Camel: Efficient Compression of Floating-Point Time Series

Anonymous Author(s)

Abstract

Time series compression encodes the information in a time-ordered sequence of data points into fewer bits, thereby reducing storage costs and possibly other costs. Compression methods are either general or XOR-based. General compression methods are time-consuming and are not suitable in streaming scenarios, while XOR-based methods are unable to consistently maintain high compression ratios. Further, existing methods compress the integer and decimal parts of floating-point values as a whole, thus disregarding the different characteristics of the two parts. We propose *Camel*, a new compression method for floating-point time series with the goal of advancing the compression ratios and efficiency achievable. *Camel* compresses the integer and decimal parts of the double-precision floating-point numbers in time series separately; and instead of performing XOR operations on values using their previous value, *Camel* identifies values that enable higher compression ratios. *Camel* also includes means of indexing compressed data, thereby making it possible to query compressed data efficiently. We report on an empirical study of *Camel* and 11 lossless and 6 lossy compression methods on 22 public datasets and three industrial datasets from AliCloud. The study offers evidence that *Camel* is capable of outperforming existing methods in terms of both compression ratio and efficiency and is capable of excellent compression performance on both time series and non-time series data.

Keywords

time series, time series compression, time series query

1 Introduction

Time series compression is a fundamental task in time series data management and analysis, providing benefits such as data storage reduction and transmission acceleration [27]. Time series data is usually produced in streaming environments with high frequencies [45], especially true for **floating-point time series** comprising floating-point values. For instance, Alibaba Cloud's autonomous database service generates millions of database metric data points per second, consuming substantial storage space [46]. Compression techniques can significantly decrease storage demands, facilitating more efficient management of large-scale time series data while utilizing less storage capacity [36, 37, 48]. Moreover, compression methods can lower the computational resources required for time series data analysis. Overall, compression plays a crucial role in supporting efficient data management and analysis processes.

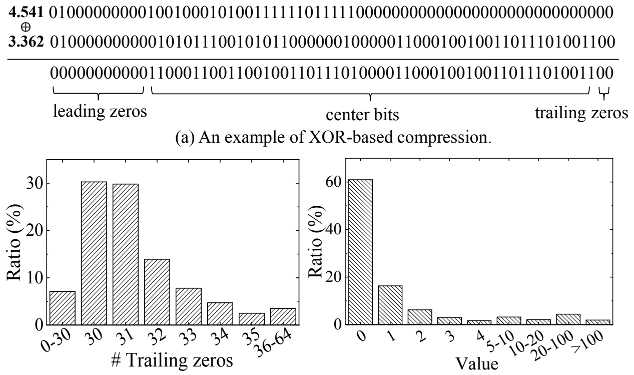
In this paper, we target floating-point time series compression, where existing methods employ either lossy or lossless techniques. Specifically, lossy floating-point compression methods (e.g., *ZFP* [39] and *MDZ* [50]) achieve high compression ratios by sacrificing data reconstruction accuracy. However, in many scenarios such as scientific computing and database applications, lossy compression is not acceptable. In contrast, traditional general-purpose lossless compression methods, such as *LZ4* [13] and *XZ* [39], often operate on batches of data by employing specific strategies like dictionaries and hash lists. While they can achieve high compression rates, the compression process often proves time-consuming, posing inefficiencies for compressing floating-point time series data. Moreover, these methods do not consider time series data features, such as temporal correlations between elements. There are also machine learning-based methods [30, 31, 47] that predict compressed values using statistical models. However, they are often time-consuming to train, and struggle to adapt to new data while maintaining effective compression performance in streaming settings.

Considering that floating-point time series are often continuously generated, streaming compression methods have gained attention [35, 38, 41], which are typically based on XOR operations. As shown in Fig. 1(a), there is a time series comprising double-precision floating-point values, with the current value and its previous value being 4.541 and 3.362, respectively. The XOR-based compression reduces storage size by preserving the XORed result between these two values, i.e., $\hat{v} = 4.541 \oplus 3.362$. Due to the similarity of consecutive values in a time series, the XORed results include numerous leading or trailing zeros. Thus, we only store its center bits along with the counts of leading zeros to restore $\hat{v}$. During decompression, it recovers the current value 4.541 by another XOR operation $\hat{v} \oplus 3.362$. This compression approach typically requires less storage compared to storing the original values themselves. Therefore, increasing the number of trailing zeros in XOR results plays an important role in improving compression ratios [35].

Following this way, the early *Gorilla* algorithm [41] assumes that XOR results contain leading zeros and many trailing zeros. Thus, it uses 5 bits to record the number of leading zeros and 6 bits to store the center bits. *Chimp* [38] points out that XOR values have many trailing zeros, implying that using 5 bits to record leading zeros is suboptimal. Thus, it optimizes the encoding of XOR values and uses fewer bits to record leading zeros. Most recently, *ELF* [35] shows that XOR results obtained by *Chimp* [38] and *Gorilla* [41] cannot obtain many trailing zeros. *ELF* [35] introduces a novel XOR encoding strategy based on erasure, i.e., erasing parts of the original data to obtain more trailing zeros. In Fig. 1(b), it's evident that the number of trailing zeros in *ELF*-compressed data varies significantly (with around 90% falling below 32). This necessitates allocating 4–6 bits for storing the counts of center bits and additional bits to store the center bits themselves, which adds to the space requirement. Moreover, none of the existing XOR-based compression supports querying compressed data. Because they rely on previous values, indexing compressed data becomes challenging. Motivated by these

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX



(a) An example of XOR-based compression.

(b) Distribution of the number of trailing zeros in a Bird-migration datasets, other datasets show a similar trends.

(c) Distribution of the difference between the integer part of the current value and its preceding value.

Figure 1: Case Study

observations, we target proposing an efficient and effective compression algorithm for floating-point time series data, addressing two key challenges in the process.

Challenge I: How to achieve stable and high compression ratios? Although the SOTA XOR-based streaming compression approaches achieve high compression ratios, the number of trailing zeros resulting from XOR operations on values and their previous values exhibit randomness and do not consistently maintain high ratios. As shown in Fig. 1(b), 95% of compressed data has fewer than 36 trailing zeros, and the number of trailing zeros varies. As illustrated in Fig. 1(b), 95% of compressed data shows fewer than 36 trailing zeros, with variability in the number of trailing zeros. This is because previous XOR-based methods relies on the similarity between two consecutive data points in the streaming time series data. The more similar two consecutive data points are, the more trailing zeros they can produce. However, instead of directly using the previous data points for XOR operations, *Camel* searches for a new value *dxor* for XOR operations in order to maintain more trailing zeros (e.g., 40-50), which can lead to a reduction in the required number of bits for storing center bits while still achieving high and stable compression ratios. Besides, as shown in Fig. 1(c), the differences in the integer parts of 95% of the time series are less than 2. Consequently, we opt to compress the integer and decimal parts of floating-point values separately to enhance the compression ratio. For the integer part, we employ a simple but effective differencing approach, capitalizing on the stability of the integer part. On the other hand, the decimal part undergoes compression using a novel XOR strategy, which incorporates new compression and decompression algorithms designed to exploit the distinct characteristics of integers and decimals, enabling significant storage space savings.

Challenge II: How to accelerate compression and decompression? Although the state-of-the-art streaming compression algorithm, *ELF* [34, 35], achieves relatively good compression ratios, it requires additional erasing and restoring, which makes it unsuitable for streaming time-series data. It is desirable to reduce the compression and decompression time while maintaining high compression ratios. Instead of directly adopting the strategy of XOR operations between a current value and the previous value, we identify a new value based on the number of decimal places, ensuring that XORing this value with the current value, e.g., $1.7 \oplus \underline{1.2}$, $1.78 \oplus \underline{1.03}$, and $1.942 \oplus \underline{1.067}$, produces a stable and regular pattern of trailing zeros. In the three examples, the underlined values represent the identified XOR values for the original values. The numbers of trailing zeros after XORing the underlined values with the original values are 51, 50, and 49, respectively. This calculation does not require complex erasure logic and enables both compression time reductions and improved compression ratios.

When reviewing existing compression methods on floating-point values with many decimal digits, they typically make a trade-off between space efficiency and precision loss, meaning that when the error threshold is lowered, the compression performance deteriorates [32]. *Camel* often discards trailing digits to achieve high compression ratios while maintaining valuable information. The retained information is then subjected to lossless compression to ensure both high precision and compression. For instance, retaining 4 significant digits, the precision achievable by *Camel* can reach 99.9999%. *Camel* is thus capable of high precision and compression on values with excessively many decimal digits. Overall, *Camel* offers both lossless and lossy compression capabilities, making it suitable for various applications. Building upon this, we further develop *Camel* index, which operates on compressed data and facilitates four types of time-series queries without decompression. We make the following contributions.

- We propose *Camel*, a streaming floating-point time series compression algorithm that compresses the integer and decimal parts of floating-point values separately. *Camel* enables both high compression ratios and efficiency in real-time compression scenarios.
- We design a strategy that involves finding XOR values that enable more stable counts of trailing zeros when compressing decimal parts. It improves the compression ratio and also accelerates the compression and decompression.
- We present a lossy compression method with high precision. To adapt to different application scenarios, the number of decimal places can be configured, maintaining lossless compression within specified decimal ranges, thus enhancing compression ratios while achieving high precision.
- We develop a new index structure for compressed data, enabling support for core query types, including range, value, time, and segment queries queries with small index sizes.
- We evaluate *Camel* with 11 state-of-the-art lossless and 6 lossy compression algorithms on 22 public datasets and 3 industrial datasets. The results offer solid evidence that *Camel* is capable of substantial improvements in compression ratio and efficiency, and that the proposed index is effective.

The rest of the paper is organized as follows. We provide the preliminaries in Section 2. In Sections 3, 4 and 5, we present the new compression algorithms and the new index for compressed data. Section 5 presents the experimental results. We review related work in Section 6 and conclude the paper in Section 7.

2 Preliminaries

We first provide basic definitions, and then introduce the IEEE 754 Double Precision Floating Point Format. Table 1 provides an overview of frequently-used notation.

Table 1: Frequently Used Notations

Notation	Description
$TS = \langle (t_1, v_1), (t_2, v_2), \ldots, (t_n, v_n) \rangle$	Floating-point time series, where t_i is a timestamp and v_i is a floating-point value.
$DF(v) = \pm(d_h d_{h-1} \ldots d_1.d_1 d_2 \ldots d_l)_{10}$	Decimal format of v, where $d_i \in \{1, 2 \ldots, 9\}$; "+" is usually omitted if $v > 0$.
$DP(v) = l$	Decimal place count l of value v.
$BF(v) = \pm(b_{h'} b_{h'-1} \ldots b_1.b_1 b_2 \ldots b_{l'})_2$	Binary format of v where $b_i \in \{0, 1\}$. "+" is usually omitted if $v > 0$.
$IE(v) = (s, e_1, e_2 \ldots e_{11}, m_1, m_2 \ldots m_{52})$	IEEE 754 Double Precision Floating Point Format of v.
$v.d^{int}, v.d^{dec}$	The integer and decimal parts of v in decimal format.
$v.b^{int}, v.b^{dec}$	The integer and decimal parts of v in binary format.
$\hat{vi}, \hat{vd}$	The references of vi and vd.
$dxor$	The value used to perform the XOR operation with v.
center_bits, leading_zeros	The number of center bits and leading zeros of in binary format.
$Diff(v.d^{int}, v'.d^{int})$	The difference between $v.d^{int}$ and $v'.d^{int}$.

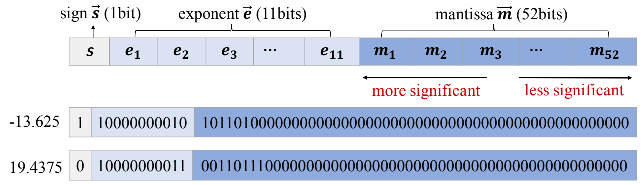


Figure 2: IEEE 754 Double-precision Floating Point Format

2.1 Floating Point Time Series

Definition 2.1 (Floating Point Time Series). A time series TS is a sequence of timestamp and value pairs in time order: $TS = \langle (t_1, v_1), (t_2, v_2), \cdots, (t_n, v_n) \rangle$. For each pair (t_i, v_i) $(1 \leq i \leq n)$, timestamp t_i is the time when value $v_i \in R$ was recorded, and v_i is a floating point number in decimal format.

Floating-point time series compression typically involves two aspects: timestamp compression and value compression. While existing methods (such as delta encoding and delta-of-delta encoding [41]) demonstrate high performance at timestamp compression, a substantial potential for floating-point compression remains. Thus, we target the compression of floating-point values in time series, exploring their representation through the utilization of the double precision floating-point format. Recall that a float number occupies 64 bits in memory.

Definition 2.2 (Decimal Format). The decimal format of a double precision floating-point number v can be denoted as $DF(v) = \pm d^{int}.d^{dec}$, where $v.d^{int}$ denotes the integer part of v that consists of h digits $(d^{int}_h d^{int}_{h-1} \cdots d^{int}_1)$, while $v.d^{dec}$ denotes the decimal part of v that consists of l digits $(d^{dec}_1 d^{dec}_2 \cdots d^{dec}_l)$. Here, each digit $d^{int}_i (1 \leq i \leq h)$ or $d^{dec}_j (1 \leq j \leq l) \in \{0, 1, \ldots, 9\}$.

We express the decimal place count of v as $DP(v) = l$. An example of a floating point time series with 3 data points is $TS = \langle (1, 2.3419), (2, 2.2346), (3, 2.4178), (4, 3.0157) \rangle$. For all four data points, their decimal place l is 4 (i.e., $DP(v) = 4$).

Definition 2.3 (Binary Format). The binary format of v can be represented as $BF(v) = \pm b^{int}.b^{dec}$, where $v.b^{int}$ denotes the integer part of v that consists of h' digits $(b^{int}_{h'} b^{int}_{h'-1} \cdots b^{int}_1)$, while $v.b^{dec}$ denotes the decimal part of v that consists of l' digits $(b^{dec}_1 b^{dec}_2 \cdots b^{dec}_{l'})$. Here, each digit $b^{int}_i (1 \leq i \leq h')$ or $b^{dec}_j (1 \leq j \leq l') \in \{0, 1\}$. The binary format satisfies:

$$d^{int} = \sum_{i=1}^{h'} b^{int}_i \times 2^{i-1}, d^{dec} = \sum_{j=1}^{l'} b^{dec}_j \times 2^{-j+1} \quad (1)$$

Here, "±" is the sign of v. If $v \geq 0$, "+" is omitted. For example, $BF(10) = (1010)_2$, $BF(0.125) = (0.001)_2$, $BF(\text{-}13.625) = -(1101.101)_2$.

The binary XOR operation on two binary values yields an output determined by the following principle. If a pair of input bits differ, the output is 1; otherwise (the two input bits are identical), the output is 0. For example, 1011101 ⊕ 1000100 = 0011001, where ⊕ is the XOR operation.

2.2 Double Precision Floating Point Format

The IEEE 754 Double Precision Floating Point Format (IEEE 754 format, for brevity) is a standard used in computing to represent double-precision real numbers using a total of 64 bits. The bits are divided into three parts as shown in Fig. 2: a 1-bit sign bit, an 11-bit exponent, and a 52-bit mantissa. First, the sign bit indicates the sign of the number, where 0 represents a positive number and 1 represents a negative number. Next, the exponent occupies 11 bits and represents an exponent value, with the actual exponent being expressed through an offset. This allows for the effective encoding of a wide range of numbers. Finally, the mantissa consists of 52 bits and is used to store the specific numerical values after the decimal point, providing high-precision representation capabilities. The IEEE 754 format of a double value v can be presented as $IE(v) = (s, e_1, e_2 \ldots e_{11}, m_1, m_2 \ldots m_{52})$, wheres $e_i (1 \leq i \leq 11)$, and $m_j (1 \leq j \leq 52)$ are binary values that satisfy:

$$DF(v) = (-1)^s \times 2^{\mathbf{e}-1023} \times (1.m_1 m_2 \ldots m_{52})_2 \, , \quad (2)$$

where $\mathbf{e}$ is the decimal value of the 11-bit exponent, i.e., $\mathbf{e} = \sum_{j=1}^{11} e_j \times 2^{j-1}$.

To illustrate, we apply the IEEE 754 format to the values -13.625 and 19.4375. Specifically, for v = -13.625, the transformation is detailed as follows: (1) The sign bit is set to 1 as the value is negative. (2) The integer and decimal parts of v are transformed into binary format, resulting in 1101.101. (3) The exponent is calculated by determining the position of the decimal point in the normalized binary format and converting it to binary exponent form. In this example, 1101.101 is converted to 1.101101, moving the decimal point three places to the left. Thus, the exponent is represented as

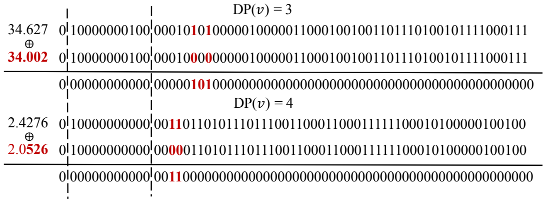


Figure 3: Case Study of *Camel* Decimal Compression

(3 + 1023) = 1026. In binary format, this is 10000000010. (4) Finally, combining the sign bit, exponent, and mantissa yields the IEEE 754 format. Representing 19.4375 follows the same process:

$$-13.625 = (-1)^{\mathbf{1}} \times 2^{\mathbf{1026}-1023} \times (1.\mathbf{101101})_2$$
$$19.4375 = (-1)^{\mathbf{0}} \times 2^{\mathbf{1027}-1023} \times (1.\mathbf{00110111})_2$$

The transformation results are shown in Fig. 2. In detail, in the IEEE 754 format, the number of leading zeros is the number of zeros before the first non-zero bit, while the number of trailing zeros is the number of zeros after the last non-zero bit, and the center bits encompass all bits between the first and last non-zero bits in the normalized part of a floating-point number.

3 Reference Discovery Strategy

To achieve efficient compression on floating-point time series data, we first propose the reference discovery strategy of *Camel*, including two components, i.e., decimal reference (i.e., $\hat{vd}$) discovery and integer reference (i.e., $\hat{vi}$) discovery.

3.1 Decimal Reference Discovery

The proposed decimal reference discovery of *Camel* is rooted in a fundamental observation: given a double value v, it is possible to identify another value $dxor$ such that $v \oplus dxor$ consistently produces a stable count of leading zeros. As shown in Fig. 3, for the value 34.627 with $DP(34.627) = 3$, we identify 34.002 such that 34.627 ⊕ 34.002 produces a output containing "101" with the rest being "0". Similarly, for the value 2.4276 with $DP(2.4276) = 4$, the result of 2.4276 ⊕ 2.0526 contains "11" while the remaining digits are "0". In the above two cases, we only need to store "101" or "11" and the count of leading zeros to compress the decimal part. Based on that, if the count of leading zeros is related to the $dxor$ value (i.e., $\hat{vd}$), storing "101" or "11" is sufficient for restoring $dxor$ of 34.627 and 2.4276. This yields two technical questions:

- Q1: How do identify $dxor$ in an effective way?
- Q2: Is there a relation between the count of leading zeros and l?

To solve Q1, we provide efficient means of calculating $dxor$ as follows. Consider a double value v with decimal format $DF(v) = \pm(d_h^{int}\, d_{h-1}^{int} \ldots d_1^{int}.\, d_1^{dec} d_2^{dec} \ldots d_l^{dec})_{10}$ where decimal place count of $DP(v) = l$. IEEE 754 format of v is $I\!E(v) = (s, e1, e_2 \ldots e_{11}, m_1, m_2 \ldots m_{52})$. We can find a value $dxor$, where $BF(dxor)$ has the same integer part as $BF(v)$. The decimal part of $dxor.d^{dec}$ is calculated as follows when $v.d^{dec} \neq 2^{-l}$:

$$dxor.d^{dec} = v.d^{dec} - 2^{-l} \times (\lfloor v.d^{dec}/2^{-l} \rfloor) \tag{3}$$

Theorem 3.1. *The difference between $BF(v)$ and $BF(dxor)$ starts at position b and ends at position e, where $b \geq 1$ and $e \leq l$. Therefore, the center bits of $I\!E(v) \oplus I\!E(dxor)$ (i.e., $\hat{vd}$) are $T' = T_b, T_{b+1} \ldots T_e$, where $T_b = 1, T_e = 1$, and the length of T' is $cn(\hat{vd}) = e - b + 1$.*

This solves Q1, on how to find $dxor$. The key advantage of the *Camel* algorithm over traditional XOR methods is its stability. In *Camel*, the central bits of $v \oplus dxor$ are related to l, while the central bits of $v \oplus dxor$ in conventional XOR algorithms are unstable.

Theorem 3.2. *For any double precision floating point value $v = (d^{int}.d_1^{dec} d_2^{dec} \ldots d_l^{dec})_{10}$, the integer part $v.d^{int} \neq 0$. Suppose the number of bits in $v.b^{int}$ is p and the center bits of $I\!E(v) \oplus I\!E(dxor)$ are T' ($T' = T_b, T_{b+1} \ldots T_e$). Then the number of leading zeros of $\hat{vd}$ (i.e., $I\!E(v) \oplus I\!E(dxor)$) is $leading\_zeros = 12 + (p - 1) + (b - 1)$.*

Note that, the proofs of Theorem 3.1 and 3.2 can be found in Appendix B[1]. Considering the decimal part of v (i.e., $v.d^{dec}$), based on Theorem 3.1, $dxor.b^{dec}$ can only differ from $v.b^{dec}$ before the l-th position. Thus, the leading zeros of $I\!E(\hat{vd})$ depend on $cn(\hat{vd})$. As $cn(\hat{vd})$ starts with b and ends with e, the center bits in $v.b^{dec} \oplus dxor.b^{dec}$ start in location $b-1$. The count of leading zeros in $I\!E(v) \oplus I\!E(dxor)$ (i.e., $\hat{vd}$) is $12 + (p - 1) + (b - 1)$). This means that we need to save b, p, the center bits of $\hat{vd}$, the decimal count l and $dxor.d^{dec}$ when compressing the decimal part vd of v.

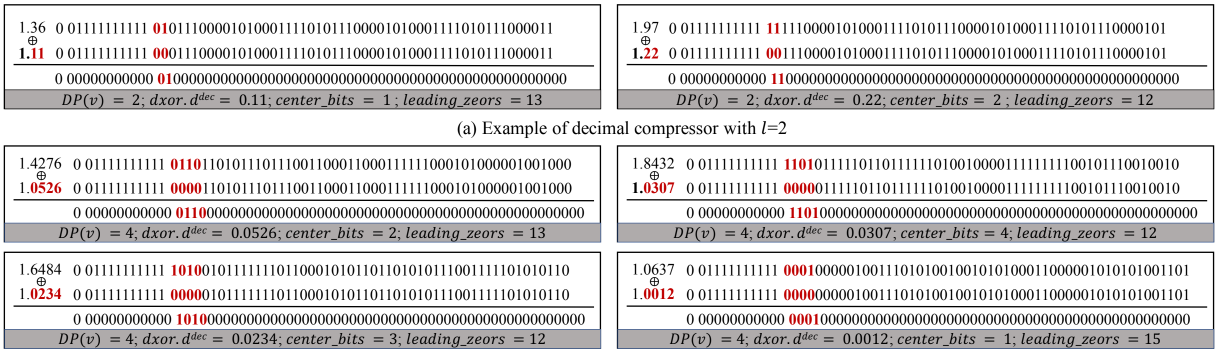
Furthermore, there are two methods for optimizing the storage. (1) Setting $b = 1$. Since b is always less than the decimal place count l, we only need to store a bit sequence of length l starting from T_1 (i.e., $b = 1$), which definitely includes the center bits. (2) Fixing $v.d^{int} = 1$. Since the decimal part is compressed separately and independent of the original data's integer part, we can fix the integer part to 1, ensuring that p is always 1. Using these two methods, the count of leading zeros in $I\!E(v) \oplus I\!E(dxor)$ is 12, and the values of b and p no longer need to be stored.

This addresses Q2 regarding the correlation between the count of leading zeros and the variable l. Based on the analysis provided, the count of leading zeros in $I\!E(v) \oplus I\!E(dxor)$ is closely linked to both the decimal place count l and the center bits of $\hat{vd}$. Consequently, we can efficiently reconstruct $\hat{vd}$ by only storing the bit series from the $12 + 1$-st bit to the $12 + l$-th bit in $\hat{vd}$. This streamlined approach minimizes storage requirements while retaining the necessary information for reconstruction.

Fig. 4 shows an example of *Camel* decimal compression with $l = 2$ (i.e., Fig. 4(a)) and $l = 4$ (i.e., Fig. 4(b)). Given the value $v = 1.36$, according to Theorem 3.1, we get $dxor = 1.11$. The central bits of $((I\!E(1.36) \oplus I\!E(1.11))$ are $\{1\}$ (where $b = 2$), indicating that $0.36 - 2^{-2} = 0.11$. Similarly, for $v = 1.0637$ with $l = 4$, the central bits of $(I\!E(1.0637) \oplus I\!E(1.0012))$ are $\{1\}$ (i.e., $b = 4$), signifying that $0.0637 - 2^{-4} = 0.0012$. As their integer parts are identical (i.e., 1), the preceding 12 elements of the compressed results are the same. Therefore, their leading zeros is calculated as $12 + (b - 1)$.

Compared to performing XOR with the previous value, employing this strategy to find $dxor$, yields fewer and more stable center bits. Specifically, the number of trailing zeros that can be calculated and deduced based on the given number of decimal places of a floating-point number. Specifically, according to Theorems 3.1 and 3.2, given a double precision floating point value $v = (d^{int}.d_1^{dec} d_2^{dec} \ldots d_l^{dec})_{10}$, the number of bits in $v.d_1^{dec}$ is p, and

[1] http:githubXXX



(a) Example of decimal compressor with l=2

(b) Example of decimal compressor with l=4

Figure 4: Examples of *Camel* Decimal Compression

the center bits of $IE(v) \oplus IE(dxor)$ ($\hat{vd}$) are $T' = T_b, T_{b+1} \dots T_e$, where b and e denote the first position and end position of $T_i = 1$, respectively. The numbers of center bits and leading zeros in $\hat{vd}$ are $e - b + 1$ and $12 + (p - 1) + (b - 1)$, respectively. Thus, the number of trailing zeros in $\hat{vd}$ is $64 - (e - b + 1) - (12 + (p - 1) + (b - 1)) = 53 - p - e$. As the integer parts during decimal compression are set to 1 for simplify (i.e., $p = 1$), the number of trailing zeros in $\hat{vd}$ is $52 - e$. The value of e is always less than the decimal count l and larger than 1. The max decimal count of the double precision float is 17. Therefore, the number of trailing zeros is always $35 \le \#trailing\_zeros < 52$.

Furthermore, other XOR methods require saving the number of leading zeros and center bits, with the irregular center bits needing several bits to be reserved for length storage. However, our calculated *dxor* is stable and regular; thus, only the number of decimal places and the center bits need to be saved. Moreover, *dxor* constrains the decimal part's value below 2^{-l} (l denotes the decimal count). We convert *doxr* to an integer by $dxor \times 10^l$. By this way, we can further optimize the space on the integer representation, i.e., we store this value in different range intervals, e.g., using 3 bits for values less than 8, 5 bits for values below 32, etc.

3.2 Integer Reference Discovery

Diving further into the handling of the integer part $v.d^{int}$, we capitalize on the inherent correlations present within the integer part of time-series data. Often, there exists a strong association between the integer part of the current value and that of the previous value, indicating a trend of either incremental or decremental movement in the time series data. To capture this trend more efficiently, we choose compression by calculating the differences between the integer parts of successive values.

Definition 3.1. For a double precision floating point time series $TS = \langle (t_1, v_1), (t_2, v_2), \dots, (t_n, v_n) \rangle$, the decimal part of each value in TS is $\langle (t_1, v_1.d^{int}), (t_2, v_2.d^{int}), \dots, (t_n, v_n.d^{int}) \rangle$ in decimal format. Given a current value $(t_t, v_t.d^{int})$ and $t > 1$, the previous value $(t_{t-1}, v_{t-1}.d^{int})$. We represent the difference between $v_t.d^{int}$ and $v_{t-1}.d^{int}$ as $Diff(v_t.d^{int}, v_{t-1}.d^{int}) = v_t.d^{int} - v_{t-1}.d^{int}$. The reference of $v_t.b^{int}$ (i.e., $\hat{vi}_t$) can be denoted as

$$\hat{vi}_t = \begin{cases} IE(v_t.d^{int}) & \text{if } t = 1 \\ BF(v_t.d^{int} - v_{t-1}.d^{int}) & \text{if } t > 1 \end{cases} \tag{4}$$

For the initial value in TS, we preserve $IE(v_1.d^{int})$. For subsequent values, we compute the difference between the decimal part of the current value and the previous one (i.e., *diff*), saving *BF*(*diff*) during the compression process for the decimal value.

This approach significantly simplifies data representation and lowers storage costs. By compressing the integer part using differences, we not only achieve a more compact data storage format but also obtain a clearer insight into the dynamic changes within the integer part. This enhanced understanding enables robust support for various applications of time-series data.

4 Compression and Decompression

We proceed to provide the details of *Camel*, including its compression (i.e., $Camel_{cmp}$) and decompression (i.e., $Camel_{decmp}$).

4.1 Overview of *Camel* Compression

$Camel_{cmp}$ initially computes the difference between the integer part of the current value and that of the previous value (excluding the first value), resulting in the compressed integer part $\hat{vi}_i$. Subsequently, $Camel_{cmp}$ determines a new value $dxor_i$ and performs an XOR operation between $dxor_i$ and the current value v_i to derive the compressed decimal part (i.e., the XORed value) $\hat{vd}_i$. By combining these compressed integer and decimal parts, we obtain the compressed binary value v_i'.

Next, we use the inverse operation $Camel_{decmp}$ for decompression. Specifically, it reads the difference between the integer part of the current value and the previous value, excluding the first element, and then it adds this to the integer part of the previous decompressed value to decompress the integer part $v_i.d^{int}$. Then we continue to read $\hat{vd}_i$ and $dxor_i$, performing an XOR operation on both to restore the decimal part. Finally, we add the decompressed integer part $vi.d^{int}$ and decimal part $v_i.d^{dec}$ to reconstruct the double-precision floating-point value v_i.

4.2 $Camel_{cmp}$

4.2.1 *Integer compression.*

Algorithm 1 presents the integer compression process of *Camel*. When $t = 1$, indicating the first value in the sequence, it directly writes v_t as a 64-bit entry to the output (lines 1–3). For subsequent values, the algorithm computes the difference *diff* between the integer part of the current value v_t and the previous value v_{t-1}. If the absolute value of *diff* is no more than

Algorithm 1 Integer-Compression(v_t, out)

```
1:  if t == 1 then
2:      out.write($v_t$, 64); // compress the first value
3:  else
4:      // compress the remaining values
5:      $diff = v_t.d^{int} - v_{t-1}.d^{int}$;
6:      if $|diff|$ <= 1 then
7:          out.write($diff$ + 1, 2);
8:      else
9:          out.writeBit($diff$ >= 0 ? ‘1’ : ‘0’);
10:         out.writeBit ($diff$ < 8) ? ‘0’ : ‘1’;
11:         out.write($diff$, ($diff$ < 8) ? 3 : 16);
12:     end if
13: end if
```

Algorithm 2 Decimal-Compression(v_t, out)

```
1:  $l$ = calculate_decimal_count($vd(v_t)$); // calculate the decimal
    count of value $v_t$
2:  out.write($l$, 2);
3:  if $v_t \geq 2^{-l}$ then
4:      out.writeBit(1);
5:      $dxor(v_t)$ = calculate_dxor($v_t.d^{dec}$, $l$); // calculate the $dxor$
        based on theorem 3.1
6:      $\hat{vd}(v_t) = v_t.d^{dec} \oplus dxor(v_t)$;
7:      out.write($\hat{vd}(v_t)$ >>> (52 − $l$), l);
8:  else
9:      out.writeBit(1);
10:     $dxor(v_t) = v_t.d^{dec}$;
11: end if
12: $dxor' = dxor(v_t) \times 10^l$;
13: if $l \leq 1$ then
14:     out.write($dxor'$, $l + 1$);
15: else if $l == 2$ then
16:     out.writeBit($dxor' \leq 4$ ? ‘0’ : ‘1’);
17:     out.write($dxor'$, $dxor' \leq 4$ ? 2 : 5);
18: else
19:     $max = \lceil \log_2(2^{-l} \times 10^l) \rceil$;
20:     $thresholds$ = [$2^{(i*max/4)}$ for i in range(1, 4)];
21:     $index$ = next(($i$ for $i$, $threshold$ in enumerate($thresholds$) if
        $dxor$ <= $thresholds$), 3);
22:     out.writeBit($dxor'$, ($index$+1)*$max$/4);
23: end if
```

Algorithm 3 Integer-Decompression(in)

```
1:  if t==1 then
2:      $v_t$ = in.read(64); // decompress the first value
3:  else
4:      // decompress the remaining values
5:      $range$ = in.read(2);
6:      if $range$ not equal 3 then
7:          $diff = range - 1$;
8:      else
9:          $symbol$ = in.read(1);
10:         $flag$ = in.read(1);
11:         $diff$ = in.read($flag$ == 0 ? 3 : 16);
12:         $diff$ = $symbol$ == 1 ? $diff$ : $-diff$;
13:     end if
14:     $v_t = diff + v_{t-1}$;
15: end if
16: return $v_t$;
```

1, the algorithm writes the sign of $diff + 1$ to the output using 2 bits (lines 7–8). Subsequently, it utilizes ‘1’ to denote positive differences and ‘0’ to denote negative differences (line 10). To further enhance compression efficiency, it adopts variable-length coding by introducing a prefix based on the range of $diff$. If $diff$ is below 8, the prefix is set to ‘0’; otherwise, it is set to ‘1’. Then, the absolute value of $diff$ is written to the output using an appropriate number of bits (3 or 16) determined by the selected prefix (lines 11–12). This method ensures effective compression of the data by considering the magnitude of differences within the sequence.

4.2.2 Decimal compression. Algorithm 2 provides a detailed description of the decimal compression process in *Camel*. First, it computes the decimal count l of v_t and then writes l with two bits to the output (lines 1–2). Next, it checks if v_t is greater than or equal to 2^{-l}. If yes, it writes 1 to the output, calculates $dxor(v_t)$ according to Theorem 3.1, updates $\hat{vd}(v_t)$, and writes the compressed version of $\hat{vd}(v_t)$ to the output (lines 3–7). In case v_t is smaller than 2^{-l}, it still writes 1 to the output and keeps the original value of $\hat{vd}(v_t)$ (lines 9–10). Afterwards, it converts the decimal part of v_t to an integer $dxor'$ based on $\hat{vd}(v_t)$ and l. Depending on the value of l, it writes different representations of $dxor'$ to the output. If l is 1 or less, it writes $dxor'$ with a length of $l + 1$ to the output (line 14). If l equals 2, it writes a bit based on whether $dxor'$ is less than or equal to 4, followed by writing $dxor'$ and choosing between ‘2’ or ‘5’ accordingly (lines 16–17). For l greater than 2, the algorithm calculates the maximum value (max), representing the number of bits in the binary representation of the integer $2^{-l} \times 10^l$. It then divides max into four intervals and sets thresholds. Based on how $dxor$ compares to these thresholds, it determines an index and writes the bit representation of $dxor'$ along with a fraction of the maximum length determined by the value of the variable index (lines 19–22).

4.3 *Camel*$_{decmp}$

4.3.1 Integer decompression. Algorithm 3 provides a comprehensive outline of the integer decompression process in *Camel*. When the variable t equals 1, indicating the processing of the decimal part of the first value, the algorithm reads a 64-bit value from the input stream and assigns it to v_t (lines 2–3). For subsequent values, it reads a $range$ value. If this value differs from 3, the $diff$ is set to $range - 1$ (lines 7–8). Then, it reads the symbol $symbol$ and sign flag $flag$ and calculates the difference $diff$ based on the range using variable-length decoding (lines 11–13). Finally, it reconstructs the value v_t using the difference, the flag, and the previous value v_{t-1} (line 15) and returns the result v_t (line 17). This decompression process ensures the accurate reconstruction of the original data.

4.3.2 Decimal decompression. Algorithm 4 provides a detailed description of *Camel*'s decimal decompression. It first reads the decimal place count l from the input (line 1). Next, it reads a flag value, denoted as $flag$, indicating whether $v_t.d^{dec} \geq 2^{-l}$ (line 2). If $flag$

Algorithm 4 Decimal-Decompression(*in*)

```
1:  $l = in.read(2)$;
2:  $flag = in.read(1)$;
3:  if $flag == 1$ then
4:      $center\_bits = in.read(l)$;
5:      $vd(\hat{v_t}) = center\_bits <<< 12$;
6:      $v_t.d^{dec} = IE(1 + Restore(l, in)) \oplus vd(\hat{v_t})$ - 1;
7:  else
8:      $v_t.d^{dec} = Restore(l, in)$;
9:  end if
10: return $v_t.d^{dec}$;
```

Algorithm 5 Restore(l, in)

```
1:  if $l \leq 1$ then
2:      in.read($l + 1$);
3:  else if $l == 2$ then
4:      $flag$ = in.read(1);
5:      $dxor$ = in.read($flag == 0$ ? 2 : 5);
6:  else
7:      $max = \lceil \log_2(2^{-l} \times 10^l) \rceil$;
8:      $flag$ = in.read(2);
9:      $indexes$ = ["00", "01", "10", "11"];
10:     $dxor$ = in.read((index+1)*max/4 for index in indexes if $index$ == $flag$);
11: end if
12: return $dxor$ ;
```

is equal to 1, it proceeds to read the center_bits from the input. The algorithm then recovers the XORed result of $vd(v_t \oplus dxor)$ by left-shifting the *center_bits* by 12 positions (lines 4–5). Further, it computes $v_t.d^{dec}$ by applying the operation $IE(1 + Restore(l, in)) \oplus vd(\hat{v_t}) - 1$ (line 6). If *flag* is not equal to 1, it directly sets $v_t.d^{dec}$ to the result of $Restore(l, in)$ (line 8).

Algorithm 5 further elaborates on the "Restore" function. First, if l is at most 1, it reads $l + 1$ bits from the input (lines 1–2). If l is equal to 2, it reads a flag value and, based on the flag value, reads either 2 or 5 bits as *dxor* from the input (lines 4–5). Conversely, if l exceeds 2, it calculates the maximum value based on the logarithm and reads a 2-bit flag from the input. Subsequently, the algorithm determines the appropriate index based on the flag and reads *dxor* accordingly (lines 7–10).

4.4 Discussion

4.4.1 Flexibility. When compressing the integer part of data in *Camel*, we can adjust the maximum bit value for compression based on the dataset's characteristics. For instance, in the case of a City Temperature (CT) dataset [5] that describes daily temperature variations of major cities, it is reasonable to assume that temperature differences generally do not exceed 100 degrees. Therefore, we can adjust the 16 bits to 8 bits to further reduce storage space.

4.4.2 Time Complexity. For each value, we calculate *dxor* in $O(1)$ time in decimal compression (i.e., Algorithm 2), and the other calculations are based on bit operations when compression (i.e., Algorithm 1), with time complexity $O(1)$. For restoring, integer and decimal decompression (i.e., Algorithms 3, 4, and 5), *Camel* sequentially reads data from an input stream, ensuring that all operations are completed in $O(1)$ time. For time series data with N values, the time complexity is $O(N)$.

4.4.3 Space Complexity. During compression and decompression, we only store and utilize the integer part of the previous value, yielding a space complexity of $O(1)$.

5 Index based on *Camel*

We proceed to describe an accompanying index for the compressed time-series data, designed to support core query types, including value, range, timestamp, and segment queries.

5.1 The overview of *Camel* Index

Figure 5 illustrates two indexing methods, including Split-index and Merge-index, for efficiently querying the timestamp based on time series value. The index (either Split-index or Merge-index) contains three parts, i.e., integer-tree, decimal-tree, and the timestamp index. Note that, the timestamp index in the split-index and the merge index is the same; thus, we only show the timestamp-index once in Fig. 5, for simplicity.

Split-index uses separate B^+-trees, i.e., Integer-tree and Decimal-tree, for managing integer and decimal parts of time-series data.

- Integer-tree stores the binary representations of integer parts. (i) Each leaf entry contains *key* (i.e., the variable binary representation of an specific integer, e.g., 110 for 5, 10 for 3) and a timestamp pointer list *tList* to link the timestamps whose corresponding integer values equal to *key*. Note that, m bits are used for each *key* (m denotes the length of *key*) and 8 bytes are used for each pointer. (ii) Each non-leaf entry contains *minkey* (i.e., the minimum *key* in its subtree) and a pointer to the root node of its sub-tree.
- Decimal-tree manages decimal parts using a XORed value. The leaf entry stores *key* (i.e., the *dxor* value of decimal part), a *falsetList* to store pointers to timestamps whose corresponding decimal values equal to *key*, and a pointer *TrueNodePtr* to a TrueNode. Each entry in TrueNode contains (i) *center_bits* center bits of XORed value, and (ii) a *truetList* to store pointers to timestamps whose center bits of XORed value equals to *centerBits*. Specifically, given a time-series data point (v, t), if $v.b^{dec} = key$, then we insert a pointer to t into *falsetList*, while if the center bits of $IE(v) \oplus IE(key)$ equals to *center_bits*, then we insert a pointer to t into *truetList*. Note that, in Decimal-tree, *key* is also with variable length m that occupies m bits, while *center_bits* is fixed to occupy l bits (l is used to control the precision).
- The timestamp index is a skip list structure that allows quick location of timestamps, with the final layer nodes providing pointers to the compressed data. Each timestamp in the skip list or timestamp list occupies 8 bytes.

Merge-index combines the integer and decimal parts into a single hierarchical structure. A B^+-tree is used to store the integer parts, with the leaf entities of the integer tree pointing to decimal trees. Thus, in the leaf entry of the integer-tree in Merge-index, it contains (i) a *key* (i.e., the variable binary representation of an

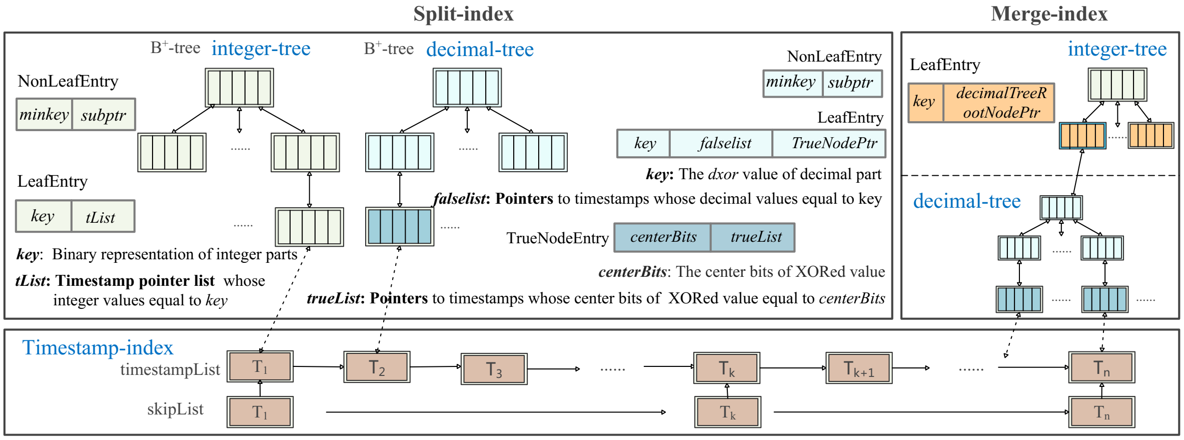

Split-index
Merge-index
B+-tree integer-tree
B+-tree decimal-tree
NonLeafEntry
minkey subptr
LeafEntry
key tList
key: Binary representation of integer parts
tList: Timestamp pointer list whose integer values equal to key
NonLeafEntry
minkey subptr
LeafEntry
key falselist TrueNodePtr
key: The dxor value of decimal part
falselist: Pointers to timestamps whose decimal values equal to key
TrueNodeEntry
centerBits trueList
centerBits: The center bits of XORed value
trueList: Pointers to timestamps whose center bits of XORed value equal to centerBits
integer-tree
LeafEntry
key decimalTreeRootNodePtr
decimal-tree
Timestamp-index
timestampList
skipList
T1
T2
T3
Tk
Tk+1
Tn
......

Figure 5: Index structure of *Camel* index

specific integer), and (ii) a pointer to the root node of the decimal tree. The decimal tree stores all possible decimal values with their integer parts equaling to *key*. Note that, the other information is same as that in Split-index.

Fig. 5 shows the differences between the Split-index and the Merge-index. The Split-index separates the integer tree and the decimal tree. In contrast, the Merge-index merges the integer tree and the decimal tree, with the leaf node entities of the integer tree pointing to the decimal tree. These two indexes can be independently applied to different query scenarios. Split-index takes longer query time due to extra computations but occupies less storage space. In contrast, queries via Merge-index are faster but Merge-index requires more space due to multiple decimal-trees, which is validated by our experiments.

Overall, the above trees facilitate computation of various query types, ensuring seamless access to different data points and enabling robust query processing. We can create different indexes based on query scenarios, like merging the decimal and integer trees (i.e., Merge-index) or building separate indexes for each (i.e., Split-index). Please refer to Appendix B[2] for the detailed index building process.

5.2 Query Processing

We proceed to cover the handling of key query types for streaming time-series data: value, range, timestamp, and segment queries. The overall query processing involves inputting the search values, compressing them, querying either the merge index or split index, and retrieving the results. Please refer to Appendix B for the detailed query processing using a running example.

i) Value Query *value*(v): Given a double-precision float value v, ***value***(v) returns a list *tList* of timestamps whose corresponding time series values equal to v. To answer ***value***(v), we first compress v using *Camel* to get $v.b^{int}$, *centerBits*, and $dxor$. For Split-index, we first search $v.b^{int}$ to get a candidate timestamp list, and then search $dxor$ and *centerBits* in the decimal-tree to get another candidate timestamp list. The final result is the intersection of these two candidate lists. For Merge-index, we first search $v.b^{int}$ in the integer-tree to get the specific decimal-tree, and then search $dxor$ and *centerBits* in this decimal-tree to get the final timestamp list.

ii) Range Query *range*(v_1, v_2): Given two float values v_1 and v_2, ***range***(v_1, v_2) also returns a list of timestamps *tList* whose corresponding time series values falling between v_1 and v_2. We first get the compressed values including $v.b^{int}$, *centerBits*, and $dxor$ of v_1 and v_2 respectively. For Split-index, we first search integer ranges to get the candidate timestamp list same as value queries, and then refine the result by searching decimal ranges. For Merge-index, we identify decimal trees by integer values and then search the result within the decimal range.

iii) Timestamp Query *time*(t): Given a timestamp t, ***time***(t) returns a time series value v. To answer ***time***(t), we locate the target node using the hierarchical structure of the skip list, similar to the binary search. This allows us to retrieve the compressed value ($v.b^{int}$, *centerBits*, and $dxor$) via a pointer to the Split-index or Merge-index. We then decompresses the time series v based on the compressed values.

iv) Segment Query: *block*(i): Given a the block index i, ***block***(i) returns a list of timestamps *tList*. To avoid memory consumption from expired data, we build indexes based on blocks. For Split-index, we only need to traverse integer-tree leaf nodes in Split-index to get all the result timestamps. However, for Merge-index, we need to visit all the leaf nodes in the decimal-trees to get all the result timestamps. Thus, Split-index is more efficient than Merge-index for block queries.

6 Experiments

6.1 Experimental Setting

<u>**Datasets.**</u> We use 22 public datasets, encompassing 14 time series (abbreviated TS) datasets and 8 non time series (abbreviated Non-TS) datasets, mirroring the dataset composition in the evaluation of *ELF*. While the TS datasets are ordered according to time, the Non-TS datasets are ordered at random. Moreover, we incorporate three real MySQL metric datasets from AliCloud: *Mysql.cpu_usage*, *Mysql.disk_usage*, and *Mysql.mem_usage*. Non-time series data exhibits considerable variation and does not follow any temporal. Such data is used to show that time series compression methods are more suitable for TS data than for non-TS data. Non-TS data poses particular challenges, as XOR-based methods rely on the principle that "XOR operations on continuous time-series data result in more

[2] http://githubXXX

Table 2: Compression Performance of *Camel* and Lossless Baselines

Datasets			Public datasets																						Industrial datasets		
			Time series														Non time series								Time series		
			CT	IR	WS	PM10	SUK	SUA	SDE	DT	BP	AP	BW	BT	BM	AS	FP	VC	BTR	SB	CLT	CLN	PLT	PLN	CPU	DISK	MEM
Compression ratio	**Streaming**	Camel	**0.15**	0.17	0.16	0.17	**0.15**	**0.17**	0.21	**0.18**	**0.38**	0.26	**0.28**	**0.28**	**0.27**	**0.27**	**0.17**	**0.21**	**0.27**	0.20	**0.37**	**0.38**	**0.28**	**0.28**	**0.35**	**0.34**	**0.36**
		ELF+	0.21	**0.14**	0.19	**0.11**	0.18	**0.17**	0.22	0.24	0.48	**0.25**	0.55	0.51	0.37	0.85	0.21	0.28	0.29	0.23	0.38	0.59	0.97	1.06	0.71	0.61	0.64
		ELF	0.25	0.21	0.25	0.16	0.22	0.24	0.26	0.31	0.56	0.31	0.59	0.58	0.42	0.85	0.23	0.34	0.36	0.27	0.56	0.63	0.96	1.06	0.77	0.69	0.72
		Chimp	0.64	0.59	0.81	0.46	0.52	0.64	0.67	0.77	0.77	0.65	0.88	0.85	0.72	0.77	0.47	0.86	0.67	0.55	0.92	0.98	0.90	0.99	0.96	0.86	0.89
		Chimp$_{128}$	0.32	0.24	0.23	0.21	0.29	0.23	0.27	0.35	0.72	0.54	0.71	0.47	0.50	0.77	0.34	0.36	0.55	0.27	0.78	0.85	0.90	0.99	0.95	0.75	0.85
		Gorilla	0.85	0.64	0.83	0.48	0.58	0.68	0.72	0.83	0.84	0.73	0.99	0.94	0.79	0.82	0.58	1.00	0.74	0.63	1.03	1.03	1.03	1.03	1.02	0.95	0.91
		FPC	0.75	0.61	0.85	0.5	0.74	0.7	0.73	0.82	0.81	0.67	0.92	0.9	0.75	0.82	0.62	0.91	0.69	0.59	0.96	1.00	0.95	1.00	0.97	0.82	0.92
	General	XZ	0.18	0.16	**0.15**	**0.11**	0.16	**0.17**	**0.19**	0.27	0.63	0.47	0.57	0.35	0.43	0.79	0.23	0.23	0.40	**0.13**	0.60	0.63	0.93	0.96	0.93	0.68	0.79
		Brotli	0.20	0.18	0.17	0.12	0.19	0.20	0.22	0.32	0.71	0.51	0.61	0.39	0.47	0.85	0.26	0.28	0.43	0.14	0.65	0.68	0.94	0.96	0.93	0.71	0.84
		LZ4	0.36	0.36	0.37	0.27	0.39	0.39	0.41	0.52	0.87	0.69	0.69	0.54	0.61	1.01	0.41	0.47	0.53	0.30	0.79	0.82	1.00	1.00	1.00	0.82	0.93
		Zstd	0.22	0.24	0.19	0.14	0.22	0.24	0.26	0.38	0.75	0.58	0.61	0.41	0.51	0.91	0.30	0.34	0.45	0.17	0.68	0.71	0.94	0.96	0.92	0.72	0.82
		Snappy	0.29	0.30	0.27	0.21	0.32	0.32	0.35	0.51	0.99	0.73	0.75	0.54	0.61	1.00	0.39	0.42	0.54	0.25	0.83	0.87	1.00	1.00	1.00	0.85	1.00
Compression rate (Mb/s)	**Streaming**	Camel	106	64	224	87	196	224	206	191	162	111	196	238	293	283	305	186	305	238	186	191	231	246	201	206	201
		ELF+	134	159	173	224	173	206	177	191	177	263	144	150	153	134	191	76	166	238	144	123	159	119	139	159	156
		ELF	50	27	159	109	109	106	105	114	89	109	86	94	99	35	125	134	141	162	127	111	94	83	91	125	117
		Chimp	177	246	318	363	318	347	332	318	332	**381**	318	332	318	283	381	263	318	381	305	283	318	293	293	305	305
		Chimp$_{128}$	**186**	318	332	293	305	363	318	293	162	206	177	238	201	153	305	212	177	318	162	162	162	170	156	156	156
		Gorilla	159	**347**	**424**	**477**	**449**	**449**	**509**	**449**	**509**	**381**	**449**	**477**	**477**	**347**	**477**	**449**	**449**	**545**	**477**	**477**	**449**	**477**	**424**	**449**	**477**
		FPC	97	212	173	182	305	283	283	186	318	305	201	212	293	272	212	201	166	224	206	170	186	173	156	201	196
	General	XZ	4	5	6	5	9	8	10	8	6	8	7	9	9	6	10	6	9	10	5	4	7	5	6	6	4
		Brotli	3	3	4	4	4	4	4	4	4	4	4	4	4	5	4	5	4	5	4	4	5	5	4	4	4
		LZ4	6	7	6	6	7	7	8	8	8	8	7	7	8	9	8	9	8	8	8	7	8	8	8	8	8
		Zstd	14	27	18	23	27	26	25	26	24	30	32	32	30	31	32	32	27	33	30	29	34	34	28	44	32
		Snappy	5	34	32	31	37	36	31	34	34	36	37	35	35	35	33	35	33	33	53	99	54	42	39	38	150
Decompression rate (Mb/s)	**Streaming**	Camel	60	132	206	424	238	246	283	231	201	**477**	173	231	293	318	224	191	293	177	166	166	186	206	173	173	182
		ELF+	136	173	224	246	254	254	246	263	254	293	186	212	224	263	283	206	224	263	196	156	238	224	166	206	212
		ELF	64	36	186	191	231	170	156	196	156	191	105	177	191	254	206	150	141	159	127	134	246	238	144	173	159
		Chimp	123	246	347	424	363	347	347	347	332	402	293	347	347	347	402	332	332	363	318	305	332	332	305	305	332
		Chimp$_{128}$	144	**402**	**477**	402	**449**	**509**	**449**	**449**	332	347	318	402	381	283	449	347	332	449	318	283	318	305	283	305	293
		Gorilla	64	206	402	**449**	**449**	449	**449**	**449**	**477**	**477**	**449**	**477**	**449**	**402**	**477**	272	449	509	449	424	424	449	424	449	477
		FPC	166	136	272	263	332	305	305	254	363	318	318	305	318	318	305	283	263	305	283	231	293	238	246	263	283
	General	XZ	17	49	51	57	48	50	49	35	18	23	19	27	22	13	43	33	28	81	19	17	13	12	13	17	14
		Brotli	42	98	71	121	112	45	112	121	102	116	109	123	107	102	125	129	139	206	103	95	112	117	106	111	107
		LZ4	**318**	231	83	173	224	238	272	254	283	231	103	272	272	305	119	**424**	**509**	**545**	**477**	**545**	**587**	587	**545**	**509**	509
		Zstd	136	105	37	97	119	127	132	173	170	201	206	212	206	218	186	272	212	318	263	254	293	283	238	139	305
		Snappy	48	212	150	162	218	224	238	231	305	272	254	254	254	246	224	272	363	477	449	509	477	**636**	**545**	477	**545**

trailing zeros” due to the similarity between consecutive data points. A detailed descriptions of the datasets is provided in Appendix A[3].

Baselines. We use six state-of-the-art lossless floating-point compression methods: *FPC* [25], *Gorilla* [41], *Chimp* [38], *Chimp*$_{128}$ [38], *ELF* [35], and *ELF+* [34]; five widely-used general compression methods: *XZ* [20], *Brotli* [21], *LZ4* [13], *Zstd* [1], and *Snappy* [17], and six compression methods: *Sim-Piece* [32], *Sim-Piece & ZStd* [32], *PMC-MR* [33], *Swing* [29], *min-max Quantization Int16* and *INT32* [15]. To enable fair comparisons, we optimize the stream implementation of *Gorilla* to be the same as that of *Chimp*, significantly enhancing *Gorilla*'s efficiency.

Metrics. We evaluate compression performance using three metrics: compression ratio and compression and decompression rate (Mb/s). We use precision to evaluate the accuracy of lossy compression methods. Precision is defined as $max(1 - |real\_value - lossy\_value|/real\_value, 0)$, where $lossy\_value$ is obtained following lossy compression.

Settings. We implement *Camel* in Java. All public datasets and source code of *Camel* are available[4]. Following the *ELF* study, we regard 1,000 records of each dataset as a block. Each compression method is executed on up to 100 blocks per dataset, and the average metrics of one block are finally reported.

We set the decimal place count $l = 4$, truncating longer decimal data accordingly. This allows us to compare *Camel* with the 4 lossy compression methods on 10 datasets (AP, BW, BT, BM, AS, PLT, PLN, CPU, DISK, MEM), where more than 4 decimal places are used. Generally, higher precision leads to lower compression ratios across all 5 methods. To ensure a fair comparison of compression ratios, the precision for Camel and the other 4 methods is maintained above 99.9%, with *Camel* achieving a higher precision compared to the other methods. This highlights *Camel*'s ability achieve high compression ratios while also achieving high precision. We evaluate query performance by averaging the query times of 100 randomly sampled sets of input values for 4 query types.

[3] http::/githubXXX

[4] https://anonymous.4open.science/r/camel

Table 3: Compression Performance of *Camel* and Lossy Baselines

Datasets			AP	BW	BT	BM	AS	PLT	PLN	CPU	DISK	MEM
Compression ratio	Streaming	Camel	0.26	0.28	0.28	0.27	0.27	0.28	0.28	0.35	0.34	0.36
		Sim-piece	0.39	0.40	0.49	0.41	0.53	0.45	0.45	0.55	0.53	0.58
		SP&ZStd	0.33	0.36	0.43	0.35	0.44	0.39	0.39	0.46	0.43	0.47
		Swing	0.73	0.99	0.96	0.73	0.93	0.96	0.98	0.99	0.95	0.99
		PMCR	0.74	0.97	0.99	0.70	0.94	0.97	0.97	0.99	0.92	0.98
	General	QINT16	**0.25**	**0.25**	**0.25**	**0.25**	**0.25**	**0.25**	**0.25**	**0.25**	**0.25**	**0.25**
		QINT32	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
Compression rate (Mb/s)	Streaming	Camel	111	196	**238**	293	283	231	246	201	206	201
		Sim-piece	25	21	22	24	9	19	20	21	23	22
		SP&ZStd	38	50	36	41	27	41	47	32	30	35
		Swing	20	17	17	16	6	17	18	17	17	18
		PMCR	58	63	63	58	24	62	64	60	60	60
	General	QINT16	**571**	**198**	158	637	599	705	714	713	642	641
		QINT32	**571**	196	186	**1056**	**838**	**1232**	**1219**	**1241**	**1041**	**1168**
Decompression rate (Mb/s)	Streaming	Camel	477	173	231	293	318	186	206	173	173	182
		Sim-piece	23	26	25	26	12	18	21	22	22	20
		SP&ZStd	52	34	30	34	11	26	31	30	29	50
		Swing	50	40	48	47	15	48	49	37	60	35
		PMCR	68	59	60	67	24	52	43	55	57	55
	General	QINT16	2318	5433	10144	5442	5707	10803	10777	7916	4315	4816
		QINT32	**6233**	**7028**	**10229**	**4408**	**5727**	**10499**	**10816**	**10795**	**5096**	**5229**
Precision	Streaming	Camel	99.99995	**99.99907**	99.99758	99.99988	99.99613	99.98715	99.97238	99.99656	**99.99985**	99.99986
		Sim-piece	99.99993	99.99796	99.98372	99.99960	99.99564	99.97203	99.97126	99.99061	99.99955	99.99968
		SP&ZStd	99.99993	99.99796	99.98372	99.99960	99.99564	99.97203	99.97126	99.99061	99.99955	99.99968
		Swing	99.99994	99.99856	99.94752	99.99686	99.99498	99.95467	99.95392	99.99470	99.99812	99.99944
		PMCR	99.99990	99.99512	99.93180	99.99857	99.99123	99.98283	99.96445	99.99549	99.99843	99.99955
	General	QINT16	99.99998	99.81893	99.97055	99.99848	99.95845	99.99314	99.97109	99.43352	86.52638	99.99367
		QINT32	**99.99999**	99.82900	**99.99999**	**99.99999**	**99.99999**	**99.99900**	**99.99600**	**99.99999**	86.61794	**99.99999**

6.2 Overall Compression Performance

6.2.1 Compression ratio. Tables 2 and 3 show comparisons with the lossless and lossy baselines, respectively. Bold represents the best results, and underline represents the second-best results.

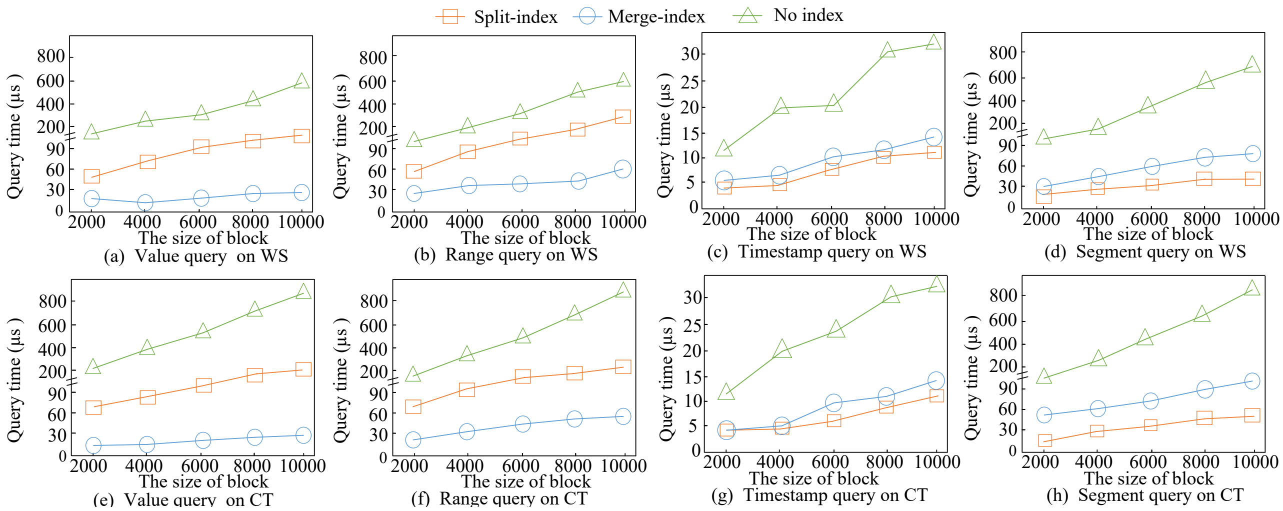
Comparison with streaming lossless methods. *Camel* achieves the highest compression ratio among all streaming lossless compression methods–*ELF+*, *ELF*, *Chimp*, *Chimp*$_{128}$, *Gorilla*, and *FPC*–on most datasets. It outperforms earlier XOR compression methods such as *Chimp*, *Chimp*$_{128}$, *Gorilla*, and *FPC* across all datasets. While *Chimp* enhances *Gorilla*'s encoding and introduces a faster data search with *Chimp*$_{128}$, *Camel* still excels with its integer-decimal split compression strategy that explots the characteristics of time series better. Notably, *Camel* outperforms *ELF+* and *ELF* on most datasets (22 out of 25), including on TS and non-TS datasets, by optimizing XOR operations to change fewer central bits. However, on datasets having many duplicates such as IR, PM10, and AP, the compression ratio is slightly higher for *ELF+*, as *ELF+* is more suitable for handling datasets with many duplicate values. Hence, in scenarios such as environmental monitoring, stock prices, database monitoring, weather observations, and traffic flow, time-series data typically changes continuously, with rare significant repetitions. *Camel* demonstrates excellent compression performance in these scenarios. However, in more static situations—such as data from malfunctioning devices and monitoring data from static scenes—where there are many zeros or identical values, other compression methods may perform better.

Comparison with general lossless methods. Most general lossless compression methods achieve good compression ratios, typically outperforming earlier XOR-based streaming lossless compression methods because they utilize an entire dataset's features. In contrast, streaming lossless compression methods do not possess knowledge of future values. Despite this, in most cases, *Camel* still outperforms *Brotli*, *LZ4*, *Zstd*, and *Snappy* across all datasets and achieves a better compression ratio than *Xz* on most datasets (20 out of 25 datasets). This is because *Camel* leverages the inherent correlation among integers in time series and identifies new XOR values to produce more trailing zeros for decimals.

Comparison with lossy methods. *Camel* provides its precision by preserving only the first four decimal digits and discarding any further ones, using lossless compression. Moreover, by employing XOR operations, it not only saves space but also achieves higher precision than the lossy compression methods.

6.2.2 Compression and decompression rate. Compression and decompression rates are presented in Tables 2 and 3.

Comparison with streaming lossless methods. *Camel*'s compression rate is notably faster than that of *ELF+* and *ELF* methods on the majority of datasets (20 out of 25), which require time-consuming erasing and restoring steps. However, *Camel*'s compression and decompression rate are slightly slower than those of earlier XOR-based data compression algorithms because it incorporates an additional step that calculates the replacement value for the decimal part and determines the number of decimal places,



(a) Value query on WS (b) Range query on WS (c) Timestamp query on WS (d) Segment query on WS

(e) Value query on CT (f) Range query on CT (g) Timestamp query on CT (h) Segment query on CT

Figure 6: Efficiency of four queries on WS and CT.

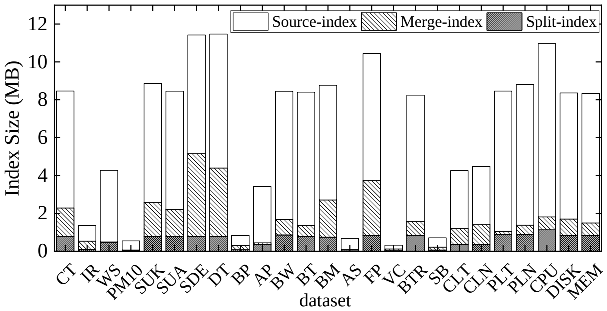


Figure 7: Comparison on the total index size.

which requires extra time. *Gorilla* achieves the fastest compression and decompression rates as it is the simplest of the XOR-based algorithms and considers fewer cases. *Camel* shows efficiency that is comparable with the other streaming algorithms. Actually, the compression/decompression time per data point is in microseconds, making *Camel* fully capable of meeting the requirements of real-time compression and streaming scenarios. In addition, *Camel* offers a superior compression ratio compared to the other streaming compression algorithms, especially Gorilla. This benefits large-scale streaming data scenarios.

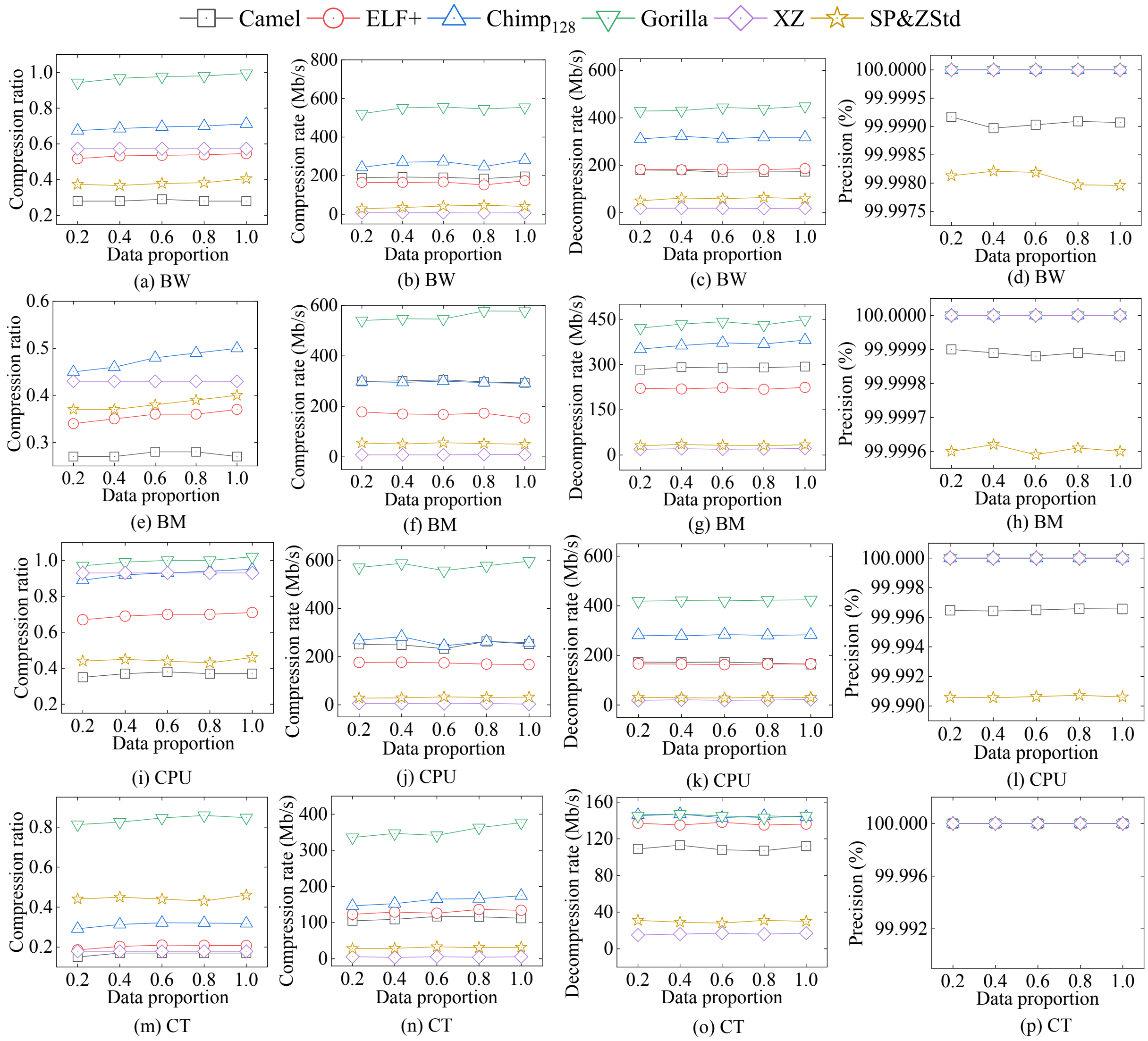
Comparison with general lossless methods. General lossless compression algorithms typically require longer compression times due to their more complex and time-consuming techniques. In contrast, floating-point compression algorithms are usually tailored for specific data types, resulting in simpler compression and faster compression rate. As shown in Table 3, the compression and decompression rates of general-purpose compression algorithms are 20–50 times slower than those of *Camel*. Although *XZ* achieves compression ratio similar to that of *Camel*, its compression rate is up to 30 times slower. Even the fastest general lossless compression algorithms, *Zstd* and *Snappy*, have compression rate that are are about 6 times slower than those of *Camel*, rendering them unsuitable for real-time scenarios.

Comparsion with lossy methods. The 4 lossy compression algorithms are 50 to 500 slower compression and decompression rate than *Camel* because they use considerable time to search for piecewise linear approximations. Next, *Sim-Piece* and *Sim-Piece & ZStd* (abbreviated SP & ZStd) require two orders of magnitude more time than do *PMCMR* and *Swing* because they need more time to organize line segments into minimum numbers of groups in order to improve the compression. Meanwhile, *Camel* only needs $O(1)$ calculations and XOR bit-wise operations to compress data. General lossy compression methods, including min-max Quantization Int16 and Int32 (QINT16 and QINT32), are very efficient due to their simple linear mapping for scaling data. However, it is unsuitable for streaming scenarios as it needs to know the maximum and minimum values of the entire datasets in advance, which are unavailable in the streaming setting. Additionally, mapping data to a smaller range may lead to precision loss, especially in cases when many outlier data points exist. For example, the precision in the DISK dataset is very low due to many outlier data points exist. We have further analyzed the stability of min-max quantization in Appendix C[5] to highlight its potential risks in the data compression.

6.3 Overall Query Performance

6.3.1 Query efficiency. To evaluate the query performance of the *Camel* index (including Split-index and Merge-index), a brute-force search method based on the compressed data (abbreviated 'No index') is included. Fig. 6 provides the results on WS and CT datasets, due to the space limitation. The first observation is that, the Merge-index performs the best for value and range queries (Figs. 6(a)-6(b) and Figs. 6(e)-6(f)), while the Split-index performs the best for timestamp and segment queries (Figs. 6(c)-6(d) and Figs. 6(g)-6(h)). This is because, for value and range queries, the Split-index needs additional intersection operations on two candidate lists obtained by Integer-tree and Decimal-tree respectively, resulting in additional query costs compared to Merge-index. In contrast, for timestamp and segment queries, Split-index does not additional intersection operations and Meter-index size is relatively larger due to redundant decimal trees, resulting in relatively higher query

[5] http://github



(a) BW (b) BW (c) BW (d) BW

(e) BM (f) BM (g) BM (h) BM

(i) CPU (j) CPU (k) CPU (l) CPU

(m) CT (n) CT (o) CT (p) CT

Figure 8: Scalability study.

costs of Merge-index. The second observation is that, the query cost increases with the growth of the block size due to larger search space. In addition, both Split-index and Merge-index support more efficient queries compared to the brute-force search baseline (i.e., No index), which verifies the effectiveness of our designed indexes.

6.3.2 Index size. Fig. 7 compares the sizes of three different index structures, where the index size includes all its components, i.e., the sizes of integer-tree, decimal tree(s) and the temporal index. As observed, the index size of Split-index or Merge-index is much smaller than that of the source index (i.e., the B$^+$-tree built on uncompressed data). This is because, the compressed float data instead of original float value is maintained in the Merge-index and Split-index. However, the Merge-index size is usually larger than that of Split-index. The reason behind is the Merge-index needs to construct a decimal-tree for each integer value, resulting in redundant storage.

6.4 Scalability Evaluation

To evaluate the scalability of Camel, we split datasets into segments at 20%, 40%, 60%, 80%, and 100% of the original data. Due to the space limitation, we select four datasets with more than 100,000 time series data points, each featuring different decimal counts. For clear illustration, we select four representative lossless baselines and one lossy baseline (that offer good performance among baselines) for comparison. Fig. 8 provides the corresponding results. Note that, for lossless baselines, the precision is 100%.

As observed, *Camel* and *XZ* demonstrate consistently stable performance in terms of compression ratio, compression rate, and decompression rate by varying the dataset size, while the performance of other methods slightly fluctuates. This stability of Camel is attributed to its XOR operations are independent of the data values and their distribution. In contrast, the performance of other XOR-based algorithms is influenced by the preceding values, leading to the performance fluctuations especially under the skewed data value distribution. *XZ* also achieves stable performance by dynamically constructing and updating dictionaries, but its performance in terms compression ratio, and compression/decompression rate is much lower compared *Camel*. Additionally, *SP&Zstd* experiences a decline in performance as the data volume increases (cf. Figs. 8(a) and 8(e)), due to the increased difficulty in achieving accurate linear approximations. Overall, *Camel* can achieve high stable compression ratio performance with high accuracy and relatively high compression/decompression rate.

7 Related work

Time series compression reduces data storage and processing overheads while preserving key information. Floating-point compression encompasses lossless and lossy algorithms.

7.1 Lossless Floating-Point Compression

Lossless floating-point compression methods reduce storage without losing any information. General and streaming compression algorithms exist.

General Compression. Universal compression algorithms are applicable across diverse data types and scenarios. Numerous highly effective universal compression exist, including *Brotli* [21], *LZ4* [13], *Zstd* [1], and *Snappy* [17]. *Brotli* [21] utilizes a dictionary and pattern matching for compression, establishing a dynamic dictionary to store previously encountered data blocks for identifying repetitive patterns. *LZ4* [13] divides input data into blocks for independent compression and decompression, while Snappy partitions data into fixed-size blocks, finding duplicate data segments within each block and replacing them with short references. *Zstd* [1] combines dictionary compression with multiple compression strategies to provide efficient universal data compression. *Zstd* [1] and *Snappy* [17] excel at balancing compression ratio and efficiency. While most universal compression methods are lossless and achieve good compression ratios, their compression efficiency is relatively low, making them unsuitable for streaming scenarios.

Streaming Compression. Lossless floating-point compression algorithms are either model-based or previous-value based. Model-based methods include the difference finite context method *DFCM* [42], which maps floating-point values to unsigned integers and predicts these values through predictors; and *FPC* [24, 25] that uses two context-based predictors in a streaming manner to better adapt to data changes. Other model-based methods [30, 31, 47] utilize machine learning models to capture distinctive features of series, ultimately selecting an optimal compression approach. The previous-value based methods include *Gorilla* [41] and *Chimp* [38], which use previous values as the predicted values, avoiding the problem of high prediction costs. Specifically, Gorilla optimizes encoding by recording the number of leading zeros, while Chimp further optimizes it. More recently, *ELF* [35] improves on *Chimp* and introduces the XOR encoding strategy, obtaining more trailing zeros by erasing part of the value; and *ELF+* [34] is an improved version of *ELF*, which optimizes *ELF*'s encoding method for significant counts, reducing redundant encoding, and achieving better compression ratios. Sprintz [23], a multivariate time series compression algorithm, supports integer time series, while we focus on floating-point compression. The *TVStore* method [22] enables users to maintain the importance of each data by integrating compressors and time-based functions. In our study, we assume that all data is equally important, or that importance information is not available.

7.2 Lossy Floating-Point Compression

Due to the complex storage format of floating-point data, compressing floating-point data without loss of precision is challenging. Thus, lossy floating-point compression methods have been proposed [32, 39, 40, 49, 50]. For example, the representative method *ZFP* [39] guarantees compression rules for grid data under a certain loss. *MDZ* [50] is an adaptive error-bounded lossy compression framework that optimizes compression for two execution models of molecular dynamics. *PMC-MR* [33] is a lossy compression algorithm for time-series data using piecewise constant approximation. It introduces a cache filter to predict the next data point's value within a certain error threshold, only recording new points when they violate the constraint. Elmeleegy et al. [29] introduce *Swing*, a method for joint and disjoint piecewise linear approximation (PLA). *GreedyPLR* [44] is a variant of *Swing* where the swing point is set as the intersection of the upper and lower boundary slopes. *Sim-piece* [32] is the state-of-the-art PLA-based approach, optimizing a representation by grouping line segments to achieve better compression. There are also some machine learning based lossy compression methods [26, 28, 43, 49] that work by training predictors to store difference between actual values and predicted values. These methods have drawbacks like many model parameters needing extra storage and a need for retraining across domains, as well as higher computational costs. Overall, in scenarios requiring high precision, lossy compression algorithms face challenges when having to achieve high precision, high compression ratios and low compression times simultaneously.

Camel bridges the gap between lossy and lossless compression. Thus, *Camel* is lossless when time series data have low decimal place counts, while it achieves high precision, high compression ratios and low compression times simultaneously when time series data has high decimal place counts. Compared with algorithms such as *ELF* and *Chimp* that use XOR operations using the previous values, *Camel* improves the compression ratio and efficiency by identifying a better value for XOR operations.

8 Conclusion

This paper presents *Camel*, an efficient compression and query algorithm for floating-point time series. *Camel* compresses the integer and decimal parts of the floating points in time series by exploiting the characteristics of time series data. It finds a new value for each XOR operations instead of using the previous value, thereby enhancing the stability of the compression ratio when compressing the decimal parts. Additionally, *Camel* includes an effective index structure for compressed data, supporting four core query types. We compare *Camel* with 17 state-of-the-art compression algorithms, including 6 floating-point and 5 general compression algorithms, and 6 lossy compression algorithms on 22 public datasets and 3 industrial datasets from Alibaba Cloud. The findings provide evidence that *Camel* is capable of outstanding compression performance. The proposed index enables low memory usage and high query performance. In future research, it is of interest to extended *Camel* to distributed environments.

References

[1] 2016. Zstandard - Fast real-time compression algorithm. https://github.com/facebook/zstd. 2024.03.09.

[2] 2024. 2D wind speed and direction. https://data.neonscience.org/data-products/DP1.00001.001/RELEASE-2022. 2024.03.09.

[3] 2024. Barometric pressure. https://data.neonscience.org/data-products/DP1.00004.001/RELEASE-2022. 2024.03.09.

[4] 2024. Blockchair Bitcoin Transactions. https://gz.blockchair.com/bitcoin/transactions/. 2024.03.09.

[5] 2024. Daily Temperature of Major Cities. https://www.kaggle.com/datasets/sudalairajkumar/daily-temperature-of-major-cities. 2024.03.09.

[6] 2024. Dust and particulate size distribution. https://data.neonscience.org/data-products/DP1.00017.001/RELEASE-2022. 2024.03.09.

[7] 2024. Electric Vehicle Charging Dataset. https://www.kaggle.com/datasets/michaelbryantds/electric-vehicle-charging-dataset. 2024.03.09.

[8] 2024. Financial data set used in INFORE project. https://zenodo.org/record/3886895#.Y4DdzHZByM_. 2024.03.09.

[9] 2024. Global Food Prices Database (WFP). https://data.humdata.org/dataset/wfp-food-prices. 2024.03.09.

[10] 2024. Historical Weather Data. https://www.meteoblue.com/en/weather/archive/export/basel_switzerland. 2024.03.09.

[11] 2024. InfluxDB 2.0 Sample Data. https://github.com/influxdata/influxdb2-sample-data. 2024.03.09.

[12] 2024. IR biological temperature. https://data.neonscience.org/data-products/DP1.00005.001/RELEASE-2022. 2024.03.09.

[13] 2024. LZ4 - Extremely fast compression. https://github.com/lz4/lz4. 2024.03.09.

[14] 2024. Points of Interest POI Database. https://www.kaggle.com/datasets/ehallmar/points-of-interest-poi-database. 2024.03.09.

[15] 2024. Quantization Algorithms. https://intellabs.github.io/distiller/algo_quantization.html. 2024.06.04.

[16] 2024. Relative humidity above water on-buoy. https://data.neonscience.org/data-products/DP1.20271.001/RELEASE-2022. 2024.03.09.

[17] 2024. Snappy - About A fast compressor/decompressor. https://github.com/google/snappy. 2024.03.09.

[18] 2024. SSD and HDD Benchmarks. https://www.kaggle.com/datasets/alanjo/ssd-and-hdd-benchmarks. 2024.03.09.

[19] 2024. World Cities of different countries. https://www.kaggle.com/datasets/kuntalmaity/world-city. 2024.03.09.

[20] 2024. Xz - The .xz File Format. https://tukaani.org/xz/xz-file-format.txt. 2024.03.09.

[21] Jyrki Alakuijala, Andrea Farruggia, Paolo Ferragina, Eugene Kliuchnikov, Robert Obryk, Zoltan Szabadka, and Lode Vandevenne. 2019. Brotli: A General-Purpose Data Compressor. *ACM Trans. Inf. Syst.* 37, 1, 4:1–4:30.

[22] Yanzhe An, Yue Su, Yuqing Zhu, and Jianmin Wang. 2022. TVStore: Automatically Bounding Time Series Storage via Time-Varying Compression. In *FAST*. USENIX Association, 83–100.

[23] Davis W. Blalock, Samuel Madden, and John V. Guttag. 2018. Sprintz: Time Series Compression for the Internet of Things. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 2, 3 (2018), 93:1–93:23.

[24] Martin Burtscher and Paruj Ratanaworabhan. 2007. High Throughput Compression of Double-Precision Floating-Point Data. In *DCC*. 293–302.

[25] Martin Burtscher and Paruj Ratanaworabhan. 2009. FPC: A High-Speed Compressor for Double-Precision Floating-Point Data. *IEEE Trans. Computers* 58, 1, 18–31.

[26] Shubham Chandak, Kedar Tatwawadi, Chengtao Wen, Lingyun Wang, Juan Aparicio Ojea, and Tsachy Weissman. 2020. LFZip: Lossy Compression of Multivariate Floating-Point Time Series Data via Improved Prediction. In *DCC*. 342–351.

[27] Antonios Deligiannakis, Yannis Kotidis, and Nick Roussopoulos. 2007. Dissemination of compressed historical information in sensor networks. *VLDB J.* 16, 4, 439–461.

[28] Sheng Di and Franck Cappello. 2016. Fast Error-Bounded Lossy HPC Data Compression with SZ. In *IPDPS*. 730–739.

[29] Hazem Elmeleegy, Ahmed K. Elmagarmid, Emmanuel Cecchet, Walid G. Aref, and Willy Zwaenepoel. 2009. Online Piece-wise Linear Approximation of Numerical Streams with Precision Guarantees. *Proc. VLDB Endow.* 2, 1, 145–156.

[30] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. 2018. ModelarDB: Modular Model-Based Time Series Management with Spark and Cassandra. *Proc. VLDB Endow.* 11, 11, 1688–1701.

[31] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. 2021. Scalable Model-Based Management of Correlated Dimensional Time Series in ModelarDB+. In *ICDE*. 1380–1391.

[32] Xenophon Kitsios, Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. 2023. Sim-Piece: Highly Accurate Piecewise Linear Approximation through Similar Segment Merging. *Proc. VLDB Endow.* 16, 8, 1910–1922.

[33] Iosif Lazaridis and Sharad Mehrotra. 2003. Capturing Sensor-Generated Time Series with Quality Guarantees. In *ICDE*. 429–440.

[34] Ruiyuan Li, Zheng Li, Yi Wu, Chao Chen, Songtao Guo, Ming Zhang, and Yu Zheng. 2023. Erasing-based lossless compression method for streaming floating-point time series. *CoRR* abs/2306.16053.

[35] Ruiyuan Li, Zheng Li, Yi Wu, Chao Chen, and Yu Zheng. 2023. Elf: Erasing-based Lossless Floating-Point Compression. *Proc. VLDB Endow.* 16, 7, 1763–1776.

[36] Tianyi Li, Lu Chen, Christian S Jensen, and Torben Bach Pedersen. 2021. TRACE: Real-time compression of streaming trajectories in road networks. *Proc. VLDB Endow* 14, 7, 1175–1187.

[37] Tianyi Li, Ruikai Huang, Lu Chen, Christian S Jensen, and Torben Bach Pedersen. 2020. Compression of uncertain trajectories in road networks. *Proc. VLDB Endow* 13, 7, 1050–1063.

[38] Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. 2022. Chimp: Efficient Lossless Floating Point Compression for Time Series Databases. *Proc. VLDB Endow.* 15, 11, 3058–3070.

[39] Peter Lindstrom. 2014. Fixed-Rate Compressed Floating-Point Arrays. *IEEE Trans. Vis. Comput. Graph.* 20, 12, 2674–2683.

[40] Tong Liu, Jinzhen Wang, Qing Liu, Shakeel Alibhai, Tao Lu, and Xubin He. 2023. High-Ratio Lossy Compression: Exploring the Autoencoder to Compress Scientific Data. *IEEE Trans. Big Data* 9, 1, 22–36.

[41] Tuomas Pelkonen, Scott Franklin, Paul Cavallaro, Qi Huang, Justin Meza, Justin Teller, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, In-Memory Time Series Database. *Proc. VLDB Endow.* 8, 12, 1816–1827.

[42] Paruj Ratanaworabhan, Jian Ke, and Martin Burtscher. 2006. Fast Lossless Compression of Scientific Floating-Point Data. In *DCC*. 133–142.

[43] Dingwen Tao, Sheng Di, Zizhong Chen, and Franck Cappello. 2017. Significantly Improving Lossy Compression for Scientific Data Sets Based on Multidimensional Prediction and Error-Controlled Quantization. In *IPDPS*. 1129–1139.

[44] Qing Xie, Chaoyi Pang, Xiaofang Zhou, Xiangliang Zhang, and Ke Deng. 2014. Maximum error-bounded Piecewise Linear Representation for online stream approximation. *VLDB J.* 23, 6, 915–937.

[45] Yunlong Xu, Peizhen Yang, and Zhengbin Tao. 2023. Dangoron: Network Construction on Large-scale Time Series Data across Sliding Windows. In *SIGMOD*. 269–271.

[46] Yuanyuan Yao, Dimeng Li, Hailiang Jie, Lu Chen, Tianyi Li, Jie Chen, Jiaqi Wang, Feifei Li, and Yunjun Gao. 2023. SimpleTS: An Efficient and Universal Model Selection Framework for Time Series Forecasting. *Proc. VLDB Endow.* 16, 12, 3741–3753.

[47] Xinyang Yu, Yanqing Peng, Feifei Li, Sheng Wang, Xiaowei Shen, Huijun Mai, and Yue Xie. 2020. Two-Level Data Compression using Machine Learning in Time Series Database. In *ICDE*. 1333–1344.

[48] Xianyuan Zhan, Haoran Xu, Yue Zhang, Xiangyu Zhu, Honglei Yin, and Yu Zheng. 2022. DeepThermal: Combustion Optimization for Thermal Power Generating Units Using Offline Reinforcement Learning. In *AAAI*. 4680–4688.

[49] Kai Zhao, Sheng Di, Maxim Dmitriev, Thierry-Laurent D. Tonellot, Zizhong Chen, and Franck Cappello. 2021. Optimizing Error-Bounded Lossy Compression for Scientific Data by Dynamic Spline Interpolation. In *ICDE*. 1643–1654.

[50] Kai Zhao, Sheng Di, Danny Perez, Xin Liang, Zizhong Chen, and Franck Cappello. 2022. MDZ: An Efficient Error-bounded Lossy Compressor for Molecular Dynamics. In *ICDE*. 27–40.

Table 4: Dataset Statistics

Dataset			#Records	l	Timespan
Public	TS	City-temp (CT) [5]	2,905,887	1	25 years
		IR-bio-temp (IR) [12]	380,817,839	2	7 years
		Wind-speed (WS) [2]	199,570,396	2	6 years
		PM10-dust (PM10) [6]	222,911	3	5 years
		Stocks-UK (SUK) [8]	115,146,731	1	1 year
		Stocks-USA (SUSA) [8]	374,428,996	2	1 year
		Stocks-DE (SDE) [8]	45,403,710	3	1 year
		Dew point-temp (DT) [16]	5,413,914	2	3 years
		Air-pressure (AP) [3]	137,721,453	5	6 years
		Basel-wind (BW) [10]	124,079	7	14 years
		Basel-temp (BT) [10]	124,079	9	14 years
		Bitcoin-price (BP) [11]	2,741	4	1 month
		Bird-migration (BM) [11]	17,964	5	1 year
		Air-sensor (AS) [11]	8,664	17	1 hour
	Non-TS	Food-price (FP) [9]	2,050,638	4	-
		Vehicle-charge (VC) [7]	3,395	2	-
		Blockchain-tr (BTR) [4]	231,031	4	-
		SD-bench (SB) [18]	8,927	1	-
		City-lat (CLT) [19]	41,001	4	-
		City-lon (CLN) [19]	41,001	4	-
		POI-lat (PLT) [14]	424,205	16	-
		POI-lon (PLN) [14]	424,205	16	-
Industrial	TS	Mysql.cpu_usage (CPU)	92,212	8	1 week
		Mysql.disk_usage (DISK)	92,212	8	1 week
		Mysql.men_usage (MEM)	92,212	8	1 week

A Data statistics

Our datasets consists of 17 time series data and 8 none time series data. The properties of the data used in our experiments are listed in Table 4 and are thoroughly discussed below:

- **Time series dataset:**
  - **City-temp (CT)**: Dataset by the University of Dayton with temperatures of major world cities.
  - NEON datasets, provided by the National Science Foundation's National Ecological Observatory Network (NEON), include **Wind-speed (WS)**, **IR-bio-temp (IR)**, **PM10-dust (PM10)**, Air-pressure **(AP)**, and **Dew point-temp (DT)**. These datasets report values for wind speed, surface temperature, PM10 in the atmosphere, barometric pressure corrected to sea and surface levels, and relative dew point temperature, respectively.
  - Stock Exchange Datasets contains stock exchange price data used in INFORE project including UK stocks (i.e., **Stocks-UK (SUK)**), USA stocks (i.e., **Stocks-USA (SUSA)**) and German stocks (i.e., **Stocks-DE (SDE)**)
  - Meteoblue datasets provides **Basel-wind (BW)** and **Basel-temp (BT)** to report wind-speed data points and temperature measurements for Basel, Switzerland historical weather.
  - InfluxDB datasets are available by InfuxDB, providing **Bitcoin-price (BP)**, **Bird-migration (BM)** and **Air-sensor (AS)** to report Bitcoin-US dollar exchange rate data from the CoinDesk API, animal migratory movements throughout 2019, sample air sensor data.
  - Mysql metric datasets contains three metrics including **Mysql.cpu_usage (CPU)**, **Mysql.disk_usage (DISK)** and **Mysql.mem_usage (MEM)** to report the values of each metric over a week.
- **None time series dataset:**
  - **Food-prices (FP)**: Global Food Prices data from the World Food Programme for December 2020.
  - **Vehicle-charge (VC)**: Electric vehicle charging sessions between November 2014 and October 2015.
  - **Blockchain-tr (BTR)**: Bitcoin transaction values for a single day (2022-03-26).
  - **SDbench (SB)**: SSD & HDD benchmark scores.
  - **Citylat (CLT)** / **Citylon (CLN)**: Points of Latitude and longitude derived from world cities.
  - **POIlat (PLT)** / **POIlon (PLN)**: Points of interest derived from parsing Wikipedia.

B Proofs of reference discovery

B.1 Proof of Theorem 3.1

Theorem 3.1. The difference between $BF(v)$ and $BF(dxor)$ starts at position b and ends at position e, where $b \geq 1$ and $e \leq l$. Therefore, the center bits of $IE(v) \oplus IE(dxor)$ (i.e., $\hat{vd}$) are $T' = T_b, T_{b+1} \dots T_e$, where $T_b = 1, T_e = 1$, and the length of T' is $cn(\hat{vd}) = e - b + 1$.

Proof. Given $v.d^{dec} = (d_1 d_2 \cdots d_l)$ and $v.b^{dec} = (b_1 b_2 \cdots b_{l'})$, $v.d^{dec} = \sum_1^{l'} (b_l \times 2^{-l})$ based on Eq. 1. We deduce that if b_i in $v.b^{dec}$ is 1, then b_i in $v.d^{dec} - 2^{-i}$ is 0. By doing this, we convert b_i in $v.b^{dec}$ from 1 to 0. We set $T = \lfloor v.d^{dec}/2^{-l} \rfloor$ in Eq. 3. Then we have:

$$T_i = \begin{cases} 0 & \text{if } i = 0 \\ \left\lfloor (T - \sum_{j=0}^{j-1} T_j \times 2^{l-j})/2^{l-i} \right\rfloor & \text{if } 0 < i < l \end{cases} \tag{5}$$

Here, $T = \lfloor v.d^{dec}/2^{-l} \rfloor$ and $T_i (1 \leq i \leq l) \in \{0, 1\}$. Next, $dxor.d^{dec}$ can also be represented as:

$$\begin{aligned} dxor.d^{dec} &= v.d^{dec} - 2^{-l} \times (\left\lfloor v.d^{dec}/2^{-l} \right\rfloor) \\ &= v.d^{dec} - 2^{-l} \times T \\ &= v.d^{dec} - 2^{-l} \times (T_1 \times 2^{l-1} + \dots + T_i \times 2^{l-i} + T_l \times 2^0) \\ &= v.d^{dec} - \sum_{i=1}^{l} (T_i \times 2^{-i}) \end{aligned} \tag{6}$$

Here, the T_i in $\sum_{i=1}^{l} (T_i \times 2^{-i})$ is same as b_j in Eq. 1. Therefore, if $T_i = 0$, element b_i in $BF(dxor.d^{dec}) = 0$. The difference between $BF(v)$ and $BF(dxor)$ is the series $T' = T_b, T_{b+1} \dots T_e$, where $b \geq 1$ and $e \leq l$ and $T_b = T_e = 1$. T' represents the central bits. □

B.2 Proof of Theorem 3.2

Theorem 3.2. For any double precision floating point value $v = (d^{int}.d_1^{dec} d_2^{dec} \dots d_l^{dec})_{10}$, the integer part $v.d^{int} \neq 0$. Suppose the number of bits in $v.b^{int}$ is p and the center bits of $IE(v) \oplus IE(dxor)$ are T' ($T' = T_b, T_{b+1} \dots T_e$). Then the number of leading zeros of $\hat{vd}$ (i.e., $IE(v) \oplus IE(dxor)$) is leading_zeros $= 12 + (p - 1) + (b - 1)$.

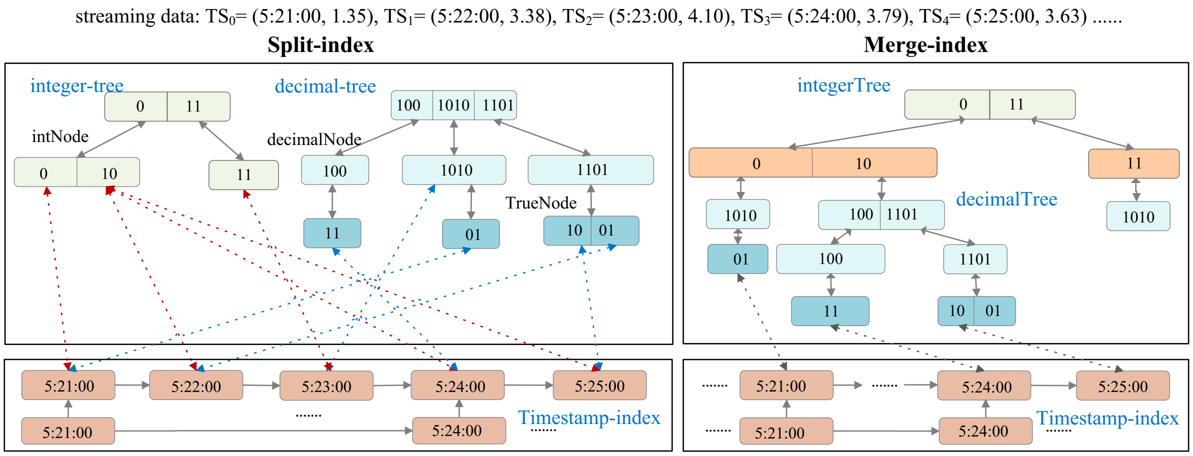


Figure 9: Index building and Query Processing in a City Temperature datasets

Proof. For any non-zero double precision floating point value $v = (v.d^{int}.d_1, d_2, d_3 \ldots d_l)_{10}$, where $v.d^{int} > 0$, the binary representation of $v.d^{int}$ always starts with 1 in the most significant bit (left-most bit). Letting p represent the number of binary bits in $v.b^{int}$, we can convert $v.b^{int}.v.b^{dec}$ to $1.m_1m_2 \ldots m_{52}$ by shifting the decimal point of $v.b^{int}$ to the left $p-1$ times during the conversion process to the IEEE 754 format. For instance, $BF(12.375) = (110.011)$ can be converted to 1.10011×2^2 by moving the decimal point to left twice (10011 represents $\vec{m}$ in the IEEE 754 format). Therefore, the integer parts of v and $dxor$ ensure identical binary representations, and their first $p-1$ bits of $\vec{m}$ are the same in the IEEE 754 format. This implies that the first $12+(p-1)$ elements of $IE(v)$ and $IE(dxor)$ are identical, i.e., the first $12+(p-1)$ elements of $IE(v) \oplus IE(dxor)$ are guaranteed to be 0. □

C An Running Example of *Camel* index

C.1 Index Construction

The *Camel* index is dynamic, which comprises three components, each fulfilling a unique role. These components include an integer tree, a decimal tree, and a timestamp index. Fig. 9 gives a running example of index building on 5 streaming time series data points with decimal place count $l = 2$: 1.35, 3.38, 4.12, 3.79, and 3.63. When inserting the TS_3 = (5:23:00, 3.79) in a streaming time series, *Camel* flows the four steps.

① **Compressing** v_2. For the integer part (i.e., 3), we first compute a secondary difference derived by subtracting the initial value from the current value and encode the binary representation of this secondary difference, e.g., $BF(3-1)$. Next, we apply *Camel* compression to the decimal part (i.e., 0.79), extracting l, $dxor$, and $centerBits$. Here, $l = 2$, $centerBits = 11$, $dxor = 100$.

② **Inserting into integer tree.** We insert the binary representation of difference (i.e., $BF(3-1) = 10$) obtained in step 1 into the integer B$^+$ tree. For the split-index, if the binary representation of difference exists in the integer-tree, we directly add a pointer to 5:23:00 in *tList*; otherwise, we create a new leaf entry in integer B$^+$ tree with *key* equaling to 10 and a timestamp list *tList* containing a pointer to 5:23:00. For the Merge-index, if the binary representation of difference does not exist in the integer-tree, we create a decimal tree, and add a leaf entry in integer tree with *key* equaling to 10 and a pointer to the root node of created decimal tree; otherwise, we do not need to do any operations in the integer tree.

③ **Inserting into decimal tree.** We first search $dxor = 100$ in the integer tree in the decimal index construction involves a meticulously designed B$^+$ tree with a capacity of $2^{-l} \times 10^l$ elements. Each leaf node gains an extra layer with $flag$ sub-nodes (1,0), followed by nodes for $dxor$, e.g., $100 \rightarrow (1, 11)$.

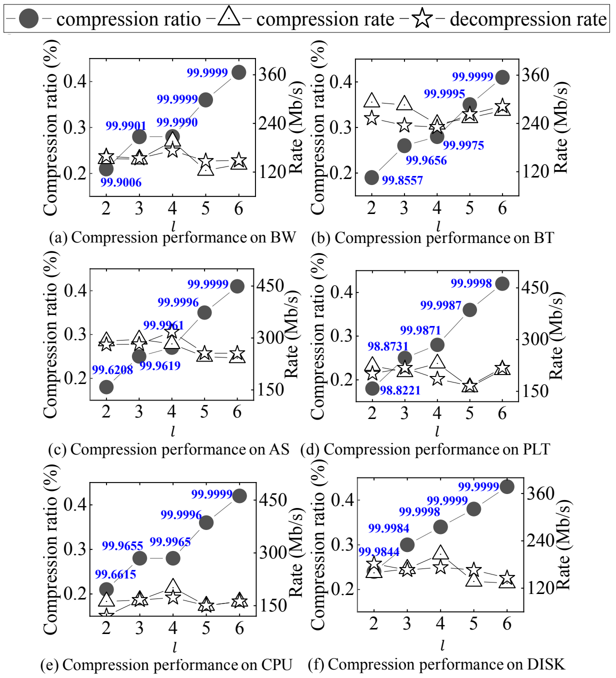
④ **Inserting into timestamp index.** We first search the nearest started position via skip list. In this example, for 5:23:00, we obtain 5:21:00 as the start poison via the skip list. After that, we search the timestamp list to find the exact position to insert 5:23:00.

C.2 Query Processing

We illustrate the detailed query processing using the index example depicted in Fig. 9.

i) Value Query: When given an integer or decimal value, we search the respective tree directly from its root node. For querying a floating-point number, two methods are available: split-index and merge-index. For example, when querying 3.38: (1) Split-index first searches the integer tree for the binary representation (i.e., 10) of 2 (i.e., 3 − 1) to get timestamps {5:22:00, 5:24:00, 5:25:00}. It then searches for the compression bits of 0.38 to get timestamp {5:22:00}. It finally calculates their intersection to get the result, {5:22:00}. (2) Merge-index first locates the decimal tree based on the integer part (i.e., the tree pointed to by 10) and then searches within it using the decimal part, finally obtaining the result, {5:22:00}.

ii) Range Query: Given a range (3.38, 3.79), (1) Split-index first performs a query based on the integer part, 3, of the range values to get timestamps {5:22:00, 5:24:00, 5:25:00}. Then, we refine the query based on the decimal part range (0.38, 0.79) of the range values to get the timestamps {5:22:00, 5:24:00, 5:25:00}. We finally calculate their intersection to get the result, {5:22:00, 5:24:00, 5:25:00}. If the query range spans different integer parts, we split it into multiple sub-ranges with the same integer part and then apply the same process to find the union of the results. For example, query range



(a) Compression performance on BW (b) Compression performance on BT

(c) Compression performance on AS (d) Compression performance on PLT

(e) Compression performance on CPU (f) Compression performance on DISK

Figure 10: Effect of the decimal count *l*

(1.35, 3.38) can be split into (1.35, 1.99), 2 and (3.00, 3.38). (2) Merge-index uses integer values to identify the decimal tree (i.e., pointed by 10) and then searches in the range of decimal values. It finally gets the result, {5:22:00, 5:24:00, 5:25:00}.

iii) Timestamp Query: Given a timestamp 5:22:00, we first locate it in the timestamp list via skiplist. (1) For Split-index, we find the corresponding leaf nodes in both the integer tree and the decimal tree pointed to by 5:22:00. Then, we can derive the integer part (i.e., 3) and the decimal part (i.e., 0.38) to get the result 3.38. (2) For Merge-index, we find the leaf node in the decimal tree pointed to by 5:22:00, and then we search its ancestor nodes to derive the final result, 3.38.

iv) Segment Query: Assume the current given dataset (with 5 points) is a stored in one block, we traverse the tree to get all the timestamps. Note that, in the streaming setting, to avoid memory consumption from expired data, we build a index for each block. (1) For Split-index, we only need to traverse the integer tree to get the all the the timestamps, {5:21:00, 5:22:00, 5:23:00, 5:24:00, 5:25:00}. (2) For Merge-index, we traverse the full tree to obtain the results.

D Additional Experiments

D.1 Effect of the decimal place count *l*

We evaluate the impact of the decimal count l on the compression performance across 6 datasets, comprising 4 TS datasets (BW, BT, CPU, and DISK) and 2 non-TS datasets (AS and PLA). Fig. 10 reports the corresponding results (including compression ratio, compression and decompression time, and precision) when varying l from 2 to 6. The digits in Fig. 10 represent precision. It is evident that increasing the precision leads to a decline in compression efficiency, attributed to the inevitable sacrifice of storage space for higher precision. Specifically, setting the decimal digits for PLA to 5 achieves a compression ratio of 36%, outperforming the other methods. However, increasing this to 6 significantly increases the space usage without proportional benefits. Finally, there is no consistent correlation between the number of decimal digits and the compression and decompression times; runtime fluctuations remain within a reasonable range.

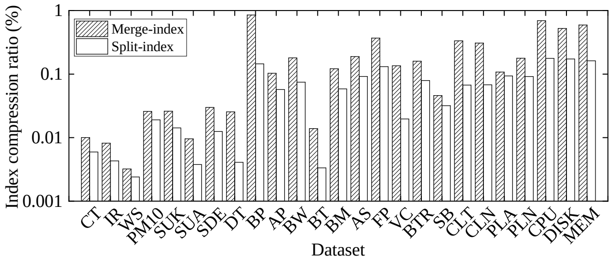
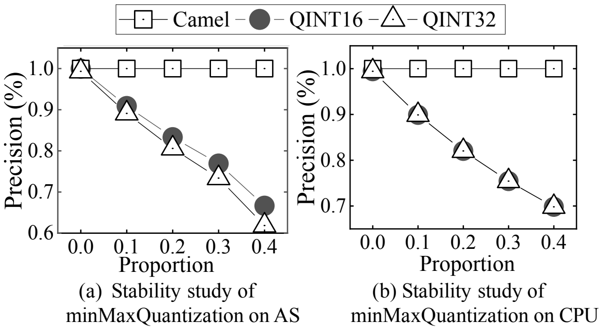


Figure 11: Compression ratio of *Camel*'s two tree indexes



(a) Stability study of minMaxQuantization on AS

(b) Stability study of minMaxQuantization on CPU

Figure 12: Stability study of minMaxQuantization.

D.2 Comparison on the two compressed index

We compare the proportions of two distinct index sizes constructed across 25 datasets, as illustrated in Fig. 11. We measure index performance using two metrics: index ratio, which compares the index size to the com- pressed data size. The merge-index represents a combination of the integer and decimal trees, whereas the split-index separates these two trees. Consequently, the merge-index is consistently larger than the split-index across all datasets. In detail, the proportion of the merge-index ranges between 0.01 and 0.5, with the exception of the BP dataset that exhibits frequent integer fluctuations, with fluctuations ranging from -145 to 513610 from each block's initial value. As a result, storing larger data necessitates the use of 64-bit storage. However, during compression, calculations involve differences relative to the preceding value, which typically have a narrower fluctuation range and do not require 64 bits. This leads to a relatively larger proportion of index sizes relative to the compressed data sizes for the BP dataset. Moreover, the split-index proportions are predominantly between 0.001 and 0.1 except BP, FP, and three industrial datasets. The large range of integer parts results in notably larger integer trees, thereby amplifying the overall index size.

D.3 Stability study of min-max Quantization

The min-max Quantization method shows significant advantages in general lossy compression methods, but it also has notable drawbacks. Firstly, it requires knowledge of the dataset's maximum and minimum values, making it unsuitable for streaming applications. Secondly, it demands a very balanced data distribution; if the distribution is uneven, accuracy can drop drastically. To validate it, we inject 10%, 20%, 30%, and 40% of repeated zero values into AS and CPU datasets, deliberately disrupting their distribution. Fig. 12 presents the precision of QINT16, QINT 32 and Camel on AS and CPU datasaets.

As shown in Fig. 12, both min-max Quantization Int16 (QINT16) and min-max Quantization Int32 (QINT32) exhibit a sharp decline in precision as the injection ratio increases. This is because, the skewed data distribution causes the quantization intervals to widen, resulting in low decompression precision. In contrast, our *Camel* maintains a stable precision of around 99.9999%. This stability is due to *Camel*'s lossy compression being dependent solely on the data values themselves instead of the data distribution. Therefore, *Camel* can maintain higher compression precision for uneven data distributions, particularly when encountering anomalies such as machine downtime or silent periods.