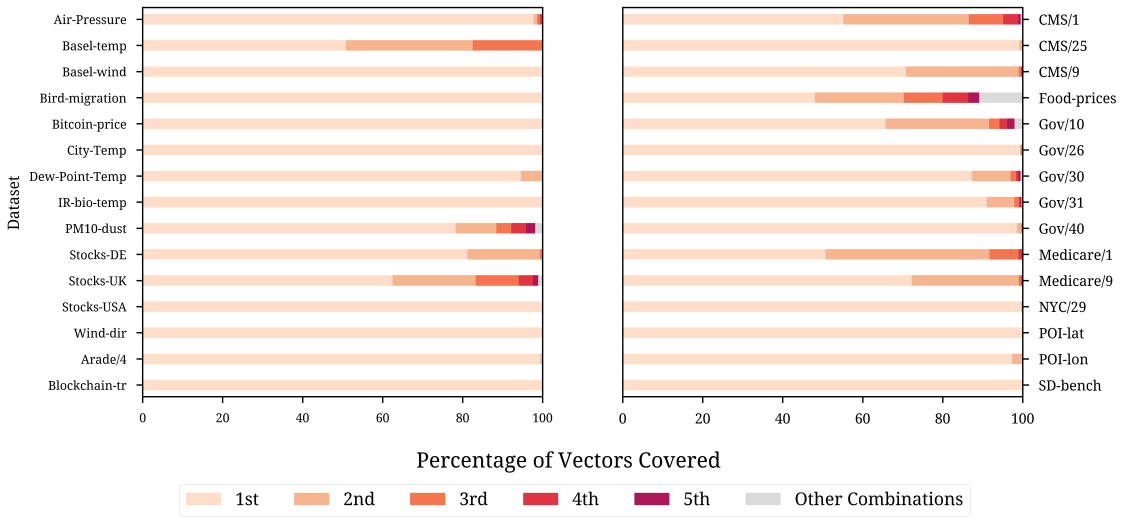

Figure 3.9: Analysis of the best combinations of exponent e and factor f for each vector of 1024 values. For most datasets, the best combination for any vector is found among a set of just 5 different combinations. For some datasets, a single combination is always the best one. A combination is the *best* when it achieves the highest encoding rates yielding the smallest integers using ALP_{enc} and ALP_{dec} procedures.

Furthermore, we have already shown that vectors exhibit a low variance in their decimal precision and in their values orders of magnitude. Hence, we believe that the search space for the combination of exponent e and factor f can be greatly reduced and it can be done on a per-vector basis. In order to prove this, we computed the best combination for each vector in each dataset by brute force. That is, for each vector we tested what is the combination that yielded the smallest integers from which the most amount of original doubles could be recovered. For this experiment, we define a `vector` as 1024 consecutive values from a dataset. Figure 3.9 shows the results of the experiment. We can see how for most datasets a search space of 5 combinations is enough to obtain the best combination among all vectors. Moreover, for some datasets such as Basel-wind, Bird-migration, City-Temp, Wind-dir and IR-bio-temp, an entire dataset search space is only one combination. Hence, performing a search among 253 possible combinations every time is unnecessary. Such optimization of the search space would greatly improve compression speed in contrast to the current state-of-the-art brute-force approach.

3.4.4 Front-Bits Similarity

When the magnitude plus decimal precision exceeds 16, it is often impossible to encode a double as an integer with our procedure ALP_{enc}. On such data, decimal-based encoding would have to deal with integers bit-packed to more than 52 bits (and similarly, Chimp variants would have to deal with trailing bit-strings of more than 52 bits). A basic observation is that such data is not very compressible in the first place (64-bit data takes at least 52 bits); nevertheless, compression may still be worthwhile.

We believe that the approach of a decimal-based encoding is not appropriate for such compression-unfriendly data; thus when encountering such data, our approach could adaptively switch to a different encoding strategy, that exploits regularities in the front-bits in a vectorized manner. In Table 3.2:C10, even on these datasets (i.e., POI-lat, POI-lon) we see that the exponent of the bitwise representation of a double exhibits a low variance. Data with low variance can be compressed with lightweight integer encodings, such as RLE and Dictionary – all building blocks provided by the FastLanes compression library (3). Furthermore, based on the analysis of leading 0-bits from XOR-ing with the previous value (Table 3.2:C14), on some of these datasets we should not limit this idea to just the exponent, because the highest bits of the mantissa often are regular (if the data stems from a particular value range).

4

Adaptive Lossless Floating-Point compression (ALP)

ALP is an adaptive lossless encoding designed to compress double precision floating-point data. ALP takes advantage of the opportunities discussed in subsection 3.3.1. Compression and decompression are built upon the ALP_{enc} and ALP_{dec} procedures described in section 3.4. Furthermore, ALP is able to *adapt* its encoding/decoding scheme if it encounters high precision doubles by taking advantage of the similarity in the front-bits uncovered in section 3.4.4. in both compression and decompression. In the following subsections, we describe the key design aspects of ALP and how it implements adaptivity.

4.1 Compression

ALP compression is built upon the ALP_{enc} procedure (Formula 3.4). ALP tries to encode all doubles $\boldsymbol{n}$ inside a vector $\boldsymbol{v}$ with the same exponent $\boldsymbol{e}$ and factor $\boldsymbol{f}$. Inside the encoding, ALP must verify that the procedures ALP_{enc} and ALP_{dec} yield the original double n. If the original double $\boldsymbol{n}$ cannot be recovered, we treat the double as an *exception*. Algorithm 4.1 shows the pseudo-code for ALP encoding.

4.1.1 Vectorized Compression

ALP introduces the use of one exponent $\boldsymbol{e}$ and factor $\boldsymbol{f}$ for all doubles inside the same vector. Note that PDE needs to store one exponent *per value* – taking more space. Based on our empirical investigation, in order for this approach to be successful we need to be able to use high exponents $\boldsymbol{e}$. Hence, ALP does not limit the encoded integers to `int32` representations, but `int64`. Furthermore, ALP incorporates the new idea of the factor $\boldsymbol{f}$

Algorithm 4.1: ALP Compression.

```
double i_F10 = {1.0, 0.1, 0.01, 0.001, ...};
double F10 = {1.0, 10.0, 100.0, 1000.0, ...};

// Adaptive search of exponent e and factor f in a vector
int e, f = ALP::ADAPTIVE_SAMPLING(input_vec, BEST_COMBINATIONS);

encoded_vec, exc_vec, exc_pos_vec = ALP::ENCODE([]() {
  for (i = 0; i < VECTOR_SIZE; ++i){ // Encode the vector
    double n = input_vec[i];
    int64 d = fast_double_round(n * F10[e] * i_F10[f]); // ALP_enc
    encoded_vec[i] = d;
    decoded_vec[i] = d * F10[f] * i_F10[e]; // ALP_dec
  }
  int exc_count = 0;
  for (i = 0; i < VECTOR_SIZE; ++i) { // Find Exceptions
    bool neq = (decoded_vec[i] != input_vec[i]);
    exc_pos_vec[exc_count] = i;
    exc_count += neq; // predicated comparison
  }
  int64 first_encoded = FIND_FIRST_ENCODED(exc_pos_vec);
  for (i = 0; i < exc_count; ++i){ // Fetch Exceptions
    encoded_vec[exc_pos_vec[i]] = first_encoded;
    exc_vec[exc_pos_vec[i]] = input_vec[i];
  }
});
FFOR(encoded_vec);
```

for reducing the trailing 0-digits, explored in subsection 3.4.2. After multiplying with the factor, the resulting integer is small again and is then *bit-packed* compactly, using the same number of bits for all values inside the same vector. The exponent, factor and bit-width parameters do not use much space, as these parameters are stored only once per vector (1024 doubles). The fact that all three parameters are the same per-vector also means that the [de]compression work is regular and thus has no control-instructions inside the loops, making them suitable for auto-vectorization.

4.1.2 Fast Rounding

The `round` operation is not supported in SIMD instruction sets. However, ALP replaces the `round` function with a procedure that takes advantage of the limitation of doubles to store exact integers of up to 52 bits, discussed earlier. An algorithmic trick resulting

from this limitation is that one can round a double by adding and subtracting 2^{52}. In other words, we take the doubles to the orders of magnitude in which they are not allowed to have a decimal part ($+2^{52}$). At this stage, the double is automatically rounded to its nearest integer without a need for an extra instruction. We named this procedure `fast_double_round`. This procedure is SIMD-friendly since it only consists of one addition and one subtraction; operations supported by SIMD intrinsics. For instance, to round a double n, `fast_double_round` will go as follows: `n_rounded = cast<int64>(n + magic_number - magic_number)`. The magic number is actually 6755399441055744.0, or 1.5×2^{52}, which is deemed as the perfect number to take doubles into this *sweet range* in which they cannot have a decimal part. This trick is not novel to ALP, since it is even used in *Lua* as a macro (`lua_number2int32`) to round numbers[1]. The use of `fast_double_round` can be seen in Algorithm 4.1: Line 10.

4.1.3 Handling Exceptions

Values which fail to be encoded as decimals become *exceptions*. Exceptions are stored uncompressed in a separate segment (i.e., `exc_vec` in Algorithm 4.1). However, since our approach is vectorized, we cannot simply skip the exceptions in the resulting vector of encoded values (i.e., `encoded_vec` in Algorithm 4.1). Hence, when exceptions occur we store an *auxiliary* value in the `encoded_vec` (i.e., `first_encoded` in Algorithm 4.1 Line 20). This auxiliary value is the first successfully encoded $\boldsymbol{d}$ which is obtained by the `FIND_FIRST_ENCODED` function in Algorithm 4.1: Line 20. Such value will not affect negatively the bit-width of the encoded vector. Note that by searching for this value after the encoding process we avoid an additional control statement in each iteration of the main encoding loop. Further, we also need to store in another storage segment the position in which each exception occurred within a vector (i.e., `exc_pos_vec` in Algorithm 4.1). For $\boldsymbol{v} = 1024$, each exception has an overhead of 80 bits: 64 bits for the uncompressed value and 16 bits to store the exception position. Lines 15 to 25 in Algorithm 4.1 show the exception handling process which is cleverly built to avoid control structures (i.e., `if-then-else`).

4.1.4 Fused Frame-Of-Reference (FFOR).

By itself, ALP encoding does not compress the data. Rather, it enables the use of lightweight integer compression to further encode its output. Based on our study of data similarity in subsection 3, we decided to encode the yielded integers using a Fused variant

[1] `http://www.lua.org/source/5.2/llimits.h.html`

of the Frame-Of-Reference encoding, available in the FastLanes library called `FFOR` (2). FastLanes (3) proposes a new data layout to accelerate the encoding and decoding of lightweight [de]compression methods with scalar code that auto-vectorizes. In FastLanes, `FFOR` fuses the implementation of bit-[un]packing with the `FOR` encoding and decoding process into a single kernel that performs both processes. The `FOR` encoding subtracts the minimum value of the integers in a vector; this will pick up on localized doubles (inside a tight range) and reduce bits needed in the subsequent bit-packing. Fusing saves a SIMD store and load instruction in between the subtraction and the bit-packing loop; improving the performance of the encoding. We note that it would also be possible to also fuse `FFOR` and ALP; this is not done yet here, and could provide a performance boost, especially in decoding. It is important to note that since FastLanes work in a fully vectorized execution, each encoded vector will have a fixed bit-width that is capable of fitting all the encoded values in it. We choose to use FastLanes due to its capabilities to perform on heterogeneous and evolving Instruction Set Architectures (ISAs). Hence, being future-proof and minimizing the technical debt by relying only on scalar code.

However, there is some more headroom as a modern compression library (e.g., (1, 3)) could try multiple different integers encodings and also *cascade* these. For instance, if the data is repetitive, one could use Dictionary coding, and compress the Dictionary with `FFOR`; or use RLE and then separately encode Run Lengths and Run Values. If the data is (somewhat) ordered, one could apply Delta encoding rather than `FFOR` to the Dictionary or the Run Values.

4.2 Adaptive Sampling

Our compression method does not perform a brute-force search for the exponent $\boldsymbol{e}$ and factor $\boldsymbol{f}$ to use in a vector. Instead, to find the best $\boldsymbol{e}$ and $\boldsymbol{f}$ for a vector, we designed a novel two-level sampling mechanism, inspired by the findings in subsection 3.4. Specifically, from Figure 3.9, we conclude that there is a *limited* set of best combinations of exponent $\boldsymbol{e}$ and factor $\boldsymbol{f}$ for the vectors in a dataset.

Our sampling mechanism goes as follows: on the first sampling level, ALP samples $\boldsymbol{m}$ equidistant values from $\boldsymbol{n}$ equidistant vectors of a `row-group`. We define a `row-group` as a set of $\boldsymbol{w}$ consecutive vectors of size $\boldsymbol{v}$. The total number of values obtained from this first sampling is equal to $\boldsymbol{m} \times \boldsymbol{n}$. For each vector $\boldsymbol{n_i}$ we find the *best* combination of exponent $\boldsymbol{e}$ and factor $\boldsymbol{f}$. This search is performed on the entire search space (i.e., 253 possible combinations). The *best* combination is the one that minimizes the sum of the exception

Algorithm 4.2: ALP Decompression.

```
1 int e, f = ALP::READ_VECTOR_HEADER(input_vec);
2 int64_vec = UNFFOR(input_vec);
3 decoded_vec = ALP::DECODE([](int64_vec) {
4   for (i = 0; i < VECTOR_SIZE; ++i){
5     decoded_vec[i] = int64_vec[i] * F10[f] * i_F10[e] }}); //ALP_dec
6 ALP::PATCH(decoded_vec, exc_vec, exc_pos_vec);
```

size and the size of the bit-packed integers resulting from the encoded $\boldsymbol{m}$ values. This process yields $\boldsymbol{n}$ combinations (one for each vector). From these $\boldsymbol{n}$ combinations we only keep the $\boldsymbol{k}$ ones which appeared the most. If two combinations appeared the same amount of times, we prioritize combinations with higher exponents and higher factors. It could be possible that fewer combinations than $\boldsymbol{k}$ are yielded. If the same best combination is found in every vector, there would only be 1 combination. Hence, we define during runtime a $\boldsymbol{k'}$ which is smaller than or equal to $\boldsymbol{k}$ that represents the number of yielded combinations. Once we have found the $\boldsymbol{k'}$ best combinations, we proceed to the second level of sampling.

The second level of sampling (Line 5 of Algorithm 4.1) samples $\boldsymbol{s}$ equidistant values from a vector. Then, it tries to find the combination of exponent $\boldsymbol{e}$ and factor $\boldsymbol{f}$ which performs the *best* on the $\boldsymbol{s}$ sampled values. However, this time, the search is performed only among the $\boldsymbol{k'}$ best combinations found from the first sampling level. To further optimize the search, we implemented a greedy strategy of early exit. If the performance of two consecutive combinations, namely k'_{i+1} and k'_{i+2}, is worse or equal to the performance of the combination k'_i, we stop the search and k'_i combination is selected to encode the entire vector. If $\boldsymbol{k'}$ is equal to 1, this second sampling level is omitted for all the vectors inside the row-group.

The first level of sampling is the most computationally demanding process of our compression scheme due to the large search space. However, it occurs only once per row-group. Hence, the time spent is amortized into $\boldsymbol{w} \times \boldsymbol{v}$ encoded values. The second sampling level happens once for each vector. Hence, it is amortized into $\boldsymbol{v}$ values. However, this second level of sampling will only occur if $k'_i > 1$. Hence, if the sampling parameters (i.e., m, n, w, k and s) are tuned optimally, the second sampling level will be skipped in datasets such as City-Temp or SD-bench, in which there exists only one best combination for all the vectors in the dataset (Figure 3.9).

Algorithm 4.3: ALP_{rd} Compression and Decompression.

```
// ENCODING //
p, DICT = ALP::RD::ADAPTIVE_SAMPLING(input_rowgroup);
left_vec, right_vec = ALP::RD::ENCODE([]() {
  for (i = 0; i < VECTOR_SIZE; ++i){
    double n = input_vec[i];
    left_vec[i], right_vec[i] = ALP::RD::CUT(p);}
});
BITPACK(right_vec);
SKEWDICT_BITPACK(left_vec, DICT);

// DECODING //
p, DICT = ALP::RD::READ_ROWGROUP_HEADER();
left_vec = BITUNPACK_DECODEDICT(encoded_left_vec, DICT);
right_vec = BITUNPACK(encoded_right_vec);
decoded_vec = ALP::RD::DECODE([]() {
  for (i = 0; i < VECTOR_SIZE; ++i){
    int16 left, int64 right = left_vec[i], right_vec[i];
    decoded_vec[i] = ALP::RD::GLUE(left, right, p);}
});
```

4.3 Decompression

ALP decompression builds upon the ALP_{dec} procedure (Formula 3.5) to recover the original doubles from a vector of integers d yielded by the encoding process. In order to do so, ALP first reads from the vector header the unique exponent e and factor f used to encode the vector. Then, ALP needs to reverse the FFOR integer encoding to recover each value. Values encoded as exceptions are directly read from the exception segment alongside their position on the original vector in order to correctly reconstruct it (i.e., patching). The pseudo-code of for ALP decoding is presented in Algorithm 4.2.

4.4 ALP for Real Doubles

During the first level of sampling ALP will detect whether the doubles in a row-group are not compressible. In that case, ALP encoding would result in a high number of exceptions and integers bigger than 2^{48}. Therefore, for such data, ALP changes its strategy to a different encoding approach based on the analysis performed in subsection 3.4.4 which hinted to us that even on these doubles, their front-bits tend to exhibit low variance. We named this approach ALP_{rd}, which stands for *ALP for Real Doubles*. ALP takes this

decision at the row-group level rather than the vector level since we found no dataset in which the decimal precision deviates on more than 3 decimals; hence taking this decision at a vector level would neither be efficient nor effective. We believe that the data in 28 of the 30 datasets analyzed originate as decimals and are thus not "real" doubles; however, we think that this is representative of the majority of data people store in data systems as doubles. The encoding and decoding of ALP_{rd} are presented in Algorithm 4.3.

4.4.1 Encoding

The first level of sampling finds at a `row-group` level which is the smallest position $p \geq 48$ where the highest 64-p front-bits still have low variance. Afterwards, it uses this number p as the position to *cut* the bits of every double of that `row-group` in two parts (Line 6 of Algorithm 4.3). The right part is compressed using p-bits bit-packing (`BP`). The position p is stored once per row-group (i.e., 8 bits of overhead per row-group, which can be safely ignored). At first glance, this method does not achieve any compression, however, the integers yielded from the left part are easily further compressible with integer lightweight encoding methods. For the version of ALP presented here, we compress them using a fixed method: skewed `DICTIONARY`+`BP` compression. A skewed dictionary is a `DICTIONARY` encoding which tolerates exceptions. Here, exceptions are values not in the dictionary, and these are stored as 16-bit values in an exception array, together with an array containing 16-bit exception positions. After sampling, we consider dictionaries of sizes 2^b with $b \leq 3$ (i.e., just 1, 2, 4, or 8 values), and fill these with the most frequent values in the sample and then choose the smallest dictionary size $b < 3$ such that the exception percentage does not exceed 10% (or else use b=3). We bit-pack the dictionary codes in b bits; and store the dictionary as 16-bit values. Both `BP` and skewed `DICTIONARY` encodings implementations are available in the FastLanes project[1]. These encodings implemented in FastLanes have already been demonstrated to achieve astonishingly fast performances.

4.4.2 Decoding

The b bits dictionary codes are bit-unpacked using a fast vectorized bit-unpacking primitive (that does this for the entire vector of 1024 values in one go) and (64-p) bits right parts of the doubles as well. Dictionary decompression requires one memory load from the dictionary for every code; which is relatively expensive. In SIMD it can be implemented with a `gather` instruction, but this is not supported on all CPU architectures nor does this

[1] `https://github.com/cwida/FastLanes`

instruction tend to be fast; hence we do not use such an approach (explicitly). Because we use small dictionaries of size $\leq 2^3 = 8$ and the front-bits are maximally 16 bits wide; we note that we could implement decoding by preloading the dictionary (maximally 8x16 bits values) in a 128-bits SIMD register and then use a `shuffle` instruction. However, the results presented in this paper are based on purely scalar dictionary decompression code, leaving space for improvement. Finally, we *glue* both parts together by left-shifting p bits the dictionary-decoded front-bits, after applying exception patching (34, 41) and adding in the decompressed right part (using vectorized SHIFT and OR, fused together in a `GLUE` primitive seen in Line 18 of Algorithm 4.3). Notice again that all operations are performed in a tight loop over arrays (vectorized query processing (17)) and the work is regular in nature such that C++ compilers get to very efficient code. Only the exception patching has some data dependencies and random memory access, but it is performed on a minority of the data only – limiting its performance effects.

5

Evaluation

We experimentally evaluate ALP with respect to its compression ratio and [de]compression speed using all analyzed datasets in Table 3.1 against six competing approaches for lossless floating-point compression: Gorilla (30), Chimp / Chimp128 (9), Patas (31), Elf (12) and PDE (1). Furthermore, we also compare against one general-purpose compression approach: Zstd (27). To further test the robustness of ALP we tested its speed on different hardware architectures which are described in Table 5.1 and using Auto-vectorized, Scalar and SIMDized code. In subsection 5.3 we present end-to-end query speed benchmarks of ALP on Tectorwise (72) to test its performance in a real system. Finally, in subsection 5.4 we present a version of ALP for 32-bit floats and evaluate it on machine learning data.

Sampling Parameters: Based on Figure 3.9, we defined the maximum number of combinations $\boldsymbol{k}$ as 5. The number of vectors $\boldsymbol{w}$ inside a `row-group` is fixed to 100 to emulate the usual modern OLAP engines row-group sizes (e.g., DuckDB (20)). The size of every vector $\boldsymbol{v}$ is fixed to 1024 to comfortably fit in the CPU cache (33). On the first sampling level, the number of vectors sampled per `row-group` $\boldsymbol{m}$ is set to 8, and the number of values sampled per vector $\boldsymbol{n}$ is set to 32. Finally, on the second sampling level, the number of values sampled per vector $\boldsymbol{s}$, is set to 32. $\boldsymbol{m}$, $\boldsymbol{n}$ and $\boldsymbol{s}$ were tuned during evaluation and showed to yield a good trade-off between compression ratio and speed.

Algorithms Implementations: ALP is implemented in C++ and is available in our GitHub repository `https://anonymous.4open.science/r/sigmod-E783`. Gorilla, Chimp, Chimp128 and Patas were implemented in C++. Gorilla was implemented by ourselves, and the other implementations were stripped from the DuckDB codebase (73) and adjusted to work as standalone algorithms. Note that Gorilla is part of a closed-source Facebook system. On the other hand, PDE and Elf[1] benchmarks were carried out using code from the

[1] `https://github.com/Spatio-Temporal-Lab/elf`

Table 5.1: Hardware Platforms Used

Architecture	Scalar ISA	Best SIMD ISA	CPU Model	Frequency
Intel Ice Lake	x86_64	AVX512	8375C	3.5 GHz
AMD Zen3	x86_64	AVX2 (256-bits)	EPYC 7R13	3.6 GHz
Apple M1	ARM64	NEON (128-bits)	Apple M1	3.2 GHz
AWS Graviton2	ARM64	NEON (128-bits)	Neoverse-N1	2.5 GHz
AWS Graviton3	ARM64	NEON (128-bits) SVE (variable)	modified Neoverse-V1	2.6 GHz

original authors. Finally, we used Facebook's implementation of zstd in C (27), configured at the default compression level (3).

5.1 Compression Ratios

Table 5.2 shows the compression ratios of all approaches measured in bits per value (uncompressed, each value is a 64-bit double). In this experiment the algorithms compressed *all* vectors in a dataset. The best-performing floating-point approach is marked in green.

ALP evidently stands out from the other floating-point encoding schemes in compression ratio. ALP shows an average improvement of ≈31% compared to PDE. When compared to Gorilla, Patas, Chimp, and Chimp128, ALP is respectively ≈49%, ≈39%, ≈43% and ≈24% better. In time series datasets, ALP achieves a ≈33% and ≈46% improvement over Chimp128 and PseudoDecimals. Similarly, on non-time series data ALP performs better than both by a ≈19% and ≈21% on average. Elf is ALP's most fierce competitor in terms of compression ratio – excluding zstd. On the other hand, zstd is the only compression algorithm that slightly takes the upper hand in compression ratio with 20.6 bits per value on average. Even so, ALP is slightly better than zstd on time series data. One has to take into account that zstd has a much lower [de]compression speed and, being block-based, has the disadvantage that one cannot optimally skip through compressed data. For instance, in zstd's 256KB block-based compression, a system has to decompress 32 8KB vectors, even if 31 of those 32 vectors are not needed.

5.1.1 When ALP shines

ALP outperforms Chimp128 and Elf on datasets with fixed or low decimal precision or with a low percentage of repeated values (e.g., Blockchain-tr, Arade-4, Dew-Point-Temp, Bitcoin-price). In other words, ALP gets its best gains when the doubles were generated

from *decimals*. ALP performs better than Chimp128 in 27 out of 30 datasets, and better than PDE in the same amount. In fact, ALP is at most 2 bits worse than PseudoDecimals on CMS/9 and Medicare/9. Both these datasets contain mostly integers encoded as doubles (Table 3.1). PDE benefits from such data since 0 bits are stored after applying BP to the exponents output due to the exponents always being equal to 0. Nevertheless, on these types of datasets Decimal-based encoding approaches are much better than XORing approaches. When ALP encounters *real doubles*, ALP_{rd} comes into the equation. There are two datasets for which ALP failed to achieve any compression and ALP_{rd} encoding was used: POI-lat and POI-lon (marked with *). These datasets are characterized by almost 0% of repeated values and a maximum decimal precision of 20 (Table 3.2:C2). In both datasets, these compression ratios achieved by ALP_{rd} represent an improvement over all the other floating-point compression approaches.

5.1.2 When ALP struggles

ALP struggles to keep up with both Elf and Chimp128 on datasets in which the XORing process benefits from a high percentage of repeated values and the decimal-based encoding process is hindered by high variability in value precision. Those datasets are: CMS/1, Medicare/1 and NYC/29. Despite ALP encoding also taking advantage of similar data, the profit of Chimp128 / Elf when it can find an exactly equal value is much higher than the profit that ALP can get. Nevertheless, on data with many duplicates, we question whether floating-point encodings were the best decision in the first place. For instance, due to the high percentage of repeated values we could plug in a `DICTIONARY` encoding before applying a floating-point encoding (or RLE, if the repeats are consecutive). We in fact tried using `DICTIONARY` and then compressing the dictionary with ALP, allowing it to achieve 33.1, 35.7 and 24.7 bits per value for CMS/1, Medicare/1 and NYC/29 respectively. The compression ratios that ALP is able to achieve by cascading compression using another lightweight encoding (i.e., `DICTIONARY` or `RLE`) are shown in the penultimate column of Table 5.2. By doing so, ALP even beats zstd in compression ratios while still retaining its advantages (higher speed, compatibility with predicate-pushdown).

5.2 [De]compression Speed Microbenchmarks

We measured speed as the amount of tuples (i.e., values) that an algorithm is capable of [de]compressing in one CPU clock cycle. In order to do so we took a vector within each of our datasets (i.e., 1024 values) and executed the [de]compression algorithms. The

Table 5.2: Compression ratio measured in Bits per Value. The smaller this metric, the more compression is achieved (uncompressed data is 64 bits per value). ALP achieves the best performance in average (excluding zstd). *: ALP_{rd} was used.

Dataset	Gor.	Ch.	Ch. 128	Patas	PDE	Elf	ALP	LWC+ ALP	Zstd
Air-Pressure	24.7	23.0	19.3	27.9	30.2	10.5	16.5	11.9^{dict}	8.7
Basel-Temp	61.6	54.1	31.2	36.5	39.3	32.9	29.8	13.8^{dict}	18.3
Basel-Wind	63.2	54.7	38.4	48.9	35.1	34.5	29.8	10.3^{dict}	14.6
Bird-Mig	48.7	41.9	26.3	35.9	35.2	19.9	20.1	19.8^{dict}	21.0
Btc-Price	51.5	48.2	45.1	57.1	44.1	31.9	26.4	26.4	49.9
City-Temp	59.7	46.2	23.0	24.2	31.5	15.1	10.7	10.0^{dict}	16.2
Dew-Temp	56.2	51.8	32.6	39.0	29.5	17.7	13.5	13.5	20.9
Bio-Temp	51.9	46.3	18.9	22.9	23.4	13.0	10.7	10.7	14.5
PM10-dust	27.7	24.4	13.7	19.9	12.9	7.1	8.2	8.2	6.9
Stocks-DE	46.9	42.9	13.6	20.8	25.1	12.3	11.0	11.0	9.4
Stocks-UK	35.6	31.3	16.8	21.5	26.1	11.0	12.7	12.7	10.7
Stocks-USA	37.7	35.0	12.2	19.2	26.1	8.8	7.9	7.9	7.8
Wind-dir	59.4	53.9	27.8	28.2	31.5	22.1	15.9	15.9	24.7
TS AVG.	**48.1**	**42.6**	**24.5**	**30.9**	**30.0**	**18.2**	**16.4**	**13.2**	**17.2**
Arade/4	58.1	55.6	49.0	59.1	33.7	30.8	24.9	24.9	33.8
Blockchain	65.5	58.3	53.2	62.6	39.1	39.2	36.2	36.2	38.3
CMS/1	37.8	34.8	28.2	36.8	40.7	25.4	35.7	33.1^{dict}	24.5
CMS/25	65.4	59.5	57.2	70.1	63.9	48.6	41.1	27.1^{rle}	56.5
CMS/9	17.1	18.7	25.7	26.0	9.7	15.8	11.7	11.3^{dict}	14.7
Food-prices	40.8	28.0	24.7	28.3	25.4	16.8	23.7	23.7	16.6
Gov/10	58.1	45.7	34.2	35.9	35.6	30.1	31.0	31.0	27.4
Gov/26	2.4	2.3	9.3	16.2	0.9	4.2	0.4	0.2^{rle}	0.2
Gov/30	10.3	8.9	12.9	19.3	8.2	8.0	7.5	6.2^{rle}	4.2
Gov/31	5.7	5.0	10.4	17.1	2.8	5.4	3.1	2.5^{rle}	1.5
Gov/40	2.7	2.6	9.4	16.4	1.2	4.3	0.8	0.5^{rle}	0.4
Medicare/1	45.9	42.7	32.3	39.9	42.8	29.9	39.4	35.7^{dict}	28.7
Medicare/9	17.9	19.1	26.0	26.3	10.2	16.0	12.3	11.3^{dict}	14.9
NYC/29	30.8	29.6	28.7	38.8	69.3	32.6	40.4	24.7^{dict}	20.5
POI-lat	66.0	57.7	57.5	71.7	69.3	62.5	55.5*	55.5*	48.1
POI-lon	66.1	63.4	63.1	75.9	69.2	68.7	56.4*	56.4*	53.1
SD-bench	51.1	45.7	19.2	23.0	30.6	18.4	16.2	12.0^{dict}	11.8
NON-TS	**37.7**	**34.0**	**31.8**	**39.0**	**32.5**	**26.9**	**25.7**	**23.1**	**23.3**
ALL AVG.	**42.2**	**37.7**	**28.7**	**35.5**	**31.4**	**23.1**	**21.7**	**18.8**	**20.6**

Table 5.3: Average compression and decompression speed as tuples processed per computing cycle of all datasets on the Ice Lake architecture.

Algorithm	Tuples per CPU Cycle (Higher is better)			
	Compression	ALP is faster by:	Decompression	ALP is faster by:
ALP	0.487	-	2.609	-
Chimp	0.042	12x	0.039	66x
Chimp128	0.040	12x	0.040	65x
Elf	0.010	47x	0.012	215x
Gorilla	0.052	9x	0.047	55x
PDE	0.002	251x	0.387	7x
Patas	0.060	8x	0.157	17x
Zstd	0.035	14x	0.101	26x

measure *tuples per cycle* is then calculated as 1024 divided by the number of computing cycles the process took. We chose one vector as the size of the experiment since every float compressor we compare against is optimized to work over a small block of values at a time; except Zstd. As such, we increased the size of the experiment for Zstd to one rowgroup (i.e. roughly 1 MB of data). In order to correctly characterize CPU cost, we repeated this process 300K times and averaged the result, to ensure all data is L1 resident. In this experiment, we prefer the metric *tuples per cycle* over *elapsed time* since it is a more effective comparison method across platforms. Furthermore, this metric makes Zstd speed measurements comparable regardless of the input data size. This experiment was performed on Ice Lake.

Table 5.4: Compression and decompression speed as tuples processed per computing cycle of every datasets on the Ice Lake architecture. ALP is faster in all datasets in both [de]compression. Zstd can't compress all datasets due to lack of sufficient data (< 1MB). *: ALP_{rd} was used.

Dataset	Compression (Tuples per CPU Cycle)								Decompression (Tuples per CPU Cycle)							
	ALP	**Chimp**	**Chimp128**	**ELF**	**Gorilla**	**PDE**	**Patas**	**Zstd**	**ALP**	**Chimp**	**Chimp128**	**ELF**	**Gorilla**	**PDE**	**Patas**	**Zstd**
Air-Pressure	0.516	0.033	0.029	0.012	0.030	0.001	0.063	0.008	1.947	0.030	0.028	0.011	0.033	0.397	0.192	0.051
Basel-temp	0.468	0.025	0.027	0.006	0.031	0.001	0.061	0.012	1.545	0.024	0.027	0.008	0.028	0.247	0.149	0.050
Basel-wind	0.482	0.025	0.025	0.007	0.023	0.001	0.062	0.014	1.606	0.024	0.025	0.008	0.030	0.316	0.162	0.051
Bird-migration	0.499	0.033	0.041	0.006	0.027	0.001	0.061	-	1.701	0.025	0.041	0.010	0.033	0.421	0.177	-
Bitcoin-price	0.489	0.030	0.023	0.007	0.030	0.001	0.062	0.210	1.636	0.024	0.024	0.008	0.031	0.167	0.104	0.772
City-Temp	0.505	0.027	0.043	0.010	0.026	0.001	0.055	0.015	1.769	0.024	0.043	0.012	0.028	0.293	0.218	0.047
Dew-Point-Temp	0.499	0.029	0.026	0.009	0.027	0.001	0.061	0.009	1.714	0.025	0.026	0.009	0.032	0.414	0.124	0.047
IR-bio-temp	0.509	0.029	0.040	0.010	0.026	0.001	0.060	0.015	1.790	0.024	0.040	0.010	0.032	0.424	0.198	0.046
PM10-dust	0.530	0.036	0.056	0.012	0.041	0.003	0.057	0.020	1.904	0.028	0.055	0.011	0.042	0.440	0.177	0.049
Stocks-DE	0.507	0.030	0.051	0.009	0.029	0.002	0.059	0.017	1.771	0.024	0.051	0.009	0.032	0.424	0.175	0.054
Stocks-UK	0.505	0.037	0.044	0.010	0.048	0.002	0.058	0.016	1.786	0.044	0.045	0.015	0.057	0.466	0.170	0.046
Stocks-USA	0.505	0.043	0.061	0.012	0.045	0.003	0.058	0.020	1.803	0.030	0.061	0.011	0.045	0.416	0.169	0.052
Wind-dir	0.519	0.025	0.040	0.008	0.025	0.001	0.060	0.008	1.964	0.023	0.040	0.009	0.032	0.425	0.089	0.043
TS. AVG.	**0.503**	**0.031**	**0.039**	**0.009**	**0.031**	**0.001**	**0.060**	**0.030**	**1.764**	**0.027**	**0.039**	**0.010**	**0.035**	**0.373**	**0.162**	**0.109**
Arade/4	0.498	0.025	0.023	0.007	0.032	0.001	0.062	0.007	1.753	0.024	0.023	0.008	0.030	0.468	0.073	0.040
Blockchain-tr	0.489	0.024	0.026	0.006	0.024	0.001	0.062	-	1.612	0.028	0.026	0.007	0.031	0.211	0.146	-
CMS/1	0.473	0.025	0.023	0.006	0.030	0.001	0.062	0.009	1.497	0.023	0.023	0.006	0.030	0.146	0.100	0.047
CMS/25	0.487	0.024	0.022	0.007	0.022	0.001	0.056	0.008	1.550	0.023	0.022	0.006	0.030	0.426	0.156	0.049
CMS/9	0.507	0.034	0.042	0.009	0.045	0.002	0.058	0.014	1.776	0.043	0.042	0.015	0.053	0.453	0.164	0.034
Food-prices	0.474	0.047	0.044	0.010	0.044	0.001	0.060	0.011	1.591	0.041	0.044	0.016	0.046	0.307	0.173	0.044
Gov/10	0.427	0.028	0.034	0.007	0.025	0.001	0.059	0.008	1.504	0.027	0.035	0.008	0.028	0.191	0.114	0.040
Gov/26	0.618	0.116	0.073	0.021	0.200	0.006	0.062	0.236	9.261	0.117	0.073	0.026	0.120	0.620	0.174	0.362
Gov/30	0.614	0.111	0.071	0.023	0.158	0.004	0.061	0.029	8.868	0.109	0.071	0.026	0.120	0.341	0.163	0.113
Gov/31	0.618	0.115	0.072	0.025	0.170	0.006	0.062	0.111	9.193	0.113	0.072	0.027	0.124	0.466	0.172	0.237
Gov/40	0.619	0.114	0.072	0.025	0.176	0.006	0.062	0.101	9.025	0.113	0.072	0.027	0.123	0.618	0.173	0.243
Medicare/1	0.464	0.031	0.031	0.007	0.032	0.001	0.061	0.009	1.498	0.026	0.031	0.008	0.037	0.173	0.163	0.045
Medicare/9	0.507	0.033	0.042	0.009	0.044	0.002	0.057	0.014	1.774	0.041	0.042	0.015	0.052	0.502	0.159	0.034
NYC/29	0.488	0.052	0.043	0.009	0.054	0.003	0.060	0.009	1.556	0.035	0.043	0.013	0.052	0.462	0.176	0.047
POI-lat*	0.142	0.024	0.023	0.006	0.026	0.001	0.059	0.007	0.574	0.024	0.023	0.007	0.028	0.462	0.154	0.045
POI-lon*	0.158	0.022	0.021	0.005	0.026	0.001	0.049	0.007	0.590	0.022	0.021	0.007	0.028	0.461	0.165	0.044
SD-bench	0.499	0.028	0.043	0.009	0.031	0.003	0.060	-	1.710	0.026	0.043	0.013	0.033	0.442	0.166	-
NON-TS	**0.475**	**0.050**	**0.042**	**0.011**	**0.067**	**0.002**	**0.060**	**0.039**	**3.255**	**0.049**	**0.042**	**0.014**	**0.057**	**0.397**	**0.152**	**0.095**
ALL AVG.	**0.487**	**0.042**	**0.040**	**0.010**	**0.052**	**0.002**	**0.060**	**0.035**	**2.609**	**0.039**	**0.040**	**0.012**	**0.047**	**0.387**	**0.157**	**0.101**

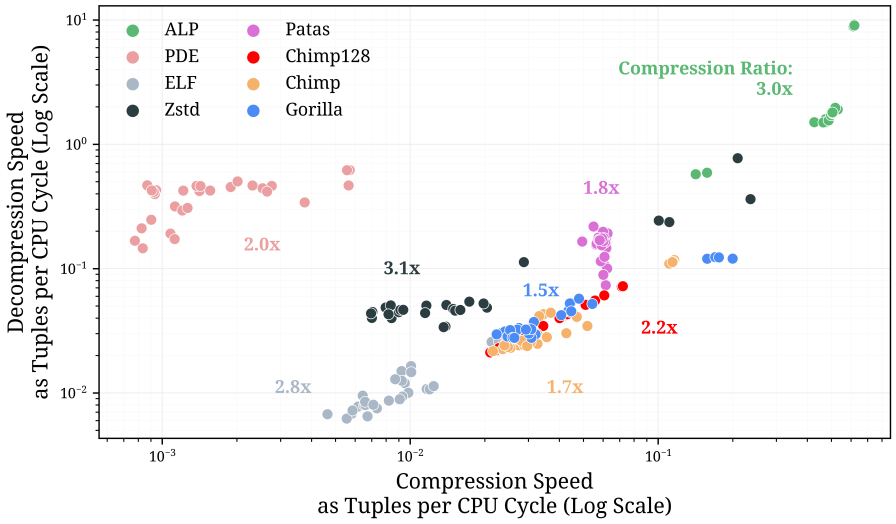

Figure 5.1: Compression performance for all schemes (on Intel Ice Lake). Each dot is one dataset. ALP is 1-2 orders of magnitude faster in [de]compression than all competing schemes, while providing excellent compression ratio. The only one to achieve a compression ratio similar to ALP is zstd, but it is slow and block-based (one cannot skip through compressed data). Elf is inferior to zstd on all performance metrics.

Figure 5.1 shows the result of this experiment aggregated by all datasets. ALP clearly outperforms every other algorithm in both compression and decompression speed in every dataset; even being able to achieve sub-cycle performance in decompression. This speed measurement also includes the FFOR encoding and decoding in ALP. Table 5.3 shows the average amount of tuples per cycle processed in compression and decompression for every algorithm along all datasets. ALP is the fastest of all other approaches in both compression and decompression.

ALP is ≈7x faster than PDE; which is the second-best at decompression speed. However, PDE is also the slowest at compression (251x slower than ALP) due to the brute force and –per value– search for a viable exponent e to encode the doubles as integers. Furthermore, ALP is ≈8x faster than Patas, which is the second-best at compression speed. This was expected since Patas is a single-case-byte-aligned variant of Chimp optimized for decoding speed. On the other hand, Elf speed under-performed against the other algorithms, with ALP being ≈47x times faster in encoding and ≈215x faster in decoding. This was also expected since Elf is a variant of Gorilla tailored to trade speed for more compression ratios. Hence, the fact that ALP achieved higher compression ratios than Elf is remarkable. ALP is x55 faster than Gorilla at decompression since the latter has complex if-then-else (i.e. branch mispredictions) and data dependencies that not only cause wait cycles, but also

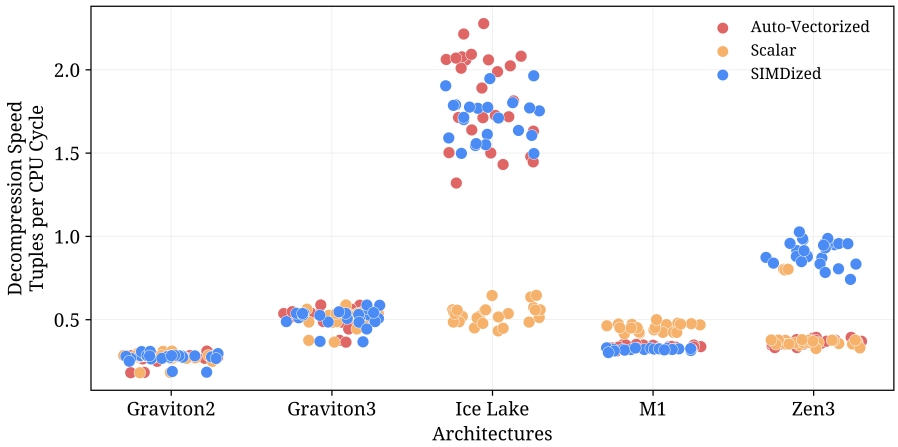

Figure 5.2: Decompression speed measured in tuples per cycle on different architectures. Each dot represents the decompression performance on a dataset in a different architecture.

prevent SIMD. Zstd resides in a middle position in that it achieves better compression speed than PDE and Elf, and decompression speed only slower than Patas and PDE.

It is important to mention how ALP is faster on some datasets than others. Table 5.4 shows the speed of ALP [de]compression against all other schemes disseminated by every dataset. These differences in speed between datasets are due to multiple factors. For instance, the most impactful one is the bit-width of the resulting integer-encoded vectors; as a bigger bit-width will result in slower execution (more information to pack/load, use of wider lane). Another factor is the number of exceptions found in the data. Exceptions negatively impact the speed of ALP since they have to be looked up after the main encoding and used to reconstruct the vector at decoding. Finally, ALP speed also depends on the amount of k combinations resulting from the first sampling phase. However, ALP decompression is always on sub-cycle performance, with outliers of astonishing speed on the datasets with a bit-width of 0.

5.2.1 ALP on Different Architectures

In order to investigate the performance robustness of ALP, we evaluated it on all currently mainstream CPU architectures, as described in Table 5.1. CPU turbo-scaling features were disabled when available to allow for reliable tuples-per-cycle measurements. In our presentation here we just show results for decompression speed (due to space reasons) as this is the most performance-critical aspect for analytical database workloads. Furthermore, on each architecture, we tested three different implementations of our decoding procedure: SIMDized, Auto-vectorized and Scalar. The SIMDized implementation uses explicit

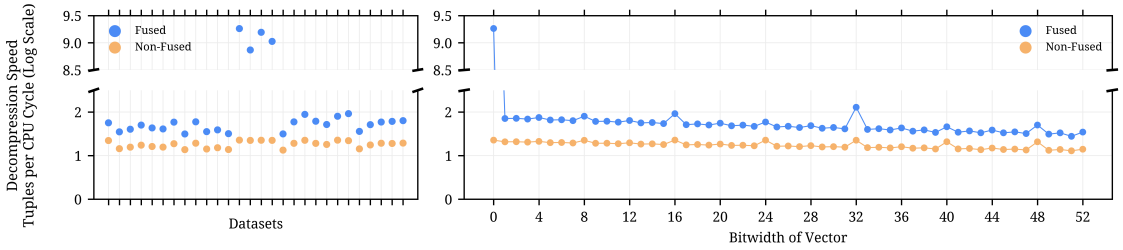

Figure 5.3: Speed comparison of ALP decoding with and without fusing ALP and FFOR into one single kernel (Ice Lake). Tests performed on our analyzed datasets (top) and on generated data with specific vector bit-width (bottom). ALP benefits from fusing consistently with a ≈40% decompression speed increase (and sometimes much more).

SIMD intrinsics. The Auto-vectorized implementation is the Scalar implementation automatically vectorized by the C++ compiler. Finally, the purely Scalar implementation is obtained when we explicitly disabled the auto-vectorization of the C++ compiler by using the following flags: `-O3 -fno-slp-vectorize -fno-vectorize`. Figure 5.2 shows the results of this experiment. We can see how Auto-vectorized and SIMDized on Ice Lake yield the best performance results. This is due to this platform having the widest SIMD register of all the platforms at 512-bits. We can also see that Gravitons have weak SIMD performance (compared to Scalar). Furthermore, in every platform, Auto-vectorization matches or surpasses Scalar code. However, Zen3 auto-vectorized performance is hurt by the scalar code using the built-in rounding function due to the lack of a SIMD instruction to perform the cast from double to int64 in our fast rounding procedure. This is a result of the robustness obtained using the FastLanes library in our implementation. The latter makes our ALP implementation portable and able to perform on heterogeneous ISAs.

5.2.2 Kernel Fusion

We performed speed comparisons of our decompression between `FFOR+ALP` as a fused kernel and as two separate kernels. The plot at the top of Figure 5.3 shows the result of this experiment. Fusing increases the decompression speed by a median ≈40% (but for some datasets 6x). However, the vectors from our datasets used for this experiment do not cover all the possible bit-widths that FFOR could use. The latter is a known factor that may affect the performance of vectorized execution (36). Hence, for robustness purposes, we performed an additional comparison on synthetic integer vectors generated with a specific vector bit-width from 0 to 52. Bit-widths from 52 to 64 are omitted from this analysis

since on these bit-widths ALP_{rd} is used. The bottom plot of Figure 5.3 shows the result of this experiment.

5.2.3 Sampling Overhead

ALP implements a two-level sampling mechanism to find the correct encoding method and parameters, described in section 4.2. The first level samples `row-groups` and the second level is done for every `vector`. We analyze the performance cost of the second sampling level, since it is on the performance-critical path of ALP compression.

When the first level sampling yields only one potential combination (e.g., Bird-Migration, Bitcoin-Price), there is 0 sampling overhead at a vector level for the entire `row-group` since ALP already knows which combination of exponent and factor to use for all the vectors. This occurs on ≈54% of the vectors in our datasets. However, when ALP has to perform the second level sampling, there is a non-negligible overhead at compression. From our experiments, this overhead represents on average ≈6% of the total compression time. The latter is a trade-off for fast decompression; which in the context of analytical databases is a more often-used operation than compression. This overhead is bounded by k factor and exponent combinations, which was set to 5 in our evaluation. 22.9% and 20.0% of the vectors tried 2 and 3 combinations respectively in search of the best one. Only 2.9% and 0.3% of the vectors tried 4 and 5 combinations respectively on the vector sample.

Finally, we question how effective our adaptive two-level sampling is. In order to do so, we compute the best combination for each vector by brute-force search and compare the compression ratios of both approaches (i.e., adaptive sampling and brute-force). In other words, we set the second-level sample size equal to the values inside a vector (i.e. 1024) and tested all the possible 253 combinations of e and f. We found that the combinations yielded by the brute-force approach only improved the compression ratio by less than 1%. Thus, demonstrating the efficiency and portability of our fixed sampling parameters across all datasets.

5.2.4 ALP_{rd} speed.

Doing a side-by-side comparison ALP_{rd} is on average ≈3x slower in compression and ≈4x slower in decompression than the main ALP encoding. In fact, the two datasets in which ALP_{rd} was used can be seen at the bottom of ALP green dots in Figure 5.1. Although ALP_{rd} is still remarkably performant compared to the competitors, we deem this speed reduction necessary to achieve compression on these types of doubles, which present problems

for every floating-point compression scheme. We believe there is room for improvement since ALP_{rd} encoding and decoding are not fused into one single kernel due to current implementation limitations. However, given that [de]compression in almost any encoding gets faster at high compression ratios, this result is not surprising: ALP_{rd} is used when only low compression ratios can be achieved (maximum ≈1.2x).

5.3 End-to-End Query Performance

We benchmarked end-to-end query speed of ALP and the other floating-point compressors, when integrated in the research system Tectorwise (72). The difference with our micro-benchmarks is that a complete dataset is decompressed by Tectorwise's scan operator (SCAN), rather than only a small part. Also, in the SUM experiment, the scan operator feeds data vector-at-a-time into an aggregation operator; using the vectorized query execution of Tectorwise. We scaled all datasets up to 1 billion doubles by concatenation (8GB uncompressed). We also test compression performance, which writes the compressed data. This also writes extra meta-data for the compressed blocks, at the least byte-offsets where they start, but for PDE and ALP also offsets where their exceptions start, as well as any other compression parameters (like bit-width for bit-packing).

For presentation purposes, we picked five datasets with diverse characteristics, such as magnitude, decimal precision, XORed 0's bits, and compressability. These datasets are: Gov/26, City-Temp, Food-Prices, Blockchain-tr and NYC/29. We benchmarked 3 queries: COMPRESSION (`COMP`), `SCAN` and `SUM` (Aggregation). For `SUM` and `SCAN` we also benchmarked the scaling of every algorithm when using multiple cores (up to 16). This experiment was again carried out on Intel Ice Lake in a machine with 16 cores (32 SMT), 256GB of RAM with a bandwidth of 18.75 Gibps. The reported results are the average of 32 executions of one query. Elf was not included in this analysis due to the lack of an implementation in C++.

5.3.1 SUM and SCAN

Table 5.5 shows that in the single-threaded `SCAN | 1` experiment, the achieved 1.33 Tuples per CPU cycle is in line with the microbenchmarks shown in Figure 5.3 – though there is about a 25% drop in performance in the end-to-end situation compared to these. We attribute this to: (i) the extra effort in reading block meta-data (not present in the micro-benchmarks), (ii) the interpretation cost of choosing and calling a decompression function based on the meta-data (always the same and thus free of CPU branch mispredictions in

Table 5.5: End-to-end performance on City-Temp in the Tectorwise system, measured in Tuples per CPU cycle per core. ALP is even faster than uncompressed, and extends its lead w.r.t. the micro-benchmarks. The competitors are so CPU bound that they scale well in SCAN (=speed stays equal), while ALP and uncompressed drop speed when running multi-core, due to scarce RAM bandwidth. But when doing query work (SUM), speed is lower, and scaling is not an issue for ALP.

Algorithm	**Tuples per CPU Cycle (Higher is Better)** **QUERY | THREADS**						
	SCAN|1	**SCAN|8**	**SCAN|16**	**SUM|1**	**SUM|8**	**SUM|16**	**COMP**
ALP	1.337	1.074	0.882	0.233	0.230	0.234	0.147
Uncompressed	0.565 [x2 slower ↓]	0.408	0.350	0.197 [x1.2 ↓]	0.186	0.175	N/A
PDE	0.070 [x19 ↓]	0.071	0.071	0.058 [x4 ↓]	0.057	0.057	0.001 [x138 ↓]
Patas	0.067 [x20 ↓]	0.063	0.065	0.055 [x4 ↓]	0.055	0.055	0.039 [x4 ↓]
Gorilla	0.030 [x44 ↓]	0.030	0.030	0.028 [x8 ↓]	0.027	0.027	0.023 [x7 ↓]
Chimp	0.021 [x64 ↓]	0.021	0.021	0.019 [x12 ↓]	0.019	0.019	0.015 [x10 ↓]
Chimp128	0.028 [x47 ↓]	0.028	0.028	0.026 [x9 ↓]	0.026	0.026	0.019 [x8 ↓]
Zstd	0.044 [x31 ↓]	0.042	0.039	0.038 [x6 ↓]	0.037	0.035	0.014 [x11 ↓]

the micro-benchmarks) and (iii) the variable amount of exceptions present in the entire dataset.

Given these extra activities in end-to-end, and just a 25% drop, we deem our micro-benchmarks as representative of *core* decompression work achieved in end-to-end situations. What is further striking is that SCAN and SUM on ALP is faster than on uncompressed data, and the fact that ALP extends its performance lead over the competitors in the end-to-end benchmarks, compared to the micro-benchmarks. Note, however, that the micro-benchmark results were aggregated for all datasets (Table 5.3) so one should not directly compare with these tables.

Regarding multi-threading, the performance metrics in Table 5.3 and Figure 5.4 are *per-core*, hence equal performance would be perfect scaling. As all cores of the CPU get loaded, per-core ALP SCAN performance slightly drops – which also happens for uncompressed. This is caused by the query becoming RAM-bandwidth bound. However, in the SUM experiment, there is additional summing work (although not much) and therefore the query runs slower. As a result, ALP is able to scale perfectly while uncompressed is not.

Note that in Figure 5.4 the performance metric is reversed: lower is better. We present the summing work in the SUM query (=SUM-SCAN, because SUM also scans) as the lower part of the stacked bar: it is roughly 3 cycles per tuple. Figure 5.4 confirms our results across the board: ALP is much faster end-to-end than the other compressors, even faster than uncompressed, and scales well.

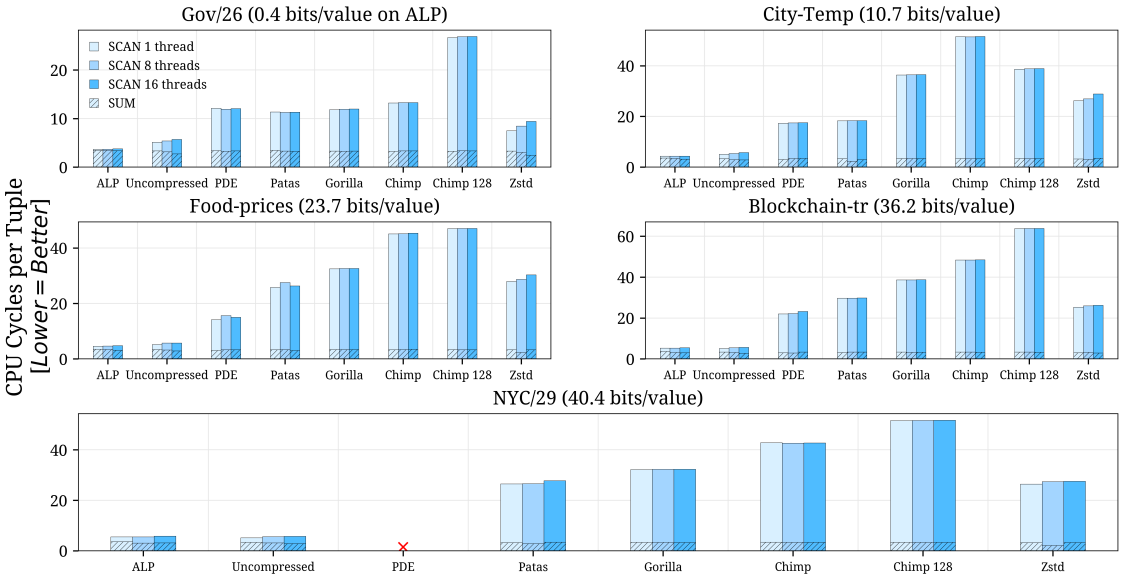

Figure 5.4: End-to-end SUM query execution speed for 5 datasets in Tectorwise (Ice Lake) measured in CPU cycles per Tuple. ALP is faster than all other schemes (even faster than *uncompressed*), while achieving perfect scaling (=speed stays the same) when using multi-core. Results show that SCAN is virtually free if data is compressed with ALP. PDE can't compress NYC/29.

5.3.2 Compression

ALP again is the fastest when compressing (Table 5.5): it is x4 and x7 times faster than the second and third-best algorithms in the City-Temp dataset (i.e. Patas, Gorilla) while still maintaining distance from Zstd (x11 slower) and PDE (x138 slower). `COMP` end-to-end performance is lower than in our micro-benchmarks. We attribute this to the extra effort in storing meta-data, the variable amount of exceptions (which are rather costly at compression time) and the first sampling phase which was not present in the micro-benchmarks.

Table 5.6: Machine Learning models presentation and detailed metrics computed on their weights. W2V Tweets model was a model trained by us using Python Gensim and 50K tokenized tweets.

			Decimal Precision		Values per Vector			Decimal Precision per Vector				IEEE 754 exponent per vector			
Name	**Model Type**	**N° of Parameters**	**Max**	**Min**	**Non-Unique%**	**Avg.**	**Std. Dev.**	**Max**	**Min**	**Avg.**	**Std. Dev.**	**Max**	**Min**	**Avg.**	**Std. Dev.**
Dino-Vitb16 (74)	Vision Transformer	86,389,248	20	13	0.000	0.005	0.311	19.5	13.5	16.7	0.7	1023.0	1011.0	1019.8	1.7
GPT2 (75)	Text Generation	124,439,808	20	0	0.001	-0.001	0.016	20.0	15.0	18.1	0.7	1018.2	1004.9	1015.0	2.0
Grammarly-lg (76)	Text2Text Generation	783,092,736	20	2	0.001	0.000	0.132	19.8	13.9	17.1	0.7	1021.6	1008.9	1018.5	1.7
W2V Tweets	Word2Vec	3,000	20	0	0.006	-0.009	0.728	19.5	13.4	16.7	0.7	1022.7	1010.2	1019.8	1.6
		AVG.	**20**	**3.75**	**0.002**	**-0.001**	**0.297**	**19.7**	**13.9**	**17.1**	**0.7**	**1021.4**	**1008.7**	**1018.3**	**1.7**

Table 5.7: Compression ratios (bits/value) that $ALP_{rd}32$ and its competitors achieved on machine learning models' weights (32-bits floats). $ALP_{rd}32$ achieved the best compression ratio.

Name	Model Type	N° of Params.	Gor.	Ch.	Ch. 128	Patas	ALP$_{rd}$	Zstd
Dino-Vitb16 (74)	Vision Transformer	86,389,248	34.1	33.4	33.4	45.8	28.3	29.7
GPT2 (75)	Text Generation	124,439,808	34.1	33.5	33.5	45.6	27.7	29.7
Grammarly-lg (76)	Text2Text	783,092,736	34.1	33.4	33.4	45.5	27.7	29.6
W2V Tweets	Word2Vec	3,000	34.1	33.3	33.3	45.5	28.8	29.8
		AVG.	34.1	33.4	33.4	45.6	28.1	29.7

5.4 Single Precision and Machine Learning Data

We have also ported *ALP* to 32-bits. Those of our double datasets with decimal precision ≤10 can be properly represented as 32-bit floating point numbers (all except Medicare/1, POI-lat, POI-lon, Basel-temp, Basel-wind and NYC/29); and 32-bit ALP works on them. This leads to the same compressed representation as in 64-bits (Table 5.2); but given that the uncompressed width is 32-bits, the average compression ratio is halved (and becomes ≈1.77).

A currently relevant different kind of 32-bit floats are found in trained machine learning models (i.e., the weights). However, these were created out of many multiplications and additions, and hence tend to have high precision. Still, there are commonalities in their sign and exponent parts (IEEE 754) that ALP_{rd} could take advantage of. Table 5.6 presents 4 different types of machine learning models and some metrics computer on their weights. As we mentioned, the IEEE 754 exponent deviation is small (with an average of ≈1.74), even on these datasets. The percentage of repeated values is ≈0% in all datasets and the average and maximum decimal precision is high.

Motivated by this, we ported ALP_{rd} to 32-bits and benchmarked it on four different ML models, against those competing schemes that have a version for 32-bit floats (i.e. Gorilla, Chimp, Chimp128, Gorilla) as well as Zstd. The results of this experiment are in Table 5.7; with ALP_{rd} for 32-bit floats achieving the best compression ratios out of all the other algorithms (28.1 bits/value; ≈12% of reduction). In fact, it is the only floating-point encoding to achieve compression. Alternatively, model weights are usually quantised (i.e. lossy reduction of precision) when deployed for inference (77). However, if this is not desired or possible; ALP_{rd} thus can provide some useful lossless compression for ML.

6

Discussion

A striking feat of our study of datasets used for database compression of doubles is that out of the 30 datasets our community uses for evaluating double compression, only the two POI datasets would not better be represented as fixed-point decimals. In fact, most POI data comes from GPS, which has an accuracy of a few meters, and the Earth's diameter is ≈12.750.000 meters (i.e., 8 digits, which corresponds to 28 bits). Indeed, when the POI-lat and POI-lon values are converted back from radians by multiplying with $\pi/180$ we observe this precision in the data – but we think it would go too far to define a specific ALP mode that deals with *pi*-multiplied data.

One may question why none of the datasets requires true double precision, nor is any all over the place in terms of magnitude – doubles allow numbers as close to zero as 10^{-308} and as large as 10^{308}. One interpretation could be that double is a catch-all type for two use cases: storing measures for which a-priori little is known about their domain (min/max), or where the magnitude is truly wide and/or variable. In the former use case, the actual data will tend to have min/max locality, leading to low variance in the high bits (equal or close exponent and highest mantissa bits). As the actual precision of actual values is limited by the measurement method, one either sees "pseudo-decimals" where the lower digits (in 10-base) are zero, or in the worst case, randomly filled in. The latter use case, high magnitude variance, seems to be rare, though weights and activations in machine learning could be the best example of this (not regarding large numbers, but numbers close to zero, i.e., highly variable negative exponents). Such data demonstrated to be hard to compress, for any scheme; and reducing their size is so crucial that it triggered the appearance of TensorFloat (Google) and Bfloat16 (Nvidia [1]). These new thin floats,

[1] https://en.wikipedia.org/wiki/Bfloat16_floating-point_format

developed with Machine Learning hardware in mind, mostly cut down on mantissa and somewhat on exponent.

The use of doubles in scientific calculations is common; though researchers have criticized the rounding errors produced (71), and proposed alternatives like unum and posit(78). There are strong arguments for compressing doubles stored in big data formats and database files: data gets smaller, reducing storage cost across the memory hierarchy, reducing also I/O time, network transfer time and usage. We think that with the increased convergence of data science and scientific computations and machine learning data with database workloads, there will be growing demand for doubles in databases, and their compressed storage.

7

Conclusions

We have presented and evaluated ALP: a strongly enhanced version of Decimal-based encoding with an adaptive fallback to front-bit compression if doubles have truly large precision. ALP beats the competition in all relevant dimensions. Its compression ratio is better than all recently proposed floating-point encodings, while being *much* faster in [de]compression speed. Its compression ratio is only equalled by heavy-weight general-purpose compression; but these methods have slow [de]compression speeds and are block-based: forcing database scan to fully decompress a large block of data. In contrast, one can skip through ALP-compressed data at the vector-level; allowing for efficient predicate push-down. We think ALP will be a valuable encoding in *cascading* lightweight compression formats (1, 3), and recall that in our evaluation it already beat zstd (18.8 vs. 20.6) when cascading on Dictionary and RLE. The *robustness* of ALP speed was tested on 30 real datasets that were also used to evaluate previous schemes and further proven in five different modern computer architectures (i.e., Intel Ice Lake, AMD Zen 3, Apple M1, and AWS Graviton 2 & 3) and using Auto-vectorized, Scalar and SIMDized code.

We would like to stress that the key idea behind ALP is to design for *vectorized execution*; it led us to analyze and uncover unexploited opportunities from a vector perspective in a variety of datasets. Vectorized execution reduces computational cost (reducing loop-, function call-, and load/store-overhead), brings out the best in compilers (vectorized code triggers loop-centered optimizations including auto-vectorization), but also amortizes storage (parameters such as exponent are stored once per-vector instead of per-value), allows for per-vector adaptivity without reducing performance due to branch-mispredictions (as happens in per-value adaptivity in e.g., the Chimp variants), and can take advantage of in-vector data commonalities. As for future work, we think that the implementation of ALP on massively parallel hardware such as GPUs and TPUs could be fruitful.

References

[1] Kuschewski Maximilian, Sauerwein David, Alhomssi Adnan, and Leis Viktor. **BtrBlocks: Efficient Columnar Compression for Data Lakes**. Association for Computing Machinery, 2023. In press. Accessed on: 2023-04-13. ii, 1, 3, 44, 45, 47, 48, 53, 61, 81, 86, 103

[2] **FastLanes Library (Github)**, 2023. Accesed on: 2023-04-13. ii, 81

[3] Azim Afroozeh and Peter Boncz. **The FastLanes Compression Layout: Decoding >100 Billion Integers per Second with Scalar Code**, April 2023. Technical Report. Accessed on: 2023-04-13. ix, 3, 8, 9, 11, 12, 13, 14, 15, 45, 47, 48, 67, 77, 81, 103

[4] Peter Boncz, Thomas Neumann, and Viktor Leis. **FSST: fast random access string compression**. *Proceedings of the VLDB Endowment*, **13**(12):2649–2661, 2020. ix, 16, 17

[5] Vadim Engelson, Peter Fritzson, and Dag Fritzson. **Lossless compression of high-volume numerical data from simulations**, 2000. ix, 19, 20, 21

[6] Paruj Ratanaworabhan, Jian Ke, and Martin Burtscher. **Fast lossless compression of scientific floating-point data**. In *Data Compression Conference (DCC'06)*, pages 133–142. IEEE, 2006. ix, 21, 23

[7] Ugur Cayoglu, Frank Tristram, Jörg Meyer, Jennifer Schröter, Tobias Kerzenmacher, Peter Braesicke, and Achim Streit. **Data Encoding in Lossless Prediction-Based Compression Algorithms**. In *2019 15th International Conference on eScience (eScience)*, pages 226–234. IEEE, 2019. ix, 23, 24

[8] Martin Burtscher and Paruj Ratanaworabhan. **FPC: A high-speed compressor for double-precision floating-point data**. *IEEE transactions on computers*, **58**(1):18–31, 2008. ix, 5, 6, 24, 25

[9] Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. **Chimp: efficient lossless floating point compression for time series databases**. *Proceedings of the VLDB Endowment*, **15**(11):3058–3070, 2022. x, 1, 5, 29, 32, 33, 34, 35, 37, 38, 44, 61, 65, 86

[10] Zhiqi Wang, Jin Xue, and Zili Shao. **Heracles: an efficient storage model and data flushing for performance monitoring timeseries**. *Proceedings of the VLDB Endowment*, **14**(6):1080–1092, 2021. x, 30

[11] Andrea Bruno, Franco Maria Nardini, Giulio Ermanno Pibiri, Roberto Trani, and Rossano Venturini. **TSXor: A Simple Time Series Compression Algorithm**. In *String Processing and Information Retrieval: 28th International Symposium, SPIRE 2021, Lille, France, October 4–6, 2021, Proceedings 28*, pages 217–223. Springer, 2021. x, 30, 32

[12] Ruiyuan Li, Zheng Li, Yi Wu, Chao Chen, and Yu Zheng. **Elf: Erasing-based Lossless Floating-Point Compression**. *Proceedings of the VLDB Endowment*, **16**(7), 2023. x, 1, 2, 6, 39, 40, 61, 86

[13] Chunwei Liu, Hao Jiang, John Paparrizos, and Aaron J Elmore. **Decomposed bounded floats for fast compression and queries**. *Proceedings of the VLDB Endowment*, **14**(11):2586–2598, 2021. x, 6, 42, 43, 70

[14] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and Marios Skounakis. **Weaving Relations for Cache Performance.** In *VLDB*, **1**, pages 169–180, 2001. x, 49, 50

[15] Yin Huai, Ashutosh Chauhan, Alan Gates, Gunther Hagleitner, Eric N Hanson, Owen O'Malley, Jitendra Pandey, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. **Major technical advancements in apache hive**. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1235–1246, 2014. xi, 52, 53

[16] Bogdan Ghita, Diego G Tomé, and Peter A Boncz. **White-box Compression: Learning and Exploiting Compact Table Representations.** In *CIDR*, **1**, page 27, 2020. xi, 56, 57

[17] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. **Super-scalar RAM-CPU cache compression**. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 59–59. IEEE, 2006. 1, 9, 11, 12, 14, 85

[18] Daniel Abadi, Samuel Madden, and Miguel Ferreira. **Integrating compression and execution in column-oriented database systems**. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 671–682, 2006. 1, 19

[19] Deepak Vohra. *Apache Parquet*, pages 325–335. 09 2016. 1

[20] Mark Raasveldt and Hannes Mühleisen. **Duckdb: an embeddable analytical database**. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1981–1984, 2019. 1, 52, 54, 59, 86

[21] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. **Velox: meta's unified execution engine**. *Proceedings of the VLDB Endowment*, **15**(12):3372–3384, 2022. 1

[22] Biswapesh Chattopadhyay, Priyam Dutta, Weiran Liu, Ott Tinn, Andrew Mccormick, Aniket Mokashi, Paul Harvey, Hector Gonzalez, David Lomax, Sagar Mittal, et al. **Procella: Unifying serving and analytical data at YouTube**. 2019. 1

[23] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. **Compressing relations and indexes**. In *Proceedings 14th International Conference on Data Engineering*, pages 370–379. IEEE, 1998. 1, 12

[24] Vijayshankar Raman and Garret Swart. **How to wring a table dry: Entropy compression of relations and querying of compressed relations**. In *Proceedings of the 32nd international conference on Very large data bases*, pages 858–869. Citeseer, 2006. 1, 13

[25] Mark A Roth and Scott J Van Horn. **Database compression**. *ACM Sigmod Record*, **22**(3):31–39, 1993. 1

[26] **IEEE Standard for Floating-Point Arithmetic**. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019. 1, 18

[27] Yann Collet. **Zstandard - Fast real-time compression algorithm**, 2015. Accesed on: 2023-04-13. 1, 86, 87

[28] Yann Collet. **LZ4 - Extremely fast compression**, 2014. Accesed on: 2023-04-13. 1

[29] Seungyeon Lee, Jusuk Lee, Yongmin Kim, Kicheol Park, Jiman Hong, and Junyoung Heo. **Efficient scheme for compressing and transferring data in hadoop clusters**. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 1256–1263, 2020. 1

[30] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. **Gorilla: A fast, scalable, in-memory time series database**. *Proceedings of the VLDB Endowment*, **8**(12):1816–1827, 2015. 1, 27, 86

[31] DuckDB Labs. **Patas Compression: Variation on Chimp**. `https://github.com/duckdb/duckdb/pull/5044`, 2022. Accessed on: 2023-04-13. 1, 2, 37, 86

[32] Boudewijn Braams. **Predicate Pushdown in Parquet and Apache Spark**. *MSc thesis*, 2018. 2

[33] Peter A Boncz, Marcin Zukowski, and Niels Nes. **MonetDB/X100: Hyper-Pipelining Query Execution.** In *Cidr*, **5**, pages 225–237, 2005. 2, 86

[34] Daniel Lemire and Leonid Boytsov. **Decoding billions of integers per second through vectorization**. *Software: Practice and Experience*, **45**(1):1–29, 2015. 2, 9, 11, 12, 13, 45, 85

[35] Johannes Pietrzyk, Annett Ungethüm, Dirk Habich, and Wolfgang Lehner. **Beyond Straightforward Vectorization of Lightweight Data Compression Algorithms for Larger Vector Sizes.** In *Grundlagen von Datenbanken*, pages 71–76, 2018. 2

[36] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. **Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses).** In *EDBT*, pages 72–83, 2017. 3, 7, 9, 45, 47, 94

[37] Jacob Ziv and Abraham Lempel. **A universal algorithm for sequential data compression**. *IEEE Transactions on information theory*, **23**(3):337–343, 1977. 4

[38] Daniel Lemire, Nathan Kurz, and Christoph Rupp. **Stream VByte: Faster byte-oriented integer compression**. *Information Processing Letters*, **130**:1–6, 2018. 5, 6, 9, 11

[39] Hugh E Williams and Justin Zobel. **Compressing integers for fast file access**. *The Computer Journal*, **42**(3):193–201, 1999. 9, 10

[40] Jeff Plaisance, Nathan Kurz, and Daniel Lemire. **Vectorized vbyte decoding**. *arXiv preprint arXiv:1503.07387*, 2015. 11

[41] Azim Afroozeh and P Boncz. **Towards a New File Format for Big Data: SIMD-Friendly Composable Compression**, 2020. 11, 85

[42] Haoyu Wang and Shaoxu Song. **Frequency domain data encoding in apache IoTDB**. *Proceedings of the VLDB Endowment*, **16**(2):282–290, 2022. 11

[43] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. **SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units**. *Proceedings of the VLDB Endowment*, **2**(1):385–394, 2009. 11

[44] Vo Ngoc Anh and Alistair Moffat. **Index compression using 64-bit words**. *Software: Practice and Experience*, **40**(2):131–147, 2010. 13

[45] Bhavik Nagda. *CHuff: Conditional Huffman String Compression*. PhD thesis, Massachusetts Institute of Technology, 2021. 17

[46] Peter Lindstrom and Martin Isenburg. **Fast and efficient compression of floating-point data**. *IEEE transactions on visualization and computer graphics*, **12**(5):1245–1250, 2006. 19, 23

[47] Nathaniel Fout and Kwan-Liu Ma. **An adaptive prediction-based approach to lossless compression of floating-point volume data**. *IEEE Transactions on Visualization and Computer Graphics*, **18**(12):2295–2304, 2012. 19

[48] Martin Isenburg, Peter Lindstrom, and Jack Snoeyink. **Lossless compression of predicted floating-point geometry**. *Computer-Aided Design*, **37**(8):869–877, 2005. 19

[49] Lawrence Ibarria, Peter Lindstrom, Jarek Rossignac, and Andrzej Szymczak. **Out-of-core compression and decompression of large n-dimensional scalar fields**. In *Computer Graphics Forum*, **22**, pages 343–348. Wiley Online Library, 2003. 19, 23

[50] Agner Fog et al. **Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs**. *Copenhagen University College of Engineering*, **93**:110, 2011. 21, 45, 73

[51] Bart Goeman, Hans Vandierendonck, and Koenraad De Bosschere. **Differential FCM: Increasing value prediction accuracy by improving table usage efficiency**. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 207–216. IEEE, 2001. 22, 24

[52] Yiannakis Sazeides and James E Smith. **The predictability of data values**. In *Proceedings of 30th Annual International Symposium on Microarchitecture*, pages 248–258. IEEE, 1997. 24

[53] Davis Blalock, Samuel Madden, and John Guttag. **Sprintz: Time series compression for the internet of things**. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, **2**(3):1–23, 2018. 26

[54] Aliaksandr Valialkin. **VictoriaMetrics: achieving better compression than Gorilla for time series data**. `https://faun.pub/victoriametrics-achieving-better-compression-for-time-series-data-than-gorilla-317bc` 2019. Accesed on: 2023-04-13. 44

[55] **Public BI Benchmark**. `https://github.com/cwida/public_bi_benchmark`, 2019. Accessed on: 2023-04-13. 45, 61

[56] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. **Get real: How benchmarks fail to represent the real world**. In *Proceedings of the Workshop on Testing Database Systems*, pages 1–6, 2018. 45, 61

[57] Steven Claggett, Sahar Azimi, and Martin Burtscher. **SPDP: An automatically synthesized lossless compression algorithm for floating-point data**. In *2018 Data Compression Conference*, pages 335–344. IEEE, 2018. 46

[58] Fabian Knorr, Peter Thoman, and Thomas Fahringer. **ndzip: A high-throughput parallel lossless compressor for scientific data**. In *2021 Data Compression Conference (DCC)*, pages 103–112. IEEE, 2021. 46

[59] George P Copeland and Setrag N Khoshafian. **A decomposition storage model**. *Acm Sigmod Record*, **14**(4):268–279, 1985. 49

[60] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. **Dremel: interactive analysis of web-scale datasets**. *Proceedings of the VLDB Endowment*, **3**(1-2):330–339, 2010. 52

[61] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. **RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems**. In *2011 IEEE 27th International Conference on Data Engineering*, pages 1199–1208. IEEE, 2011. 52

[62] Mark Raasveldt. **Lightweight Compression in DuckDB**. `https://duckdb.org/2022/10/28/lightweight-compression.html`, 2022. Accesed on: 2023-04-13. 54

[63] Hao Jiang, Chunwei Liu, John Paparrizos, Andrew A Chien, Jihong Ma, and Aaron J Elmore. **Good to the last bit: Data-driven encoding with codecdb**. In *Proceedings of the 2021 International Conference on Management of Data*, pages 843–856, 2021. 56

[64] Hao Jiang, Chunwei Liu, Qi Jin, John Paparrizos, and Aaron J Elmore. **Pids: attribute decomposition for improved compression and query performance in columnar storage**. *Proceedings of the VLDB Endowment*, **13**(6):925–938, 2020. 57

[65] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A Boncz, Thomas Neumann, and Alfons Kemper. **Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation**. In *Proceedings of the 2016 International Conference on Management of Data*, pages 311–326, 2016. 59

[66] National Ecological Observatory Network (NEON). **Barometric pressure (DP1.00004.001)**, 2021. 60

[67] National Ecological Observatory Network (NEON). **Relative humidity above water on-buoy (DP1.20271.001)**, 2021. 60

[68] National Ecological Observatory Network (NEON). **IR biological temperature (DP1.00005.001)**, 2021. 60

[69] National Ecological Observatory Network (NEON). **Dust and particulate size distribution (DP1.00017.001)**, 2021. 60

[70] National Ecological Observatory Network (NEON). **2D wind speed and direction (DP1.00001.001)**, 2021. 60

[71] David Goldberg. **What every computer scientist should know about floating-point arithmetic**. *ACM computing surveys (CSUR)*, **23**(1):5–48, 1991. 71, 102

[72] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. **Everything you always wanted to know about compiled and vectorized queries but were afraid to ask**. *Proceedings of the VLDB Endowment*, **11**(13):2209–2222, 2018. 86, 96

[73] Mark Raasveldt and Hannes Muehleisen. **DuckDB**, 2019. Accesed on: 2023-04-13. 86

[74] Mathilde Caron, Hugo Touvron, Ishan Misra, Hervé Jégou, Julien Mairal, Piotr Bojanowski, and Armand Joulin. **Emerging Properties in Self-Supervised Vision Transformers**. *CoRR*, **abs/2104.14294**, 2021. 99, 100

[75] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. **Language Models are Unsupervised Multitask Learners**. 2019. 99, 100

[76] Vipul Raheja, Dhruv Kumar, Ryan Koo, and Dongyeop Kang. **CoEdIT: Text Editing by Task-Specific Instruction Tuning**. 2023. 99, 100

[77] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E Gonzalez, et al. **High-throughput generative inference of large language models with a single gpu**. *arXiv preprint arXiv:2303.06865*, 2023. 100

[78] John L Gustafson and Isaac T Yonemoto. **Beating floating point at its own game: Posit arithmetic**. *Supercomputing frontiers and innovations*, **4**(2):71–86, 2017. 102