ALP: Adaptive Lossless Floating-Point Compression

Azim Afroozeh
CWI
Amsterdam, The Netherlands

Leonardo Kuffó
CWI
Amsterdam, The Netherlands

Peter Boncz
CWI
Amsterdam, The Netherlands

ABSTRACT

IEEE 754 doubles do not exactly represent most real values, introducing rounding errors in computations and [de]serialization 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, 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 (Figure 1). 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 [31] 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 our FastLanes library [6], and an efficient two-stage compression algorithm that first samples row-groups and then vectors.

KEYWORDS

lossless compression, floating point compression, lightweight compression, vectorized execution, columnar storage, big data formats

ACM Reference Format:


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 [4, 12, 37, 41, 50, 51], 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) [20, 44, 46]. 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 [1], 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 [13, 14, 26]. 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 [38], Chimp and Chimp128 [29], PseudoDecimals (PDE) [31], Patas [24] and Elf [28]. 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 [24], is a version of Chimp128 that stores non-zero byte-sequences rather than bit-sequences. Whereas...
Patas trades compression ratio for faster decompression, Elf [28] does the opposite: it uses a mathematical formula to zero more XOR bits and improve the compression ratio, at the cost of lower \( [\text{de}] \) 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 [8], their \([\text{de}]\) compression speed (as well as compression ratio) is not much higher than that of general-purpose schemes [28]; in other words, these encodings are not quite lightweight.

We introduce ALP, a lightweight floating-point encoding that is vectorized [7]: 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 [27, 39]. 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 \([\text{de}]\) compression itself has very few data- or control dependencies, leading to higher speeds.

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
- an evaluation 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 1).

2 DATASETS ANALYSIS

Compression methods achieve their best performance when they are capable of exploiting properties of the data. However, the same methods could fail to achieve any compression if the data lacks these exploitable properties. In this section we analyze a number of floating-point datasets, aiming to uncover properties relevant to compression performance. 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 [25, 41]: they deliver vector-sized chunks of data, and use decompression kernels that decompress one vector (e.g., 1024 values) at-a-time.

We start by explaining in detail the IEEE 754 doubles representation in subsection 2.1. Then, we introduce the analyzed datasets in subsections 2.2 and 2.3. Next, in subsection 2.4 we analyze the data similarities at the vector level. In subsection 2.5 we revisit decimal-based encoding approaches and perform further analysis of these methods from a vectorized point of view. Finally, in subsection 2.6 we elaborate on the compression opportunities we found.

2.1 IEEE 754 Doubles Representation

IEEE 754 [1] represents 64-bit doubles in 3 segments of bits (Figure 2): 1 bit for sign (0 for positive, 1 for negative), 11 bits for an exponent (represents an unsigned integer from 0 to 2047) and 52 bits for the fraction (represents a summation of inverse powers of two; also known as mantissa or significand) – which together represent a real number defined as: \((-1)^{\text{sign}} \times 2^{\text{exponent}} \times \left(1 + \sum_{i=1}^{52} b_{52-i} \times 2^{-i}\right)\). This definition allows for up to 17 significant decimal places of precision. However, it introduces errors in arithmetic (e.g., addition, multiplication) and limitations on the integer part of numbers which we will discuss later on in this section. The same standard also defines 32-bit floats (8 bits for exponent and 23 for mantissa).

![Figure 2: IEEE 754 doubles bitwise representation.](image)

2.2 Datasets

Table 1 presents an overview of the 30 datasets that we analyzed in detail in order to design ALP: 18 of these datasets were previously analyzed and evaluated to develop Elf [28] and Chimp [29], the other 12 were used to evaluate PDE [31]. We consider these 30 datasets to be relevant because they capture a variety of distributions, and because they played a role in the analysis, design and evaluation of competing floating-point encodings. Identifying new properties, we gained important clues guiding the design of ALP. Finally, by using these datasets we are able to perform a fair comparison between these methods and our new ALP compression.

2.3 Dataset Semantics

The first 13 datasets presented in Table 1 contain time series data. On these datasets, each double value \(v_i\) is recorded further in time than value \(v_{i+1}\). The next 17 datasets are more representative of doubles stored in classical database workloads: 12 of these non-time series datasets are part of the Public BI Benchmark [2] a collection of the biggest Tableau Public workbooks [49]. Note that all datasets are user-contributed data (non-synthetic).

The datasets have significant variety in their semantics. As presented in Table 1, 14 datasets contain doubles that represent monetary values (i.e., Exchange rates, public funds, product prices, stocks...
and trailing-zero bits [10, 38, 45]. However, in Table 2:C14 and C15, we see that the average number of leading and trailing zeros bits after XORing is comparable 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 also reflected by the fact that Chimp and Chimp128 do really well on this data [29]. Regardless of semantics, leading and trailing zero bits go down with lower percentages of duplicates (Table 2:C6 non-unique values) and higher decimal precision (Table 2:C2). For instance, in both datasets in which decimal precision reaches 20 digits (i.e., POI-lat and POI-lon), the leading and trailing 0-bit average of XORed values is the lowest.

Human-readable similarity. From a human perspective, two doubles are similar if their orders of magnitude (exponent) and their visible decimal precision are similar. On our time series datasets, the standard deviation of the magnitudes (Table 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), though never extremely high when compared to the average magnitude (Table 2:C7).

Decimal precision varies between datasets (Table 2:C2 and C3). For instance, datasets that contain geographic 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 values with just 1 decimal of precision. Despite these differences inside a dataset, the deviation of this property is usually small from a vector perspective (Table 2:C5). In fact, for 25 out of 30 datasets, the decimal precision deviation inside vectors is smaller than 1. That means that most of the values inside a vector share the same decimal precision.

Decimal-based encoding approaches such as PDE exploit these human-readable similarities of doubles by trying to represent them as integers [31]. The more similar the decimal precision and the orders of magnitude of doubles inside a block of values, the better compression ratio can be achieved.

### 2.4 Data Similarity

The underlying temporal property of time series data has been shown to result in similar values stored close-by [29, 38]. We can analyze similarity of doubles from two different points of view: (i) their bitwise representation (IEEE 754 [1]) and (ii) their human-readable representation.

#### Bitwise similarity.

From a bitwise point of view, two double floating-point values are considered similar if their sign, exponent and fraction parts are similar. Table 2:C9 and C10 show the double exponent average and deviation per vector. We define a vector as 1204 consecutive values [7]. In most of the datasets, the exponent deviation is small, particularly in time series data. These small deviations are reflected by the number of leading-0 bits resulting from XORing the doubles with their previous value. When similar doubles are XORed, the result typically has a high number of leading-zero bits [10, 38, 45]. However, in Table 2:C14 and C15, we see that the average number of leading and trailing zeros bits after XORing is comparable 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 also reflected by the fact that Chimp and Chimp128 do really well on this data [29]. Regardless of semantics, leading and trailing zero bits go down with lower percentages of duplicates (Table 2:C6 non-unique values) and higher decimal precision (Table 2:C2). For instance, in both datasets in which decimal precision reaches 20 digits (i.e., POI-lat and POI-lon), the leading and trailing 0-bit average of XORed values is the lowest.

#### Human-readable similarity.

From a human perspective, two doubles are similar if their orders of magnitude (exponent) and their visible decimal precision are similar. On our time series datasets, the standard deviation of the magnitudes (Table 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), though never extremely high when compared to the average magnitude (Table 2:C7).

Decimal precision varies between datasets (Table 2:C2 and C3). For instance, datasets that contain geographic 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 values with just 1 decimal of precision. Despite these differences inside a dataset, the deviation of this property is usually small from a vector perspective (Table 2:C5). In fact, for 25 out of 30 datasets, the decimal precision deviation inside vectors is smaller than 1. That means that most of the values inside a vector share the same decimal precision.

Decimal-based encoding approaches such as PDE exploit these human-readable similarities of doubles by trying to represent them as integers [31]. The more similar the decimal precision and the orders of magnitude of doubles inside a block of values, the better compression ratio can be achieved.

### 2.5 Representing Doubles as Integers

Representing double-precision floating-point values as integers is non-trivial. Take for instance the number \( n = 8.0605 \). At first glance, to encode \( n \) as an integer we could be tempted to move the decimal point \( e \) spaces to the right until there are no decimals left (i.e., \( e \) spaces). The latter can be achieved with the following procedure: \( P_{en} = \text{round}(n \times 10^e) \). Since one of the multiplication operands of \( P_{en} \) 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 \( d = 80605 \) (i.e., the result of \( P_{en} \)) and another 32-bit integer representing the number of spaces \( e \) we moved the decimal point (i.e., a factor of 10). Hence, from the encoded integer \( d \) result of \( P_{en} \), and the number of spaces \( e \) we moved the decimal point, we should be able to recover the original double by performing the following procedure: \( P_{dec} = d \times 10^{-e} \).

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 [19]. The real representation of
the number 8.0605 as a double based on the IEEE 754 definition is: 8.06009999999993209. To achieve lossless compression, this has to be the exact result of our procedure \( P_{dec} \). However, in our example \( P_{dec} \) yields 8.0605000000000001084. 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 inexact decimal representation either. Hence, \( 10^{-14} \) is not 0.0001 but something like 0.0000000000000000000082. 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 \( 10^8 \) has an exact double representation for \( e \leq 21 \).

Table 2:C11 depicts the percentage of doubles in each dataset that can be losslessly represented by an integer \( d \) and an exponent \( e \) using the \( P_{enc} \) and \( P_{dec} \) procedures. But, always using the visible precision of the doubles as the exponent \( e \) (e.g., for \( 0.001 \)), 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 \( e \) and (ii) the visible precision of the double \( n \).

### High exponents work for all values

Table 2:C12 shows the exponent \( e \) which leads to the highest success-rate of \( P_{enc} \) and \( P_{dec} \) on each dataset. It is evident that higher exponents such as 14 and 16 are predominant, with an average of 95% successfully encoded values in all 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 \( e \) the closer we can get to obtaining the real double with the procedures. This is due to higher exponents \( e \) resulting in a more precise inverse factor of 10 on \( P_{dec} \). For instance, \( 10^{-14} \) represented as a double is equal to 1.0000000000000000000771E-15. As a consequence, the result of \( P_{dec} \) 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 2:C13, when optimizing to use a different exponent \( e \) 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 \( e \) for each value is needed – which is what PDE does.

However, by using higher exponents \( e \) the integers resulting from the procedure \( P_{enc} \) become big (i.e., 64-bits). These high exponents that lead to big integers are not used by PDE since they lead to a worse compression ratio than leaving the data uncompressed (because storing a 64-bit integer plus an exponent takes more space than a 64-bit double). Note that the doubles in datasets such as NYC/29, POI-lat and POI-lon are only representable as big integers.
The 52-bit limit for integers. Exponent $e = 14$ is the most successful in most of the datasets to represent doubles as integers using $P_{\text{enc}}$ and $P_{\text{dec}}$. 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 $P_{\text{dec}}$ result. However, there are two datasets in which even higher exponents $e$ are needed (i.e., POI-lat, POI-lon) because the visible precision of the double values inside those datasets on average exceeds 14 (Table 2:C4). As we explain subsequently, when the order of magnitude of a double $n$ plus its visible decimal precision reaches 16, $P_{\text{enc}}$ is prone to fail due to a limitation of the IEEE 754 doubles.

The multiplication inside $P_{\text{enc}}$ yields a double due to having a double operand. Hence, before rounding, our resulting integer $d$ 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 $2^{54}$ can be exactly represented in the integer part of a double number. Going beyond this threshold is problematic. Between $2^{53}$ and $2^{54}$, 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 $P_{\text{enc}}$ 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 $P_{\text{enc}}$ and $P_{\text{dec}}$. This is why POI-lat and POI-lon achieve a relatively low successful encoding rate of 76.4% and 70.5% respectively. Also, this is why we stated earlier that $10^{e}$ only has an exact double representation for $e \leq 21$.

2.6 Unexploited Opportunities

All recently proposed competing floating-point encoding already exploit some of the properties discussed in the previous subsections. However, there is room for substantial improvement both in terms of compression ratio and [de]compression speed.

Vectorizing Decimal Encoding. In subsection 2.5 we demonstrated that it is possible to achieve near 100% success rate of our procedures $P_{\text{enc}}$ and $P_{\text{dec}}$ by using only one exponent $e$ for every vector. The current state-of-the-art Decimal-based approach PDE [31] embeds the exponent $e$ in every value. Hence, by exploiting this opportunity, compression ratio could be improved.

Cutting trailing 0s with an extra multiplication. In subsection 2.5 we demonstrated that high exponents $e$ achieve the highest success rate on our procedures $P_{\text{enc}}$ and $P_{\text{dec}}$ 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 $e$ per vector opens the opportunity to encode big integers without instantly falling behind in compression ratio against uncompressed values.

High exponents $e$ in combination with low-precision decimals datasets (e.g., SD-bench, City-Temp, Stocks-UK) result in 64-bit integers that contain tails of repeated trailing 0-digits (e.g., $n = 37.3$ and $e = 14$, yields $P_{\text{enc}} = 3730000000000000$; $n = 100.8333$ and $e = 14$, yields $P_{\text{enc}} = 10083330000000000$). These tails of repeated 0-digits will have the same length in datasets with low magnitude variance and low decimal precision variance (e.g., SD-bench, City-Temp, PM10-Dust). Cutting these tails with an extra multiplication with an inverse factor of $10$, namely $f$, results in a smaller integer that can be used to recover the 64-bit integer with the inverse operation (i.e., a multiplication with a factor $f$ of 10). Hence, we can redefine $P_{\text{enc}}$ and $P_{\text{dec}}$ as follows:

\[
ALP_{\text{enc}} = \text{round}(n \times 10^{e} \times 10^{-f})
\]

\[
ALP_{\text{dec}} = d \times 10^{f} \times 10^{-e}
\]

Based on the analysis done in subsection 2.5 one might fear that this new multiplication with another inverse factor of 10 in $ALP_{\text{enc}}$ could result in new rounding errors. However, the error introduced by these inverse factors of 10 turns out to pose no problems. To illustrate, with $n = 8.0605$, $e = 14$ and $f = 10$, $ALP_{\text{enc}}$ and $ALP_{\text{dec}}$ will execute as follows:

\[
ALP_{\text{enc}} = \text{round}(8.0604999999999933209 \times 10^{14} \times 10^{-10})
\]

\[
ALP_{\text{dec}} = \text{round}(8060499999999999.875 \times 10^{-10})
\]

\[
ALP_{\text{enc}} = \text{round}(80604.999999999985448)
\]

\[
ALP_{\text{dec}} = 80605 \times 10^{10} \times 10^{-14}
\]

\[
ALP_{\text{dec}} = n = 8.0604999999999933209
\]

In the third step of $ALP_{\text{enc}}$, the error introduced by $10^{-10}$ is negligible for the resulting integer $d$. Using this reducing factor $f$ in the procedures is a way of taking advantage of the high coverage and success rates or large exponents, without having to encode big integers $d$. Note that this example is the same $n$ we used at the beginning of subsection 2.5, which could not be encoded by simply using $e = 4$.

Limited Search Space. Until now, we have ignored the process of finding the exponent $e$ for our decimal-based encoding procedures $ALP_{\text{enc}}$ and $ALP_{\text{dec}}$. The current state-of-art on decimal-based encoding (i.e., PDE) performs a brute-force search for each value in a dataset in order to find the exponent $e$. For our $ALP$ procedures, an additional nested brute-force search needs to be performed in order to find the best combination of exponent $e$ and factor $f$. We define the best combination as the one in which $ALP_{\text{enc}}$ yields the smallest integer $d$ with which $ALP_{\text{dec}}$ succeeds in recovering the original double $n$. This translates into a search space of 253 possible exponent $e$ and factor $f$ combinations (given that $f \leq e$ and $0 \geq e \leq 21$). However, we have already discussed that most values inside a vector can be encoded by using one single exponent. Furthermore, we have also mentioned that vectors exhibit a low variance in their decimal precision and in their magnitudes. Hence, our intuition was that the search space for the combination of exponent $e$ and factor $f$ can be greatly reduced and that it should be done on a per-vector basis. In order to confirm this, we computed the best combination for each vector in each dataset. For this experiment, the search was performed on all the possible search space of 253 combinations for every vector. Figure 3 shows that for most datasets a search space of 5 combinations is enough to obtain the best combination among all vectors in the dataset. For some datasets such as Basel-wind, Bird-migration, City-Temp, Wind-dir and IR-bio-temp, the entire search space is just one combination.
3 ALP

ALP is an adaptive lossless encoding designed to compress double-precision floating-point data. ALP takes advantage of the opportunities discussed in subsection 2.6. Compression and decompression are built upon the ALPenc and ALPdec procedures described in section 2.6. 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 2.6. In both compression and decompression. In the following subsections, we describe the key design aspects of ALP and how it implements adaptivity.

3.1 Compression

ALP compression is built upon the ALPenc procedure (Formula 1). ALP tries to encode all doubles \( n \) inside a vector \( \mathbf{v} \) with the same exponent \( e \) and factor \( f \). Inside the encoding, ALP must verify that the procedures ALPenc and ALPdec yield the original double \( n \). If the original double \( n \) cannot be recovered, we treat the double as an exception. Algorithm 1 shows the pseudo-code for ALP encoding.

**Algorithm 1: ALP Compression**

```plaintext
double i.F10 = {1.0, 0.1, 0.01, 0.001, ...};
double F10 = {1.8, 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([i]);
for (i = 0; i < VECTOR_SIZE; ++i){ // Encode the vector
    double n = input_vec[i];
t64 d = fast_double_round(n * F10[e] * i_F10[f]); // ALPenc
    encoded_vec[i] = d;
    decoded_vec[i] = d * F10[f] * i_F10[e]; // ALPdec
}
int exc.count = 0;
for (i = 0; i < VECTOR_SIZE; ++i){ // Fetch Exceptions
    if (i = 0; i < VECTOR_SIZE; ++i){ // Predicted comparison
        exc_vec[exc.pos_vec[i]] = first_encoded;
        exc_vec[exc.pos_vec[i]] = input_vec[i];
    }
}
```

**3.2 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); but nevertheless, compression may still be worthwhile.

We believe that the approach of a decimal-based encoding is not appropriate for such compression-unfriendly data; and 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 2:C10, even on these datasets (i.e., POI-lat, POI-lon) we see that the exponent of the bit-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 our FastLanes compression library [6]. Furthermore, based on the analysis of leading 0-bits from XOR-ing with the previous value (Table 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).
Handling Exceptions. Values which fail to be encoded as decimals become exceptions. Exceptions are stored uncompressed in a separate segment (i.e., \texttt{exc_vec} in Algorithm 1). However, since our approach is vectorized, we cannot simply skip the exceptions in the resulting vector of encoded values (i.e., \texttt{encoded_vec} in Algorithm 1). Hence, when exceptions occur we store an auxiliary value in the \texttt{encoded_vec} (i.e., \texttt{first_encoded} in Algorithm 1 Line 20). This auxiliary value is the first successfully encoded \(d\) which is obtained by the \texttt{FIND\_FIRST\_ENCODED} function in Algorithm 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., \texttt{exc_pos_vec} in Algorithm 1). For \(a = 1024\), 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 1 show the exception handling process which is cleverly built to avoid control structures (i.e., \texttt{if-then-else}).

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 2.4, we decided to encode the yielded integers using a Fused variant of the Frame-Of-Reference encoding available in the FastLanes library called \texttt{FFOR}. FastLanes \cite{fastlanes} proposes a new data layout to accelerate the encoding and decoding of lightweight [de]compression methods with scalar code that auto-vectorizes. \texttt{FFOR} fuses the implementation of bit-[un]packing with the \texttt{FOR} encoding and decoding process into a single kernel that performs both processes. The \texttt{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).

However, there is some more headroom as a modern compression library (e.g., \cite{fastlanes, dictionary}) could try multiple different integers encodings and also \texttt{cascade} these. For instance, if the data is repetitive, one could use Dictionary coding, and compress the Dictionary with \texttt{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 \texttt{FFOR} to the Dictionary or the Run Values.

### 3.2 Adaptive Sampling

Our compression method does not perform a brute-force search for the exponent \(e\) and factor \(f\) to use in a vector. Instead, to find the best \(e\) and \(f\) for a vector, we designed a novel two-level sampling mechanism, inspired by the findings in subsection 2.6. Specifically, from Figure 3, we conclude that there is a \texttt{limited} set of best combinations of exponent \(e\) and factor \(f\) for the vectors in a dataset. Our sampling mechanism goes as follows: on the first sampling level, ALP samples \(m\) equidistant values from \(n\) equidistant vectors of a \texttt{row-group}. We define a \texttt{row-group} as a set of \(w\) consecutive vectors of size \(v\). The total number of values obtained from this first sampling is equal to \(m \times n\). For each vector \(n_i\) we find the best combination of exponent \(e\) and factor \(f\). This search is performed on the entire search space (i.e., 253 possible combinations). The \texttt{best} combination is the one which minimizes the sum of the exception size and the size of the bit-packed integers resulting from the encoded \(m\) values. This process yields \(n\) combinations (one for each vector). From these \(n\) combinations we only keep the \(k\) 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 \(k\) are yielded. If the same best combination is found in every vector, there would only be 1 combination. Hence, we define during runtime a \(k'\) which is smaller than or equal to \(k\) that represents the number of yielded combinations. Once we have found the \(k'\) best combinations, we proceed to the second level of sampling.

The second level of sampling (Line 5 of Algorithm 1) samples \(s\) equidistant values from a vector. Then, it tries to find the combination of exponent \(e\) and factor \(f\) which performs the \texttt{best} on the \(s\) sampled values. However, this time, the search is performed only among the \(k'\) 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 \(k_{e_{1}}\) and \(k_{e_{2}}\), is worse or equal to the performance of the combination \(k_{e_{j}}\), we stop the search and \(k_{e_{j}}\) is selected to encode the entire vector. If \(k'\) is equal to 1, this second sampling level is omitted for all the vectors inside the \texttt{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 \texttt{row-group}. Hence, the time spent is amortized into \(w \times v\) encoded values. The second sampling level happens once for each vector and it will only occur if \(k' > 1\). 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).

### 3.3 Decompression

ALP decompression builds upon the \texttt{ALP\_dec} procedure (Formula 2) 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 \texttt{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 for ALP decoding is presented in Algorithm 2.

### 3.4 ALP\(_{rd}\): Compression for Real Doubles

During the first level of sampling ALP will detect whether the doubles in a \texttt{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
Algorithm 3: \(\text{ALP}_{d}\) Compression and Decompression.

```cpp
1 // ENCODING. //
2 p, DICT = ALP::RD::ADAPTIVE_SAMPLING(input_rowgroup);
3 left_vec, right_vec = ALP::RD::READ_ROWGROUP_HEADER();
4 for (i = 0; i < VECTOR_SIZE; ++i) {
5   double n = input_vec[i];
6   int64 left_vec[i], right_vec[i] = ALP::RD::DECODE(p);
7   BITPACK(right_vec);
8   SKEWDIR_BITPACK(left_vec, DICT);
9 }
10 // DECODING. //
11 p, DICT = ALP::RD::READ_ROWGROUP_HEADER();
12 left_vec = BITUNPACK_DECODEDICT(encoded_left_vec, DICT);
13 right_vec = BITUNPACK(encoded_right_vec);
14 decoded_vec = ALP::RD::DECODE(p) {
15   for (i = 0; i < VECTOR_SIZE; ++i) {
16     int64 left, int64 right = left_vec[i], right_vec[i];
17     decoded_vec[i] = ALP::RD::GLUE(left, right, p);
18   }
19 }
```

in subsection 2.6 which hinted to us that even on these doubles, their front-bits tend to exhibit low variance. We named this approach \(\text{ALP}_{d}\), which stands for \textit{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 \(\text{ALP}_{d}\) is presented in Algorithm 3.

**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 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-bits values in an exception array, together with an array containing 16-bits 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-bits values. Both BP and DICTIONARY encodings implementations are available in our FastLanes library.[3]

**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 instruction tend to be fast; hence we do not use such an approach (explicitly). Because we use small dictionaries of size \(2^3 = 8\) and the front-bits are maximally 16-bits wide; we note that we could implement decoding by preloading the dictionary (maximally 8x16-bit 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 [5, 27] and adding in the decompressed right part (using vectorized SHIFT and OR, fused together in a GLUE primitive seen in Line 18 of Algorithm 3). Notice again that all operations are performed in a tight loop over arrays (vectorized query processing [51]) 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.

### 4 EVALUATION

We experimentally evaluate ALP with respect to its compression ratio and [de]compression speed using all analyzed datasets in Table 1 against six competing approaches for lossless floating-point compression: Gorilla [38], Chimp / Chimp128 [29], Patas [24], Elf [28] and PDE [31]. Furthermore, we also compare against one general-purpose compression approach: Zstd [14]. To further test the robustness of ALP we tested its speed on different hardware architectures which are described in Table 3 and using Auto-vectorized, Scalar and SIMDized code. In subsection 4.3 we present end-to-end query speed benchmarks of ALP on Tectorwise [23] to test its performance in a real system. Finally, in subsection 4.4 we present a version of ALP for 32-bits floats and evaluate it on machine learning data.

**Sampling Parameters.** Based on Figure 3, we define the maximum number of combinations \(k\) as 5. The number of vectors \(w\) inside a row-group is fixed to 100 to emulate the usual modern OLAP engines row-group sizes (e.g., DuckDB [41]). The size of every vector \(v\) is fixed to 1024 to comfortably fit in the CPU cache [7]. On the first sampling level, the number of vectors sampled per row-group \(m\) is set to 8, and the number of values sampled per vector \(n\) is set to 32. Finally, on the second sampling level, the number of values sampled per vector \(a\), is set to 32. \(m, n, a\) and \(w\) 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[10]. ALP uses the FastLanes library [3] to perform the lightweight encoding and decoding on its output (i.e., FFOR, DICTIONARY, BP). Gorilla, Chimp, Chimp128 and Patas were implemented in C++, Gorilla was implemented by ourselves, and the other implementations were stripped from

---

**Table 3: Hardware Platforms Used**

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Scalar ISA</th>
<th>Best SIMD ISA</th>
<th>CPU Model</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Ice Lake</td>
<td>x86_64</td>
<td>AVX512</td>
<td>8375C</td>
<td>3.5 GHz</td>
</tr>
<tr>
<td>AMD Zen3</td>
<td>x86_64</td>
<td>AVX2 (256-bits)</td>
<td>EPYC 7R13</td>
<td>3.6 GHz</td>
</tr>
<tr>
<td>Apple M1</td>
<td>ARM64</td>
<td>NEON (128-bits)</td>
<td>Apple M1</td>
<td>3.2 GHz</td>
</tr>
<tr>
<td>AWS Graviton2</td>
<td>ARM64</td>
<td>NEON (128-bits)</td>
<td>Neoverse-N1</td>
<td>2.5 GHz</td>
</tr>
<tr>
<td>AWS Graviton3</td>
<td>ARM64</td>
<td>NEON (128-bits)</td>
<td>Neoverse-V1</td>
<td>2.6 GHz</td>
</tr>
</tbody>
</table>

[10]https://github.com/cwida/ALP
the DuckDB codebase [40] 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 benchmarks were carried
out using code from the original authors. Finally, we used the
Facebook’s implementation of Zstd in C [14], configured at the default
compression level (3).

4.1 Compression Ratio
Table 4 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
KB vectors, even if 31 of those 32 vectors are not needed.

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 in-
tegers encoded as doubles (Table 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
to have a low compression ratio, the profit that ALP can get. Nevertheless, on data with many dupli-
cates, 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 con-
secutive). 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 com-
pression is then calculated as 1024 divided by the tuples per cycle.

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 a high variability in value precision. Those

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Gor.</th>
<th>Ch.</th>
<th>Ch.128</th>
<th>Patas</th>
<th>PDE</th>
<th>Elf</th>
<th>ALP</th>
<th>LWC+</th>
<th>Zstd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Air-Pressure</td>
<td>24.7</td>
<td>23.9</td>
<td>19.3</td>
<td>27.9</td>
<td>30.2</td>
<td>10.5</td>
<td>16.5</td>
<td>16.0</td>
<td>8.7</td>
</tr>
<tr>
<td>Blockchain</td>
<td>20.2</td>
<td>19.0</td>
<td>18.3</td>
<td>25.2</td>
<td>26.5</td>
<td>7.9</td>
<td>12.8</td>
<td>12.3</td>
<td>10.2</td>
</tr>
<tr>
<td>CMS/1</td>
<td>37.8</td>
<td>34.6</td>
<td>28.2</td>
<td>36.8</td>
<td>40.7</td>
<td>25.4</td>
<td>35.7</td>
<td>31.0</td>
<td>24.5</td>
</tr>
<tr>
<td>CMS/25</td>
<td>65.4</td>
<td>59.5</td>
<td>57.2</td>
<td>70.1</td>
<td>63.9</td>
<td>48.6</td>
<td>61.1</td>
<td>57.1</td>
<td>56.5</td>
</tr>
<tr>
<td>CMS/9</td>
<td>17.1</td>
<td>18.7</td>
<td>25.7</td>
<td>26.5</td>
<td>18.9</td>
<td>7.1</td>
<td>15.8</td>
<td>11.3</td>
<td>14.7</td>
</tr>
<tr>
<td>Crypto</td>
<td>56.6</td>
<td>58.6</td>
<td>57.3</td>
<td>72.8</td>
<td>74.1</td>
<td>16.8</td>
<td>22.7</td>
<td>14.2</td>
<td>17.2</td>
</tr>
<tr>
<td>Gov/10</td>
<td>58.1</td>
<td>45.9</td>
<td>34.2</td>
<td>35.9</td>
<td>36.8</td>
<td>30.1</td>
<td>51.0</td>
<td>31.0</td>
<td>22.4</td>
</tr>
<tr>
<td>Gov/5</td>
<td>2.4</td>
<td>2.3</td>
<td>1.9</td>
<td>2.6</td>
<td>2.6</td>
<td>2.4</td>
<td>4.2</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>Gov/30</td>
<td>10.3</td>
<td>8.9</td>
<td>12.0</td>
<td>19.3</td>
<td>8.2</td>
<td>8.0</td>
<td>7.5</td>
<td>4.2</td>
<td>4.2</td>
</tr>
<tr>
<td>Gov/54</td>
<td>5.7</td>
<td>5.0</td>
<td>10.4</td>
<td>17.1</td>
<td>2.8</td>
<td>5.4</td>
<td>3.1</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>Gov/60</td>
<td>2.7</td>
<td>2.4</td>
<td>1.8</td>
<td>2.7</td>
<td>2.7</td>
<td>2.2</td>
<td>3.4</td>
<td>0.6</td>
<td>0.6</td>
</tr>
<tr>
<td>Medicare/1</td>
<td>49.9</td>
<td>42.9</td>
<td>32.5</td>
<td>35.9</td>
<td>42.8</td>
<td>29.8</td>
<td>59.4</td>
<td>38.8</td>
<td>30.4</td>
</tr>
<tr>
<td>Medicare/9</td>
<td>17.0</td>
<td>17.1</td>
<td>20.6</td>
<td>26.3</td>
<td>10.2</td>
<td>16.0</td>
<td>12.3</td>
<td>11.3</td>
<td>13.9</td>
</tr>
<tr>
<td>NYC/29</td>
<td>36.8</td>
<td>29.6</td>
<td>28.7</td>
<td>38.8</td>
<td>69.5</td>
<td>32.6</td>
<td>60.4</td>
<td>24.7</td>
<td>20.5</td>
</tr>
<tr>
<td>POI-lat</td>
<td>66.0</td>
<td>57.7</td>
<td>73.7</td>
<td>71.7</td>
<td>69.3</td>
<td>62.5</td>
<td>55.3</td>
<td>55.3</td>
<td>48.1</td>
</tr>
<tr>
<td>POI-lon</td>
<td>38.0</td>
<td>32.3</td>
<td>28.8</td>
<td>43.2</td>
<td>35.7</td>
<td>23.9</td>
<td>31.8</td>
<td>18.8</td>
<td>18.8</td>
</tr>
<tr>
<td>SD-bench</td>
<td>51.1</td>
<td>45.7</td>
<td>19.2</td>
<td>23.6</td>
<td>30.8</td>
<td>34.3</td>
<td>16.2</td>
<td>22.0</td>
<td>11.8</td>
</tr>
<tr>
<td>NON-TS</td>
<td>37.7</td>
<td>34.0</td>
<td>31.8</td>
<td>39.6</td>
<td>32.8</td>
<td>24.9</td>
<td>25.7</td>
<td>23.1</td>
<td>23.3</td>
</tr>
<tr>
<td>ALL-AVG</td>
<td>42.2</td>
<td>37.7</td>
<td>28.7</td>
<td>35.5</td>
<td>31.4</td>
<td>23.1</td>
<td>21.7</td>
<td>18.8</td>
<td>20.6</td>
</tr>
</tbody>
</table>

ALP: Adaptive Lossless floating-Point Compression SIGMOD ’24, June 09–15, 2024, Santiago, Chile

11https://github.com/Spatio-Temporal-Lab/elf

The smaller this metric, the more compression is achieved (un-
compressed data is 64 bits per value). ALP achieves the best
performance in average (excluding Zstd). *: ALP was used.
Table 5: Average compression and decompression speed as tuples processed per computing cycle of all datasets on the Ice Lake architecture.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Compression</th>
<th>Decompression</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALP</td>
<td>0.487</td>
<td>2.609</td>
</tr>
<tr>
<td>Chimp</td>
<td>0.042</td>
<td>0.039</td>
</tr>
<tr>
<td>Chimp128</td>
<td>0.040</td>
<td>0.040</td>
</tr>
<tr>
<td>Elf</td>
<td>0.010</td>
<td>0.012</td>
</tr>
<tr>
<td>Gorilla</td>
<td>0.052</td>
<td>0.042</td>
</tr>
<tr>
<td>PDE</td>
<td>0.002</td>
<td>0.387</td>
</tr>
<tr>
<td>Patas</td>
<td>0.060</td>
<td>0.157</td>
</tr>
<tr>
<td>'Zstd'</td>
<td>0.035</td>
<td>0.101</td>
</tr>
</tbody>
</table>

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.

Figure 1 shows the result of this experiment. 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 shows the average amount of tuples per cycle processed in compression and decompression for every algorithm along all datasets. ALP is faster than all other approaches in both compression and decompression.

ALP is ≈7x faster than PDE; which is the second-best at compression 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 decomposition since the latter has complex if-then-else (i.e. branch mispredictions) and data dependencies that not only cause wait cycles, but also prevent SIMD. Zstd resides in a middle position in that it achieves better compression speed than PDE and Elf, and decomposition speed only slower than Patas and PDE.

**ALP on Different Architectures.** In order to investigate the performance robustness of ALP, we tested on all currently mainstream CPU architectures, as described in Table 3. CPU turbo-chasing 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 decompression procedure: SIMDized, Auto-vectorized and Scalar. The SIMDized implementation uses explicit 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 4 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 the 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.

**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 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 [15]. 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 is used. The bottom plot of Figure 5 shows the result of this experiment.

**Sampling Overhead in Compression** ALP implements a two-level sampling mechanism to find the correct encoding method and parameters, described in section 3.2. 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.
4.3 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 [23]. 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) and 256GB of RAM with a bandwidth of 18.75 Gbps. The reported results are an average of 32 executions of one query. Elf was not included in this analysis due to the lack of an implementation in C++.

### Table 6: End-to-end performance on City-Temp in the Tectorwise system, measured in Tuples per CPU cycle per core.

We benchmarked end-to-end query speed of ALP and the other floating-point compressors, when integrated in the research system Tectorwise [23]. 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) and 256GB of RAM with a bandwidth of 18.75 Gbps. The reported results are an average of 32 executions of one query. Elf was not included in this analysis due to the lack of an implementation in C++.

### Table 6: End-to-end performance on City-Temp in the Tectorwise system, measured in Tuples per CPU cycle per core.

We benchmarked end-to-end query speed of ALP and the other floating-point compressors, when integrated in the research system Tectorwise [23]. 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) and 256GB of RAM with a bandwidth of 18.75 Gbps. The reported results are an average of 32 executions of one query. Elf was not included in this analysis due to the lack of an implementation in C++.

### Table 6: End-to-end performance on City-Temp in the Tectorwise system, measured in Tuples per CPU cycle per core.

We benchmarked end-to-end query speed of ALP and the other floating-point compressors, when integrated in the research system Tectorwise [23]. 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) and 256GB of RAM with a bandwidth of 18.75 Gbps. The reported results are an average of 32 executions of one query. Elf was not included in this analysis due to the lack of an implementation in C++.

### Table 6: End-to-end performance on City-Temp in the Tectorwise system, measured in Tuples per CPU cycle per core.

We benchmarked end-to-end query speed of ALP and the other floating-point compressors, when integrated in the research system Tectorwise [23]. 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) and 256GB of RAM with a bandwidth of 18.75 Gbps. The reported results are an average of 32 executions of one query. Elf was not included in this analysis due to the lack of an implementation in C++.

### Table 6: End-to-end performance on City-Temp in the Tectorwise system, measured in Tuples per CPU cycle per core.

We benchmarked end-to-end query speed of ALP and the other floating-point compressors, when integrated in the research system Tectorwise [23]. 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) and 256GB of RAM with a bandwidth of 18.75 Gbps. The reported results are an average of 32 executions of one query. Elf was not included in this analysis due to the lack of an implementation in C++.
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 6 the performance metric is reversed: lower is better. We present the summing work in the SUM query (=SUM/scans, because SUM also scans) as the lower part of the stacked bar: it is roughly 3 cycles per tuple. Figure 6 confirms our results across the board: ALP is much faster end-to-end than the other compressors, even faster than uncompressed, and scales well.

COMP. ALP again is the fastest when compressing (Table 6): 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: (i) the extra effort in storing meta-data, (ii) the variable amount of exceptions (which are rather costly at compression time) and (iii) the first sampling phase which was not present in the micro-benchmarks.

4.4 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 POI’s, Basel’s, Medicare/1, and NYC/29); and 32-bit ALP works on them. This leads to the same compressed representation as in 64-bits (Table 4); but given that the uncompressed width is 32-bits, the 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 will be commonalities in their sign and exponent parts (IEEE 754) that ALP could take advantage of. Therefore, we also ported ALP 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 7; with ALPrd for 32-bit floats achieving the best compression ratios out of all the other algorithms (28.1 bit/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[47]. However, if this is not desired or possible; ALPrd thus can provide some useful lossless compression for ML.

5 RELATED WORK
The techniques developed for floating-point compression can be categorized mainly into three groups: (i) Predictive schemes, (ii) XOR schemes and (iii) Integer encoding schemes.

Predictive Schemes were one of the first novel approaches designed to compress floating-point data [4, 16, 30]; even in the context of geometry data [18, 22]. In these approaches, a function is used to generate a predicted value based on patterns found within the data prior to the value to encode. The idea behind this approach is that the predicted value and the value to encode are similar enough such that an operation (usually ADD) between their exponent and mantissas represented as integers yield a compressible chain of bits. Ratanaworabhan et al. [45] demonstrated that such an operation could be a bitwise XOR. Based on that, Burtscher and Ratanaworabhan developed FPC [10], which achieved better compression ratios and speed compared to previous approaches.

XOR Schemes. Pelkonen et al. [38] re-evaluated the 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 works as well as using a predictor. This assumption motivated the development of Gorilla. Gorilla compresses floats by doing a bitwise XOR with the immediate previous value. Next, it encodes the resulting chain of bits as 0 in case of a perfect XOR (i.e. equal values), otherwise, it encodes the resulting number of leading zeros and significant bits. Gorilla is faster on [de]compression than prediction schemes since encoding and decoding are achieved using a simple XOR with the immediate previous value instead of tuning and running a prediction function.

Gorilla Variants. Chimp [29] refined Gorilla by exploiting properties of the bit-chains yielded by the XORing process in time series data. Chimp distinguishes four different encoding modes based on the number of leading and trailing zeros of the XOR result to optimize compression ratios. It was jointly developed with a variant...
called Chimp128 in which 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. This idea of looking among previous values for the XOR was first introduced by Bruno et al. [9]. Chimp128 proved to be substantially better than FPC, Gorilla and other general-purpose compression schemes (e.g. Snappy, LZ4) in terms of compression ratio and speed [29].

In order to improve Chimp decompression speed, DuckDB Labs developed Patas [24]. The goal was to get a variant of Chimp128 faster at [de]compression, which it achieves by its design with a single encoding method (fewer branch-mispredictions) and byte-aligned bit-manipulation (less CPU work). Patas encodes for every value a block of 2 bytes containing the 7 bits previous value index, the number of significant bytes and the number of trailing zeros. Patas trades compression ratio for a ∼75% speed improvement at decompression time compared to Chimp128. In the context of analytical databases decompression speed is important for obtaining fast query results. On the other hand, a recently proposed XOR scheme called Elf [28] trades [de]compression speed for compression ratio by erasing bits from the mantissa at encoding time to make the XOR result more compressible. Afterwards, it losslessly reconstructs the double at decoding. As seen by our results, Elf gains ∼19% in compression ratio over Chimp128 at the expense of ∼4x slower compression and decompression. In contrast to Patas and Elf, ALP improves Chimp128 in all aspects.

While Chimp128 seemed to be clearly superior to Gorilla, our results show that it can actually perform better than Chimp128 (and even Elf) in datasets with consecutive runs of zeros (e.g. Gov/26, Gov/40). On this type of data Gorilla (and also Chimp) do not need the extra 7-bits to make a reference to one of the past 128 values since the most optimal value to XOR is always the previous one.

Integer Encoding Schemes. Doubles can also be compressed by taking advantage of their visible decimal representation [48]. PseudoDecimals [31] (PDE) formally introduces a lossless approach to perform this encoding process. PDE tries to encode a double with a division between an integer and an inverse factor of 10 under the assumption that the double was generated from a DECIMAL. This is why we refer to this type of encoding as Decimal-based encoding. ALP presents a strongly enhanced version of this approach introducing the idea of using large exponents and mitigating the effects of those with an additional multiplication that gets rid of trailing zeros. ALP is designed for vectorized execution, and introduces an adaptive mechanism for high-precision decimals (i.e. ALPRd). ALP prefers multiplication over division since division is an expensive operation in most ISAs [17]. PDE and ALP have the advantage that their output is further compressible using other lightweight encoding schemes such as DICTIONARY, RLE, FOR or DELTA [6, 15, 31].

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 π/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 π-multiplicated data. One may question why none of the datasets requires true double precision, nor is any 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). These new thin floats, 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 [19], and proposed alternatives like unum and posit[21]. 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 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 scans to fully compress 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 [6, 31], and recall that in our evaluation it already beat zstd (18.8 vs. 20.6) when cascading on Dictionary and RLE.

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