Serf: Streaming Error-Bounded Floating-Point Compression

RUIYUAN LI and ZECHAO CHEN, Chongqing University, China and Start Lab, China
RUYUN LU and XIAOLONG XU, Chongqing University, China and Start Lab, China
GUANGCHAO YANG, Chongqing University, China and Start Lab, China
CHAO CHEN*, Chongqing University, China
JIE BAO and YU ZHENG, JD iCity, China and JD Intelligent Cities Research, China

In IoT (Internet of Things) scenarios, massive floating-point time series data are generated in a streaming manner and transmitted within limited bandwidth for real-time analysis. To enhance the efficiency, it is acknowledged to compress the data before transmission. Existing floating-point compression methods are either for batched compression that may cause long delays, or for streaming lossless compression that has an unsatisfactory compression ratio when certain errors are allowed. In this paper, we propose the first Streaming ERror-bounded Floating-point compression *Serf*, which has two implementations: *Serf-Qt* and *Serf-XOR*. *Serf-Qt* first quantizes each floating-point value into an integer, and then encodes the integer with Elias gamma coding. *Serf-XOR* is the first lossy floating-point compression based on the XORing operation. To enhance the compression ratio of *Serf-XOR*, we propose a novel data offset technique to increase the leading zeros of the XORed values, and design a novel approximation technique to search for an error-qualified value that produces an XORed value with many trailing zeros. To improve the compression efficiency, we propose a pruning strategy to accelerate the process of approximated values search. We further build a streaming transmission prototype system based on a real development board, and deploy the proposed methods to it. Extensive experiments using 13 datasets show that, compared with 17 competitors, both *Serf-Qt* and *Serf-XOR* enjoy remarkable compression ratios with high efficiency in streaming scenarios. The transmission experiments based on the proposed system also showcase that *Serf-XOR* always takes the least overall time when the bandwidth is limited.

CCS Concepts: • **Information systems → Data compression**.

Additional Key Words and Phrases: Floating-Point Compression, Lossy Compression, Error-Bounded Compression, Streaming Compression

ACM Reference Format:
Ruiyuan Li, Zechao Chen, Ruyun Lu, Xiaolong Xu, Guangchao Yang, Chao Chen, Jie Bao, and Yu Zheng. 2025. *Serf*: Streaming Error-Bounded Floating-Point Compression. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 216 (June 2025), 27 pages. https://doi.org/10.1145/3725353

1 Introduction

Recent years have witnessed the advancement of IoT (Internet of Things) [52], where sensors are distributed at various locations over a region of interest. As shown in Fig. 1, in a typical IoT scenario,

*Corresponding author

Authors' Contact Information: Ruiyuan Li; Zechao Chen, Chongqing University, China and Start Lab, China, ruiyuan.li@cqu.edu.cn, thatcher@cqu.edu.cn; Ruyun Lu; Xiaolong Xu, Chongqing University, China and Start Lab, China, ruyun.lu@cqu.edu.cn, xiaolong.xu@stu.cqu.edu.cn; Guangchao Yang, Chongqing University, China and Start Lab, China, gchao_yang@cqu.edu.cn; Chao Chen, Chongqing University, China, cschaochen@cqu.edu.cn; Jie Bao; Yu Zheng, JD iCity, China and JD Intelligent Cities Research, China, baojie@jd.com, msyuzheng@outlook.com.

ACM 2836-6573/2025/6-ART216
https://doi.org/10.1145/3725353

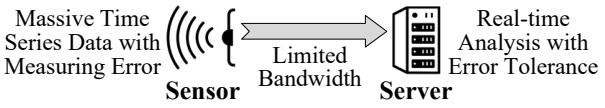


Fig. 1. Scenario of Streaming Error-Bounded Floating-Point Compression.

massive floating-point time series data (abbr. time series) are generated constantly by a wide range of sensors in a *streaming* fashion, and then transmitted to servers for further real-time analysis and visualization [27, 28]. It has the following characteristics. 1) **Limited Bandwidth**. The volume of time series is sheer, but the wireless network bandwidth between sensors and servers is usually limited. For example, Zigbee [71], a widely used wireless network communication protocol, has a rate of only 20∼250 kbit/s. To accelerate the transmission process and reduce the bandwidth costs, it is universally acknowledged to compress the data before transmitting them. 2) **Error Tolerance**. On the one hand, sensors themselves commonly have certain measuring errors. For example, civil GPS (Global Position System) usually has a measuring error of 5∼10 meters. On the other hand, some applications do not require exact readings from sensors. For instance, in daily life, we only care about the temperature readings with at most two decimal places. Hence, it is allowed to compress time series with a specified error bound. 3) **Real-Time Requirement**. Many critical applications depend on real-time data analysis, which means that each record should be sent to the server as early as possible after it is generated. For example, in a vehicle tracking system [65], the position information of vehicles should be reported to the server in a real-time manner, thereby enabling timely taxi dispatching [56], effective school bus monitoring [33], real-time path planning [57], etc.

There are numerous time series compression algorithms, which can be divided into three main categories: 1) batched lossless compression, 2) batched lossy compression, and 3) streaming lossless compression (note that in this paper, streaming compression means that we compress each record and output the compressed bits immediately after it is generated, as formally defined in Definition 3).

Batched lossless compression algorithms (e.g., ALP [8]) can compress time series without any information loss, while **batched lossy compression algorithms** (e.g., SZ [16], MOST [66] and Machete [61]) can usually achieve a better compression ratio at the sacrifice of some errors. These batched compression algorithms may suffer from three limitations. 1) *Long Latency When the Sampling Rate is Low.* They need to wait for all records in a batch to be ready before actually performing the compression operation. If the data sampling rate is low, batched compression will lead to long latency, which does not meet the real-time requirement described above. For example, GPS sensors usually report a position every 1∼10 seconds [35, 58]. If the batch size is 1,000, the latency will be as high as 1.7∼16.7 minutes. In the school bus monitoring scenario [33], it might be too late to take action when we realize that the bus has deviated from the planned route (maybe the school bus has been hijacked). For taxi dispatching [56], long latency will cause a poor user experience. 2) *Poor Performance if the Batch Size is Small.* To reduce the latency, an alternative solution is to adopt the mini-batch technique, e.g., the batch size is set to less than 50. However, the performance of most batched compression algorithms will deteriorate dramatically in mini-batches, since they have difficulty in discovering the shared information among records (cf. Section 7). 3) *Additional Memory Consumption for Caching the Records in a Batch.* Batched compression algorithms need to cache all records in a batch in memory, which might impose certain pressure on edge devices, since tiny sensor devices typically have a limited memory size that is less than 256KB [44]. For example, our development board Dofly CC2530 has only 8KB memory (cf. Section 6). Please note that the limited memory should also support the execution of some programs, such as compressors and transport modules.

Streaming lossless compression algorithms can compress each record immediately once it is generated while retaining all information. Several state-of-the-art methods in this line, such as Gorilla [55], Chimp [39] and Elf [37], are based on the XORing operation over consecutive records, because the consecutive records in a time series usually do not vary significantly, thus producing XORed values with many leading zeros and/or trailing zeros. Due to the lossless constraint, however, these algorithms usually achieve unsatisfactory compression ratios when certain errors are allowed.

To the best of our knowledge, there is no **streaming lossy compression algorithm** for time series yet. Streaming lossy compression algorithms not only address all the limitations encountered by the batched algorithms, but also achieve remarkable compression ratios compared to the lossless ones. Designing an efficient and compact streaming lossy compression algorithm will face the following two challenges. **Challenge I**: *How to achieve an excellent compression ratio with error bound guarantee in the streaming environments*? Streaming processing requires us to process each record once it is generated, making it hard to discover the shared information among records. **Challenge II**: *How to ensure high efficiency under the limited resources in sensors*? Note that the computing and storage resources of sensor devices are rather precious. Therefore, we must process the data efficiently to avoid piling up.

This paper proposes *Serf*, the first Streaming ERror-bounded Floating-point compression as far as we know. Specifically, we first propose a basic method *Serf-Qt* based on the quantization technique, where each record is first quantized into an integer, and then encoded with Elias gamma coding [18]. However, *Serf-Qt* only supports absolute error bound, and may perform poorly when the error bound is small. Therefore, we further propose an XORing-based method *Serf-XOR*. **To address Challenge I**, *Serf-XOR* first adds an offset to each record, which significantly increases the number of leading zeros in the XORed results. Then, *Serf-XOR* finds an approximated value that satisfies both conditions: 1) the difference between the approximated value and the original one is within the error bound, and 2) the current approximated value shares many "tail" bits with the previous approximated value, thus producing an XORed value with many trailing zeros. **To address Challenge II**, we propose a pruning strategy to avoid unnecessary computations during the search for the approximated values, which enhances the efficiency tremendously. *Serf-XOR* supports both absolute error bound and relative error bound, and usually enjoys a better compression ratio than *Serf-Qt* when the error bound is small.

Overall, the main contributions of this paper are as follows:

(1) We propose the first streaming error-bounded floating-point compression *Serf*. It consists of the quantization-based *Serf-Qt* and the XORing-based *Serf-XOR*, both of which enjoy remarkable compression ratios with high efficiency in streaming scenarios.

(2) For *Serf-XOR*, we propose a novel data offset technique to increase the leading zeros of the XORed values, where the optimal offset can be calculated with rigorous theoretical guidance.

(3) We propose a novel approximation technique to find an error-qualified approximated value that shares the most "tail" bits with the previous approximated value, thereby producing an XORed value with many trailing zeros. We further propose a pruning strategy to accelerate the process of approximated values search.

(4) We compare *Serf* with 17 state-of-the-art competitors using 13 time series from different sources, showing that *Serf* achieves the best compression ratio in streaming scenarios. We build a streaming transmission prototype system based on a real development board Dofly CC2530, and deploy multiple streaming compression algorithms, proving that *Serf* always takes the least total time (compression time + transmission time + decompression time) when the bandwidth is limited.

In the rest of the paper, we discuss the related works in Section 2, and give some preliminaries in Section 3. We introduce *Serf-Qt* and *Serf-XOR* in Section 4 and Section 5, respectively. We build a

Table 1. Comparison of Different Compression Algorithms.

Streaming/Batched	Accuracy	Technique	Compression Ratio	Examples
Batched	Lossless	Dictionary	Medium	LZ77 [70], Zstd [15], Snappy [24]
		Quantization	Medium	ALP [8]
	Lossy	Prediction/Quantization	High	SZ series [16, 40, 42, 62, 69], SZ_ADT [46], Machete [61]
		Fitting	High	DWT [59], Swing [19], Slide [19], Sim-Piece [32], MOST [66]
		Splitting	High	Buff [45]
Streaming	Lossless	Dictionary	Low	Deflate [53], LZ4 [10]
		Prediction/Quantization	Low	FPC [12]
		XORing	Medium	Gorilla [55], Chimp [39], Elf [37]
	Lossy	Quantization	High	*Serf-Qt* (this paper)
		XORing	Highest	*Serf-XOR* (this paper)

streaming data transmission prototype system and provide some extension discussion in Section 6. The experimental results are presented in Section 7, followed by the conclusion in Section 8.

2 Related Works

Existing time series compression algorithms can be divided into batched algorithms and streaming algorithms. Batched algorithms first divide the data into blocks, and then compress each block in a batch. Streaming algorithms compress each record once it is generated, and output the compressed bits immediately. These algorithms can be further categorized into lossless and lossy methods depending on whether the data information is fully retained. Table 1 shows the characteristics of the compression algorithms in each category.

2.1 Batched Lossless Algorithms

There are two types of batched lossless algorithms: general compression algorithms and floating-point-oriented compression algorithms. 1) Batched general lossless compression algorithms include LZ77 [70], Zstd [15], Snappy [24], etc. LZ77 uses a sliding window to search for duplicate substrings to compress the data in the window. Zstd combines a dictionary-matching stage with a fast entropy-coding stage, where the dictionary is trainable. Snappy resorts to a dictionary and stores the shift from the current position back to the uncompressed stream. 2) ALP [8] is an advanced floating-point-oriented batched lossless compression algorithm. If a value is originated from decimals, ALP first transforms it into an integer, and then encodes the integer losslessly. Otherwise, ALP uses vectorized compression on the floating-point values' front bits. Since floating-point-oriented compression algorithms take the advantage of the characteristics of time series, they usually achieve better performance than the general algorithms.

However, due to the lossless requirement, this type of compression algorithms may encounter the bottleneck in terms of compression ratio. Besides, the batched compression algorithms need to wait for the arrival of all data in a batch before compression, which introduces a long delay. To this end, they are not applicable to the streaming scenarios requiring immediate responses.

2.2 Batched Lossy Algorithms

Batched lossy algorithms can be divided into quantization-based algorithms, splitting-based algorithms, and fitting-based algorithms. 1) Representative quantization-based algorithms SZ series [16, 41, 42, 62, 68] take the prediction-quantization-encoding framework as the core, and use various predictors and encoders to achieve good performance. Based on the basic framework of SZ, SZ_ADT [46] introduces an adaptive data transcoding scheme to map quantization factors to codes, and uses finite state entropy to compress the codes. Machete [61] simplifies the prediction model and optimizes the encoder, thus achieving better performance in mini-batches. Instead of controlling the errors on raw data, the work [29] focuses on the data usability in the downstream analysis based on the quantization technique. By quantizing the floating points into

integers, Sprintz [11] can also achieve error-bounded floating-point compression. It first employs a forecaster to predict each sample, then bit packs the errors as a "payload" and prepends a header with sufficient information, and finally encodes the headers and payloads with Huffman coding. 2) The splitting-based algorithm Buff [45] first splits floating-point numbers into integer parts and decimal parts, and then compresses them respectively. If the maximum decimal precision is provided, Buff can act in a lossless way [14]. 3) Fitting-based methods use some functions to fit time series data. Traditional transformation methods such as DFT (Discrete Fourier Transforms) and DWT (Discrete Wavelet Transforms) belong to this line. For example, the work [20] utilizes DFT to extract features from time series and builds an efficient time series subsequence-matching algorithm, while these works [13, 59] employ DWT to extract the features. However, these techniques do not provide absolute error guarantees for a compressed point [17]. PAA [31] breaks the time series into segments, and uses the average value of each segment to represent the segment. Based on PAA, SAX [43] discretizes the coefficients using the Gaussian distribution. Instead of using the average value, PLA [60] and PAA [22] fit each segment using a linear function and a polynomial function, respectively. The work [17] learns multiple polynomial functions of different degrees to fit the time series data. Built upon PLA, Sim-Piece [32] finds the minimum number of segment groups and representing them jointly, while MOST [66] proposes to detect outliers before segmentation. HIRE [9] constructs a synopsis data structure with a recursion of partitioning and residualizing steps, where the partitioning step approximates a time series with a piecewise approximation, and the residualizing step calculates a signal representing the approximation error. HIRE can achieve effective multiresolution compression. Although the work [54] claims to summarize time series data in a streaming way, it is in fact a batched method according to our definition, because it is based on PLA and outputs the compressed data in batches. Similarly, ModelarDB [25] proposes an online model-agnostic compression algorithm with latency guarantees. It tries to fit the records with a model, and outputs the compressed data only if the model cannot fit the upcoming record or the latency reaches a specified value. Therefore, ModelarDB [25] is in practice a batched method.

Although achieving a better compression ratio than batched lossless compression algorithms, these batched lossy compression algorithms still cannot be adapted to streaming scenarios. Besides, most of these algorithms do not provide relative error guarantees.

2.3 Streaming Lossless Algorithms

Streaming compression algorithms compress the floating-point values one by one. Early streaming lossless compression algorithms include Deflate [53], LZ4 [10] and FPC [12]. Deflate [53] takes the basic ideas of LZ77 [70], but introduces more complex compression techniques such as Huffman coding. Specifically, Deflate uses sliding windows and Huffman coding to identify repeated patterns and frequent symbols during data stream generation, achieving efficient data compression. However, Deflate needs to maintain large compression tables and Huffman trees, leading to potentially high memory usage. The core idea of LZ4 [10] is to rapidly identify the duplicate segments in a stream using a sliding window and a hash table when the data arrives, making it suitable for scenarios with high compression speed requirements. Since LZ4 uses a simple compression mode that relies only on repeated segments, it may not be ideal for scenarios with high compression ratio demands. FPC [12] uses hash prediction to store the four-bit encoding, and then records the XORed result of residual, which has a relatively poor compression ratio. In recent years, more advanced lossless streaming compression algorithms have appeared, such as Gorilla [55], Chimp [39], Elf [37] and *SElf** [36]. They first perform an XORing operation on two consecutive values, and then encode the XORed results carefully. Their differences lie in the efforts to increase the trailing zeros, and the ways to encode the XORed results.

Table 2. Symbols and Their Meanings.

Symbols	Meanings
$TS = \langle v_1, v_2, ... \rangle$, $TS' = \langle v'_1, v'_2, ... \rangle$	Original floating-point time series, and its decompressed time series
s, $\vec{e} = \langle e_1, e_2, ..., e_{11} \rangle$, $\vec{m} = \langle m_1, m_2, ..., m_{52} \rangle$	Sign bit, exponent bits, mantissa bits of a double value, where $s, e_i, m_j \in \{0, 1\}$
ϵ_a, ϵ_r, ϵ	Absolute error bound, relative error bound, and error bound (can be either ϵ_a or $\epsilon_r \times \lvert v_i \rvert$)
q_i, p_i, out, in, bn	Quantized value of v_i, predicted value of v_i, stream writer, stream reader, number of bits in a value
$[v]$, $\lVert v \rVert$, $v_i \oplus v_j$, $v_i \vdash v_j$	Round operation, truncation operation, XORing operation, concatenation operation
s_i, a_i, λ, u	Shifted value, approximated value, offset, minimum integer making $[2^u, 2^{u+1})$ contain all shifted values
pre_j, suf_j, $lead_i$, $trail_i$, $rule$	The first/last j bits in prefix/suffix anchor, approximated # leading/trailing zeros, approximation rule

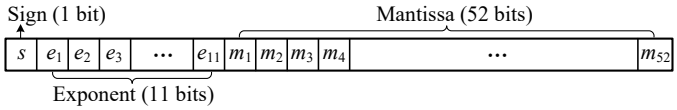


Fig. 2. Underlying Layout of Double Values.

All of these algorithms aim at lossless compression, resulting in an unsatisfactory compression ratio in the scenarios where certain errors are allowed. TRACE [38] is a framework for real-time compression of streaming trajectories, but it does not apply to generic floating-point data. As far as we know, there is still no any research work for streaming lossy floating-point compression. Therefore, in this paper, we propose the first streaming error-bounded floating-point compression *Serf* that has two implementations: *Serf-Qt* (based on the quantization technique) and *Serf-XOR* (based on the XORing operation), both of which can achieve a remarkable compression ratio with high efficiency in streaming scenarios.

3 Preliminaries

In this section, we first introduce the floating-point layout, and then provide some definitions. Table 2 shows the symbols used frequently throughout this paper.

3.1 Floating-Point Layout

A floating-point value v can be double precision (i.e., double value), single precision (i.e., single value) or half precision, etc. In this paper, if not specified, the value refers to a double value, since double values are the most versatile. In accordance with IEEE 754 Standard [30], a double value has a layout occupying 64 bits shown in Fig. 2, which consists of the 1-bit sign s, the 11-bit exponent $\vec{e} = \langle e_1, e_2, ..., e_{11} \rangle$, and the 52-bit mantissa $\vec{m} = \langle m_1, m_2, ..., m_{52} \rangle$. In particular, if all 64 bits are zeros, $v = 0$. Otherwise, we have[1]:

$$v = (-1)^s \times 2^{\sum_{i=1}^{11} e_i \times 2^{11-i} - 1023} \times (1 + \sum_{i=1}^{52} m_i \times 2^{-i}) \tag{1}$$

[1]In fact this is only true for normal numbers. Special numbers [30, 37] of double values (which are rather uncommon) are out of the discussion scope of this paper.

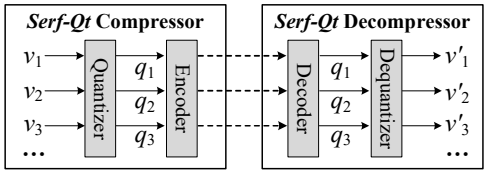


Fig. 3. Framework of *Serf-Qt*.

Given two values $v_1 > 0$ and $v_2 > 0$, 1) if $\exists k \in [1, 11)$, $\forall j \in [1, k]$, $v_1.e_j = v_2.e_j$ and $v_1.e_{k+1} > v_2.e_{k+1}$, then $v_1 > v_2$; 2) if $\forall i \in [1, 11]$, $v_1.e_i = v_2.e_i$, but $\exists k \in [1, 52)$, $\forall j \in [1, k]$, $v_1.m_j = v_2.m_j$ and $v_1.m_{k+1} > v_2.m_{k+1}$, then $v_1 > v_2$.

3.2 Definitions

Definition 1 (Time Series[2]). *A time series $TS = \langle v_1, v_2, ...\rangle$ refers to a sequence of floating-point values arranged in chronological order.*

Definition 2 (Absolute/Relative Error-Bounded Lossy Compression). *Suppose the original time series is $TS = \langle v_1, v_2, ...\rangle$, and its decompressed time series is $TS' = \langle v'_1, v'_2, ...\rangle$. The absolute error-bounded lossy compression satisfies that $\forall i > 0$, $|v_i - v'_i| \leq \epsilon_a$, where $\epsilon_a > 0$ is a user-specified parameter. The relative error-bounded lossy compression satisfies that, $\forall i > 0$, if $v_i = 0$, then $v'_i = 0$; if $v_i \neq 0$, then $|(v_i - v'_i)/v_i| \leq \epsilon_r$, where $\epsilon_r > 0$ is specified by users.*

Definition 3 (Streaming Error-Bounded Lossy Compression). *Suppose the original time series is $TS = \langle v_1, v_2, ...\rangle$, and its corresponding decompressed time series is $TS' = \langle v'_1, v'_2, ...\rangle$. $\forall i > 0$, v'_i and v_i satisfy the error bound ϵ_a or ϵ_r defined in Definition 2. During compression, we compress v_i and output the compressed bits b_i immediately once v_i is obtained. During decompression, we recover v'_i from b_i immediately without waiting for any other bits.*

4 Basic Method *Serf-Qt*

Quantization is widely used for lossy floating-point compression. In this section, we propose a basic quantization-based streaming method *Serf-Qt*, the framework of which is shown in Fig. 3.

4.1 *Quantizer* and *Dequantizer*

The intuition of quantization is converting the floating-point values into integers, since integers can be compressed more easily than floating-point values. Suppose the original values are $\langle v_1, v_2, ...\rangle$ and the quantized integers are $\langle q_1, q_2, ...\rangle$, *Quantizer* performs:

$$q_i = [\frac{v_i - p_i}{2 \times \epsilon_a}] \tag{2}$$

where $[*]$ is an operation that rounds a floating-point value to an integer, ϵ_a is the absolute error bound given by users, and p_i is the predicted value of v_i (see the next paragraph).

Given q_i, *Dequantizer* calculates the recovered value v'_i by:

$$v'_i = 2 \times \epsilon_a \times q_i + p_i \tag{3}$$

It can be proved that $|v_i - v'_i| \leq \epsilon_a$, i.e., the recovered value v'_i strictly satisfies the absolute error-bound constraint [61]. Because two consecutive values in a time series usually vary little, we can

[2]Like Gorilla [55], Chimp [39], Elf [37] and ALP [8], we mainly focus on the compression of floating-point values in a univariate (i.e., one-dimensional) time series.

simply let $p_i = v'_{i-1}$ (note that we can calculate v'_{i-1} during both compression and decompression). In particular, $v'_0 = v_0 = 0$.

4.2 *Encoder* and *Decoder*

Existing quantization-based compression methods [16, 41, 61, 62, 68] can only work in batch mode, because they encode the quantized integers with batched strategies, e.g., Huffman coding [16, 61] or FSE (Finite State Entropy) coding [46]. To support streaming compression, we propose to replace the batched encoder with a streaming one. Specifically, since the difference between v_i and p_i is minor, i.e., q_i tends to be distributed around 0, we employ Elias gamma coding [18], a self-interpreting streaming encoding method that is suitable for compressing small values. To cater to the fact that Elias gamma coding can only encode positive integers, we first convert q_i into a positive integer based on Zigzag coding [23], the result of which is then encoded with Elias gamma coding, i.e.,

$$code_i = \text{EliasGammaEn}(\text{ZigzagEn}(q_i) + 1) \tag{4}$$

Here, $\text{ZigzagEn}(q_i) = (q_i << 1) \oplus (q_i >> 63)$ where $\oplus$ is the bitwise XORing operation. If $q_i = 0$, then $\text{ZigzagEn}(q_i) = 0$, so we need to add an additional 1 to ensure that the input of Elias gamma coding is positive. Suppose $N_i = \text{ZigzagEn}(q_i) + 1$, Elias gamma coding writes $\lfloor log_2 N_i \rfloor$ zeros, and then appends the binary digits of N_i. For example, if $q_i = 5$, then $N_i = \text{ZigzagEn}(q_i) + 1 = 11$, where $\lfloor log_2 N_i \rfloor = 3$ and the binary digits of N_i are $(1011)_2$. Therefore, $code_i = \text{EliasGammaEn}(N_i) = (000\ 1011)_2$.

Decoder is a reverse process of *Encoder*. It first decodes $code_i$ using Elias gamma decoding method, the result of which is then decoded with the Zigzag decoding method, i.e.,

$$q_i = \text{ZigzagDe}(\text{EliasGammaDe}(code_i) - 1) \tag{5}$$

Specifically, Elias gamma decoding first reads and counts the zeros from the binary digits of $code_i$ until reaching the first 1 (suppose the count of zeros is n_i), and then reads the remaining n_i digits. The first read 1 and the n_i digits combine the decoded result. For example, if $code_i = (000\ 1011)_2$, then the count n_i of zeros is 3, so we need to read 3 more bits in addition to the already read 1, forming $N_i = \text{EliasGammaDe}(code_i) = (1011)_2 = 11$. After that, we subtract 1 to offset the added 1 in Equ. (4). Suppose x is the input number of Zigzag decoding, then $\text{ZigzagDe}(x) = (x >>> 1) \oplus (-(x\&1))$.

4.3 Discussion

Serf-Qt is extremely fast, and its time complexity and space complexity are both $O(1)$ for a given value. However, *Serf-Qt* does not support relative error-bounded compression, since in Equ. (3), we need to know the error bound that depends on the original values for relative error-bounded compression, which is unavailable. Besides, when ϵ_a is small, *Serf-Qt* would perform poorly, because q_i tends to be large according to Equ. (2). In extreme circumstances, if ϵ_a is small enough, q_i may overflow (i.e., exceeding the range that an integer can represent), which makes *Serf-Qt* invalid.

5 XORing-Based Method *Serf-XOR*

To address the issues of *Serf-Qt*, we propose an XORing-based method *Serf-XOR*. As far as we know, this is the first lossy floating-point compression method based on the XORing operation.

5.1 Main Idea

Existing XORing-based floating-point compression methods [37, 39, 55] are based on the assumption: any two consecutive values in a time series do not vary significantly, so their XORed value tends to contain many **leading zeros** and **trailing zeros**. However, this assumption is not always true. First, if two consecutive values have different signs (e.g., -0.96 and 2.02 shown in Fig. 4(a)), their XORed

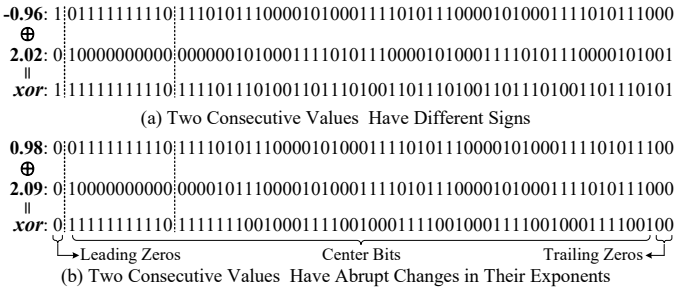


Fig. 4. Motivation of *Serf-XOR*.

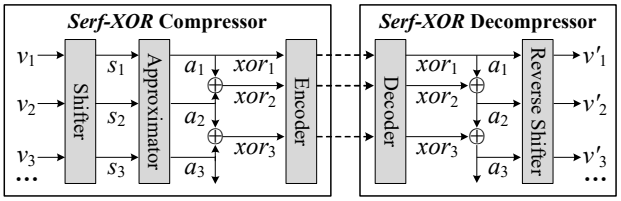


Fig. 5. Framework of *Serf-XOR*.

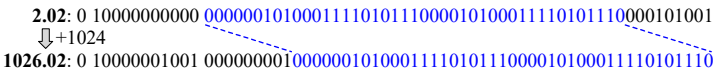


Fig. 6. Adding an Offset Leads to Information Loss.

value does not contain any leading zeros. Second, even if the two consecutive values have the same sign, if their exponent representations have abrupt changes (e.g., 0.98 and 2.09 shown in Fig. 4(b)), the leading zeros are still extremely few. Third, as pointed out by [39], there are in fact rather few trailing zeros in most cases (e.g., 0 in Fig. 4(a) and 2 in Fig. 4(b)). Few leading zeros and trailing zeros would result in poor compression performance. To this end, we propose a **shift technique** to increase the leading zeros, and an **approximation technique** to increase the trailing zeros.

Fig. 5 gives the framework of *Serf-XOR*, which contains two main parts: *Serf-XOR* Compressor and *Serf-XOR* Decompressor.

5.2 *Serf-XOR* Compressor

In *Serf-XOR* Compressor, the floating-point values $\langle v_1, v_2, v_3 \ldots \rangle$ sequentially flow into *Shifter*, obtaining a sequence of shifted values $\langle s_1, s_2, s_3 \ldots \rangle$ with the same sign and same exponent. After that, *Approximator* converts the shifted values one by one into a sequence of approximated values $\langle a_1, a_2, a_3 \ldots \rangle$. Each approximated value a_i (except for the first value a_1) performs an XORing operation with its previous value a_{i-1}, resulting in a sequence of XORed values $\langle xor_1, xor_2, xor_3 \ldots \rangle$ with both many leading zeros and trailing zeros, which can be compressed compactly with *Encoder*.

5.2.1 Shifter. *Shifter* adds an integer offset λ to each v_i, obtaining a shifted value $s_i = v_i + \lambda$ that has the same sign (i.e., positive) and the same exponent with other shifted values. Consequently, any two consecutive shifted values can produce an XORed value with many leading zeros. But how to determine an appropriate λ?

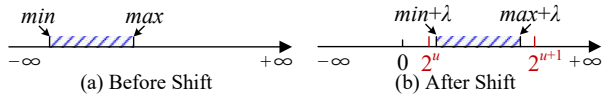


Fig. 7. Illustration of Shifter.

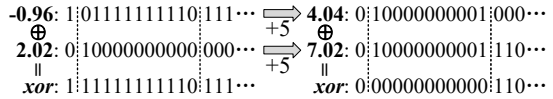


Fig. 8. Example of Shift.

On one hand, λ should be large enough to make all s_i positive and have the same exponent. On the other hand, as shown in Fig. 6, adding an offset may cause the mantissa to shift right, resulting in some information loss. A larger offset usually means more information loss. In extreme cases, if λ is too large, s_i may overflow. To this end, we let λ be the smallest value that makes all s_i fall in the range $[2^u, 2^{u+1})$, where u is a positive integer, as shown in Fig. 7. For the convenience of description, we give the following definition.

Definition 4 (Truncation). *Given a floating-point value v, the truncation operation is to truncate the decimal part of v, denoted as $\|v\|$. For example, $\|3.14\| = 3$, $\|-3.14\| = -3$.*

If $v \geq 0$, we have $\|v\| = \lfloor v \rfloor \leq v$. If $v < 0$, we have $\|v\| = \lceil v \rceil \geq v$. Besides, for any floating-point value v and any integer λ, we have $\lfloor v + \lambda \rfloor = \lfloor v \rfloor + \lambda$. We leverage Theorem 1 to find u and λ.

Theorem 1. *Given a time series $TS = \langle v_1, v_2, ... \rangle$, suppose $\forall v_i \in TS$, $min \leq v_i \leq max$. If $u = \lceil log_2(\lfloor max \rfloor - \lfloor min \rfloor + 1) \rceil$ and $\lambda = 2^u - \lfloor min \rfloor$, we can guarantee that all shifted values $s_i = v_i + \lambda$ are in the range $[2^u, 2^{u+1})$, and λ is the smallest at the same time.*

Proof. We first prove that $u = \lceil log_2(\lfloor max \rfloor - \lfloor min \rfloor + 1) \rceil$. Since $min \leq v_i \leq max$, we have $min + \lambda \leq s_i = v_i + \lambda \leq max + \lambda$. In accordance with the smallest requirement of λ and $s_i = v_i + \lambda \in [2^u, 2^{u+1})$, u should be the smallest as well. That is to say, the range $[2^u, 2^{u+1})$ should cover all the integer parts in $[min + \lambda, max + \lambda]$, but the range $[2^{u-1}, 2^u)$ should not. The number of integers in $[min + \lambda, max + \lambda]$ is $\|max + \lambda\| - \|min + \lambda\| + 1$, and there are $2^{u+1} - 2^u = 2^u$ integers and $2^u - 2^{u-1} = 2^{u-1}$ integers in $[2^u, 2^{u+1})$ and $[2^u, 2^{u-1})$, respectively. Therefore, we have:

$$\begin{aligned} 2^{u-1} < \|max + \lambda\| - \|min + \lambda\| + 1 \leq 2^u &\Leftrightarrow 2^{u-1} < \lfloor max + \lambda \rfloor - \lfloor min + \lambda \rfloor + 1 \leq 2^u \\ &\Leftrightarrow 2^{u-1} < \lfloor max \rfloor - \lfloor min \rfloor + 1 \leq 2^u \\ &\Leftrightarrow u - 1 < log_2(\lfloor max \rfloor - \lfloor min \rfloor + 1) \leq u \\ &\Leftrightarrow u = \lceil log_2(\lfloor max \rfloor - \lfloor min \rfloor + 1) \rceil \end{aligned}$$

We now prove that $\lambda = 2^u - \lfloor min \rfloor$ can make $s_i \in [2^u, 2^{u+1})$. On one hand, since $v_i \geq min$, we have $s_i = v_i + \lambda \geq min + (2^u - \lfloor min \rfloor) \geq 2^u$. On the other hand, since $u = \lceil log_2(\lfloor max \rfloor - \lfloor min \rfloor + 1) \rceil \geq log_2(\lfloor max \rfloor - \lfloor min \rfloor + 1)$ and $v_i \leq max$, we have:

$$2^u \geq \lfloor max \rfloor - \lfloor min \rfloor + 1 > max - \lfloor min \rfloor \Leftrightarrow 2^{u+1} > max + 2^u - \lfloor min \rfloor \geq v_i + \lambda = s_i$$

□

Example. Suppose $min = -1.0$ and $max = 2.5$. Hence, $u = \lceil log_2(\lfloor 2.5 \rfloor - \lfloor -1.0 \rfloor + 1) \rceil = 2$, and $\lambda = 2^2 - \lfloor -1.0 \rfloor = 5$. The original two values -0.96 and 2.02 have different signs and exponents, resulting in no leading zeros in their XORed result (i.e., the left part in Fig. 8). After adding $\lambda = 5$,

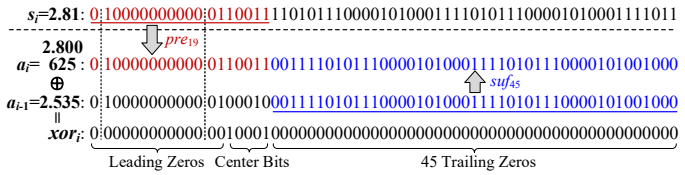


Fig. 9. Basic *Approximator* ($\epsilon = \epsilon_a = 0.01$).

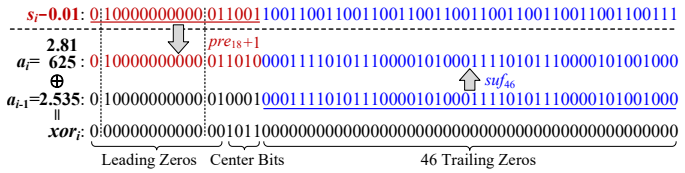


Fig. 10. Optimized Approximator ($\epsilon = \epsilon_a = 0.01$).

the two shifted values 4.04 and 7.02 share the same sign and same exponent, leading to an XORed result with as many as 12 leading zeros (i.e., the right part in Fig. 8).

The maximum and minimum values in a time series can be obtained by domain knowledge (e.g., the latitude of a GPS point is between -180 and 180, and sensors usually have a certain measuring range). They can also be set to the historical maximum and minimum values of a time series, respectively, and adjust over time in streaming scenarios. The maximum and minimum values in specific time ranges of a time series can be recorded in the meta data. Even if there might be some outlier values falling out of the recorded minimum and maximum range, it only harms the compression ratio of the outlier values, but imposes little effect on the overall compression performance since the outlier values are usually extremely few. The error bound for the outlier compression can be still guaranteed as we would check the output values during the approximated values search (see below).

5.2.2 Approximator. In this module, for each shifted value s_i, we try to find an approximated value $a_i \in [s_i - \epsilon, s_i + \epsilon]$, such that a_i and a_{i-1} share the most suffix bits, i.e., their XORed result has the most trailing zeros. Here, $\epsilon = \epsilon_a$ (i.e., based on an absolute error bound) or $\epsilon = \epsilon_r \times |v_i|$ (i.e., based on a relative error bound).

Fig. 9 shows the basic idea of *Approximator*, where $a_{i-1} = 2.535$, $s_i = 2.81$, and $\epsilon = \epsilon_a = 0.01$. To find $a_i \in [s_i - \epsilon, s_i + \epsilon]$, we can simply set the last 45 bits of s_i as these of a_{i-1}, obtaining $a_i = 2.800625$ (i.e., a_i is composed of the 19 prefix bits of s_i and the 45 suffix bits of a_{i-1}, denoted by $a_i = pre_{19} \vdash suf_{45}$. We call s_i **prefix anchor** and a_{i-1} **suffix anchor**). By XORing a_i and a_{i-1}, we can get an XORed value with both many leading zeros and trailing zeros. During decompression, since $a_i \in [s_i - \epsilon, s_i + \epsilon]$, we have $v_i' = a_i - \lambda \in [s_i - \lambda - \epsilon, s_i - \lambda + \epsilon]$, i.e., $v_i' \in [v_i - \epsilon, v_i + \epsilon]$, which satisfies the error bound requirement. But how can we find a qualified a_i efficiently?

One straightforward solution is to iteratively check whether $a_i = pre_{bn-j} \vdash suf_j$ satisfies the error bound requirement, where j is from bn to 0, and bn is the number of bits in a value (e.g., $bn = 64$ for double values). If so, we stop the verification, and return a_i. However, there might be two problems. **Problem I**: the obtained a_i might be suboptimal (i.e., it does not share the most suffix bits with a_{i-1}). For example, there exists another qualified $2.81625 \in [2.80, 2.82]$ shown in Fig. 10 that shares more suffix bits with a_{i-1} than 2.800625 shown in Fig. 9. **Problem II**: the enumeration of all suffixes suf_j from scratch might take much time, which hurts the compression efficiency.

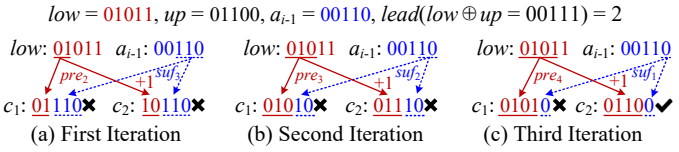


Fig. 11. Examples of Approximated Values Search (for simplicity, the number of bits bn in a value is 5).

To this end, this paper proposes a novel method to find the theoretically best-approximated value a_i in a more efficient way. To solve Problem I, we employ two optimizations. First, we let the prefix anchor be $low = s_i - \epsilon$ instead of s_i. Second, in addition to checking whether the concatenation $c_1 = pre_{bn-j} \vdash suf_j$ is in $[s_i - \epsilon, s_i + \epsilon]$, we also check if $c_2 = (pre_{bn-j} + 1) \vdash suf_j$ is in $[s_i - \epsilon, s_i + \epsilon]$. Fig. 11 shows three examples of concatenations. In Fig. 11(c), we find that $c_2 = (pre_4 + 1) \vdash suf_1$ satisfies the error bound requirement, so we set $a_i = c_2 = (01100)_2$ finally. The correctness of these two optimizations can be guaranteed by Theorem 2.

Theorem 2. *Suppose $low = s_i - \epsilon$ is the prefix anchor, and a_{i-1} is the suffix anchor. If we iteratively check both concatenations $c_1 = pre_{bn-j} \vdash suf_j$ and $c_2 = (pre_{bn-j} + 1) \vdash suf_j$, where j is from bn to 0, we can always obtain a qualified approximated value a_i that shares the most suffix bits with a_{i-1}.*

Proof. Because $low = s_i - \epsilon$ is the prefix anchor, given a specified suffix anchor suf_j, we have the following two observations: (1) $c_1 = pre_{bn-j} \vdash suf_j$ is the smallest possible qualified value, since any $(pre_{bn-j} - 1) \vdash suf_j$ must be smaller than low; (2) $c_2 = (pre_{bn-j} + 1) \vdash suf_j$ is the smallest value that is greater than c_1 and low.

There are three cases for c_1. **Case 1**: $c_1 < s_i - \epsilon$. In this case, we further check if $c_2 \le s_i + \epsilon$. There are still two subcases. **Case 1.1**: If $c_2 \le s_i + \epsilon$, we return c_2 as the final result. **Case 1.2**: If $c_2 > s_i + \epsilon$, since c_2 is the smallest value with the suffix suf_j that is greater than c_1, any other value with the suffix suf_j must be greater than $s_i + \epsilon$ as well. **Case 2**: $c_1 > s_i + \epsilon$. Because c_1 is the smallest possible qualified value with the suffix suf_j, any other value with the suffix suf_j would be greater than $s_i + \epsilon$ too. **Case 3**: $c_1 \in [s_i - \epsilon, s_i + \epsilon]$. It means that c_1 is the final result.

Overall, for Case 1.1 and Case 3, we can find a qualified value with suffix suf_j; for Case 1.2 and Case 2, we will not miss any possible qualified values. As we iterate j from bn to 0, we can always get a qualified a_i with the longest suffix suf_j. □

In Theorem 2, we may iterate j for $bn + 1$ times (i.e., from bn to 0). We find that it is in fact unnecessary to iterate j from scratch. Specifically, we can utilize Theorem 3 to solve Problem II.

Theorem 3. *Suppose $low = s_i - \epsilon$, $up = s_i + \epsilon$, and $l = lead(low \oplus up)$ is the number of leading zeros of $low \oplus up$. If we iterate j from $bn - l$ to 0, we can still obtain the best a_i.*

Proof. Since j is from $bn - l$ to 0, we have $k = bn - j$ from l to bn. Let $c_1 = pre_k \vdash suf_{bn-k}$ and $c_2 = (pre_k + 1) \vdash suf_{bn-k}$.

For any $k^* < l$, we have two observations: (1) for $c_1 = pre_{k^*} \vdash suf_{bn-k^*}$, if $c_1 \in [low, up]$, the first l bits of c_1 must be the same with low and up. In this case, c_1 is equal to $pre_l \vdash suf_{bn-l}$, i.e., we will not miss any qualified c_1 since we can still get it when $k = l$; (2) for $c_2 = (pre_{k^*} + 1) \vdash suf_{bn-k^*}$, it must be greater than up, i.e., there definitely does not exist any qualified c_2 when $k^* < l$. □

In accordance with Theorem 3, it is enough to iterate j from $bn - l$ to 0. Since ϵ is usually very small, the difference between low and up is commonly marginal, i.e., $l = lead(low \oplus up)$ is supposed to be large, which reduces the cost of iteration significantly. For example, as shown in Fig. 11, because $lead(low \oplus up) = 2$, we iterate j starting from $5 - 2 = 3$ instead of from 5.

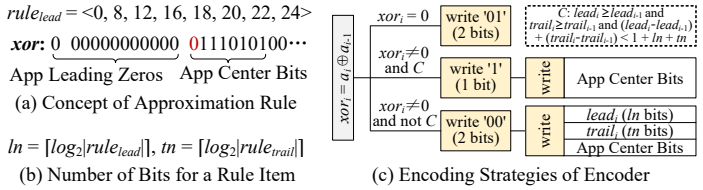


Fig. 12. Main Idea of Encoder.

Algorithm 1: $Compressor_{Serf-XOR}(v_i, a_{i-1}, \epsilon, rule, out)$

```
   /* Shifter                                                              */
1  s_i ← v_i + λ; // λ is defined in Theorem 1
   /* Approximator                                                         */
2  low ← s_i − ε; up ← s_i + ε; l ← lead(low ⊕ up); a_i ← s_i;
3  for j from bn − l to 0 do // bn is # bits in a value
4      pre_{bn−j} ← the prefix bn − j bits of low;
5      suf_j ← the suffix j bits of a_{i−1};
6      if (pre_{bn−j} ⊢ suf_j) ∈ [s_i − ε, s_i + ε] then
7          a_i ← pre_{bn−j} ⊢ suf_j; break;
8      if ((pre_{bn−j} + 1) ⊢ suf_j) ∈ [s_i − ε, s_i + ε] then
9          a_i ← ((pre_{bn−j} + 1) ⊢ suf_j); break;
   /* Encoder                                                              */
10 xor_i ← a_i ⊕ a_{i−1}; lead_i ← AppLead(xor_i, rule); trail_i ← AppTrail(xor_i, rule);
11 if xor_i = 0 then // Case 1
12     out.write('01');
13 else if Condition C holds then // Case 2
14     out.write('1'); out.write(App Center Bits);
15 else // Case 3
16     out.write('00'); out.write(lead_i); out.write(trail_i); out.write(App Center Bits);
17 Update and write rule if needed;
```

5.2.3 Encoder. To encode the XORed values, we adopt the adaptive encoding strategies [36], which encompasses a concept of approximation rule (a sorted integer array). As shown in Fig. 12(a), suppose the approximation rule for leading zeros is $rule_{lead} = \langle 0, 8, 12, 16, 18, 20, 22, 24 \rangle$, although the number of leading zeros is 13, we regard it as 12 (since $12 \leq 13 < 16$). Thus, the number of leading zeros can be represented by $ln = \lceil log_2 |rule_{lead}| \rceil = 3$ bits (called **representation cost**) instead of $\lceil log_2 65 \rceil = 7$ bits (the number of leading zeros can be 0~64). The 13^{th} leading zero is considered as a part of center bits, which costs one more bit (called **approximation cost**) for center bits. The work [36] proposes a dynamic programming-based method to find the best trade-off between the representation cost and approximation cost efficiently.

Fig. 12(c) shows the encoding strategies for $xor_i = a_i \oplus a_{i-1}$, which has three cases. **Case 1**: if $xor_i = 0$, it simply writes a flag '01' of 2 bits. **Case 2**: if $xor_i \neq 0$ and the condition C holds, it writes a flag '1' of 1 bit, followed by the approximated center bits. Here, C is "$lead_i \geq lead_{i-1}$ and $trail_i \geq trail_{i-1}$ and $(lead_i - lead_{i-1}) + (trail_i - trail_{i-1}) < 1 + ln + tn$", where $lead_i$ and $trail_i$ are the approximated numbers of leading zeros and trailing zeros in xor_i respectively, and ln and tn are defined in Fig. 12(b). **Case 3**: if $xor_i \neq 0$ and the condition C does not hold, it writes a flag '00'

Algorithm 2: $Decompressor_{Serf-XOR}(a_{i-1}, rule, in)$

```
   /* Decoder                                                              */
1  flag ← in.read(1); // First read 1 bit
2  if flag = 0 then // The flag contains 2 bits
3      flag ← flag ⊢ in.read(1); // Read another bit
4  if flag ='01' then // Case 1
5      lead_i ← lead_{i−1}; trail_i ← trail_{i−1}; xor_i ← 0;
6  else if flag ='1' then // Case 2
7      lead_i ← lead_{i−1}; trail_i ← trail_{i−1}; AppCenterBits ← in.read(bn − lead_i − trail_i);
8      xor_i ← (lead_i zeros)_2 ⊢ AppCenterBits ⊢ (trail_i zeros)_2;
9  else // Case 3
10     lead_i ← in.read(ln); trail_i ← in.read(tn); AppCenterBits ← in.read(bn − lead_i − trail_i);
11     xor_i ← (lead_i zeros)_2 ⊢ AppCenterBits ⊢ (trail_i zeros)_2;
12 a_i ← a_{i−1} ⊕ xor_i;
   /* Reverse Shifter                                                      */
13 v'_i ← a_i − λ; // λ is defined in Theorem 1
14 Read and update rule if needed;
15 return v'_i;
```

of 2 bits, $lead_i$ of ln bits, $trail_i$ of tn bits, and finally the approximated center bits. The flag codes are carefully selected according to the frequency of each case. For more details, please refer to [36].

5.2.4 Summary. Algorithm 1 depicts the pseudocode of *Serf-XOR* Compressor, which takes as input the current value v_i, the last approximated value a_{i-1} ($a_0 = 0$ in our implementation), the error bound ϵ, approximation rules *rule* for leading/trailing zeros and the stream writer *out*. Most codes are self-explanatory. In Line 17, we update and write *rule* at the end of each window (e.g., 1,000 values) to get a better approximation rule in the subsequent compression.

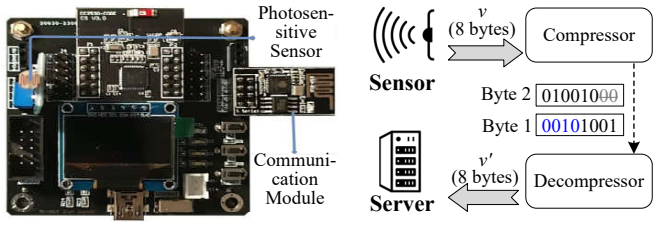
We iterate j for at most $bn - l$ times, so its time complexity is $O(bn - l)$. We need to store the statistics of leading/trailing zeros (for updating approximation rules), which contains no more than $2 \times bn$ records, so the space complexity of Algorithm 1 is $O(bn)$.

5.3 *Serf-XOR* Decompressor

As shown in the right part of Fig. 5, in *Serf-XOR* Decompressor, the compressed bit stream is fed into *Decoder* to get a sequence of XORed values $\langle xor_1, xor_2, xor_3 ... \rangle$, with which we can obtain the approximated values $\langle a_1, a_2, a_3 ... \rangle$ using the XORing operation. *Reverse Shifter* converts each a_i to a floating-point value v'_i. Note the decompressor is not a strict reverse procedure of the compressor.

As presented in Algorithm 2, *Serf-XOR* Decompressor takes as input the last approximated value a_{i-1}, the approximation rules *rule* and the stream reader *in*. Lines 1~12 are the codes of *Decoder*. Since the flag codes can be '1', '00' or '01', we first read one bit. If it is '0', we need to read one more bit to assemble $flag$. Subsequently, we handle the three cases separately based on the value of $flag$, and finally obtain the approximated value a_i in Line 12. In Line 13, *Reverse Shifter* computes the decompressed value $v'_i = a_i - \lambda$. Like *Serf-XOR* Compressor, we need to read and update *rule* at the end of each window for the subsequent decompression if necessary.

Since there is no any loop in Algorithm 2, its time complexity is $O(1)$. Furthermore, we only keep several states (e.g., $lead_{i-1}$ and $trail_{i-1}$) in memory, so the space complexity is $O(1)$.



(a) Development Board: Dofly CC2530 (b) Transmission Framework

Fig. 13. Transmission Prototype System.

6 Prototype System and Extension

In this section, we first build a streaming transmission prototype system, and then extend *Serf* to other floating-point precisions.

6.1 Transmission Prototype System

We build a streaming transmission prototype system based on a real development board, Dofly CC2530, which has a photosensitive sensor and a communication module, as shown in Fig. 13(a). The photosensitive sensor can generate a time series with a user-defined rate, while the communication module can send the time series to the server through the Zigbee protocol [71]. Dofly CC2530 is equipped with a 32-bit arm cortex-m4 @ 2.4GHz CPU and only 8KB memory. Therefore, it is crucial for these devices to perform the streaming floating-point compression to avoid memory overflow.

Fig. 13(b) presents the prototype framework, where the Sensor constantly generates a sequence of records, compresses the records one by one once they are generated, and sends the compressed bits through the network; the Server monitors an input stream, reads the received bytes, and decodes the bytes into values. As the network can only transmit data by bytes and we cannot distinguish the boundary between two records in the stream, we use 4 bits to indicate the number of bytes for decoding each value (it should be enough since the number of bytes of one compressed value would not be greater than $2^4 = 16$ bytes). For example, as shown in Fig. 13(b), the first 4 bits in Byte 1 indicate that we should read two bytes (including the 4 bits) for decoding v. The last two bits in Byte 2 just pad the byte and should be discarded directly.

6.2 Precision Extension

In addition to double precision ($bn = 64$), there are floating-point values of other precisions, such as single precision ($bn = 32$) and half precision ($bn = 16$). As deep learning advances, there also emerge other floating-point data types, such as Float16, Float32 and Tensor Float32. All of these data types have a similar underlying layout to double values, but with different bits for the exponent and mantissa. For example, for single precision, its exponent and mantissa occupy 8 bits and 23 bits, respectively. Our proposed algorithms, both *Serf-Qt* and *Serf-XOR*, can be easily extended to floating-point values with other precisions. In specific, for *Serf-Qt*, we should not make any modifications, since it has nothing to do with the precision. For *Serf-XOR*, in Algorithm 1 and Algorithm 2, it is enough to set bn as the number of bits of the target precision.

7 Experiments

7.1 Datasets and Experiment Settings

7.1.1 Datasets. To verify the performance, we adopt 13 time series datasets from various domains with different sampling intervals, the details of which are described in Table 3. We randomly extract

Table 3. Details of Datasets.

Dataset	Description	Range	Interval
Air-pressure (AP) [48]	Barometric air pressure	[96.29, 98.55]	1 min
Basel-temp (BT) [3]	Temperature of Basel	[-15.21, 38.49]	1 hour
Basel-wind (BW) [3]	Wind speed of Basel	[2.52, 102.24]	1 hour
Chengdu-traj (CDT)	Latitude of taxi trajectories	[30.65, 30.73]	3 sec
City-temp (CT) [1]	City Temperature	[-99, 100.2]	1 day
Dew-point (DT) [51]	Dew point temperature	[19.68, 100.45]	1 min
IR-bio-temp (IR) [50]	Infrared biotemperature	[-6.78, 21.98]	1 min
Motor-temp (MT) [5]	Temperature of motors	[19.04, 131.44]	0.5 sec
PM10-dust (PM10) [49]	Concentration of PM10	[0.001, 150]	30 min
Smart-grid (SG) [6]	Voltage in the grid system	[0, 2.05]	4.2 ms
Stocks-USA (SUSA) [2]	Stock prices of USA	[11.11, 152.92]	1 min
T-drive (TD) [67]	Longitude of taxi trajectories	[101.63, 122.37]	1 min
Wind-Speed (WS) [47]	Speed of winds	[0.01, 5.29]	2 min

up to 100,000 consecutive records from each dataset. We also utilize the photosensitive sensor data generated by our development board to test the transmission performance. Moreover, we employ the datasets in the time series benchmark [4] to verify the robustness of the proposed methods.

7.1.2 Baselines. We compare *Serf* with **17** representative competitors, including **4** batched lossless methods (i.e., LZ77 [70], Zstd [15], ALP [8] and Snappy [24]), **7** batched lossy methods (i.e., SZ2 [41], Machete [61], Sim-Piece [32], SZ_ADT [46], Sprintz [11], HIRE [9] and Buff [45]) and **6** streaming lossless methods (i.e., Deflate [53], LZ4 [10], FPC [12], Gorilla [55], Chimp_{128} [39] and Elf [37])[3]. For batched methods, we run them in mini-batches. Deflate and Zstd are set to the default compression level, while LZ77 is set to the compression level of 2 for a better compression ratio. Most source codes are written in C++ (except for Buff in Rust and HIRE in Python) with gcc 11.4.0, and released along with the datasets [7].

7.1.3 Metrics. The performance of various methods is verified by three main metrics: compression ratio, compression time and decompression time. Note that the compression ratio is defined as the ratio of the compressed data size to the original one.

7.1.4 Settings. Most experiments are conducted on a personal computer equipped with Ubuntu 22.04, i7-12500H CPU, 16GB RAM and 512GB SSD disk. The default absolute error bound ϵ_a and relative error bound ϵ_r are set 0.001 and 1%, relatively. In daily life, people usually care about at most two decimal places in a floating-point value, e.g., price of commodities and temperature readings, so setting $\epsilon_a = 0.001$ is enough to maintain the fidelity of the data. For batched competitors, we set the default batch size as 50, which makes it as fair as possible for streaming methods by limiting the latency and memory consumption of batched methods.

7.2 Overall Comparison with Baselines

In this set of experiments, we run each method on each dataset using an absolute error bound $\epsilon_a = 0.001$, and report the average performance on a batch of 50 data records. ALP [8] throws a segment fault when the batch size is smaller than 600, due to the vectorized execution and sampling requirements. We try to give up these two features, but find it still leads to a poor compression ratio (e.g., ≥ 2). SZ_ADT [46] also throws an exception when the batch size is less than 150. HIRE [9]

[3]MOST [66] does not provide the complete source codes. ModelarDB [26] is a database system having a poor compression ratio when the batch size is less than 10,000, so we do not compare it. We add a quantizer before Sprintz [11] to support error-bounded floating-point compression. SZ3 [42] performs worse than SZ2 [41] in our settings, so we select SZ2 instead of SZ3 as a baseline.

Table 4. Overall Comparison - Compression Ratio (The best results in batched and streaming groups are marked in bold, separately. The same for the tables bellow).

		Dataset	AP	BT	BW	CDT	CT	DT	IR	MT	PM10	SG	SUSA	TD	WS	Avg.
Batched	Lossless	LZ77	0.27	0.86	0.66	0.96	0.51	0.72	0.42	0.66	0.36	0.85	0.53	0.77	0.57	0.63
		Zstd	0.26	0.86	0.63	0.99	0.43	0.68	0.39	0.53	0.33	0.86	0.50	0.77	0.52	0.60
		Snappy	0.27	0.88	0.66	0.99	0.51	0.73	0.42	0.67	0.35	0.87	0.53	0.81	0.56	0.64
	Lossy	SZ2	0.44	0.89	0.68	0.95	0.57	0.76	0.48	0.64	0.45	0.49	0.53	0.51	0.63	0.62
		Machete	0.21	0.35	0.34	0.30	0.36	0.35	**0.24**	**0.28**	0.23	0.22	**0.26**	0.23	0.30	0.28
		Sim-Piece	**0.15**	0.44	0.40	0.34	0.37	0.42	0.26	0.35	**0.16**	**0.14**	0.31	**0.21**	0.34	0.30
		Sprintz	0.31	0.38	0.43	0.35	0.44	0.43	0.34	0.34	0.32	0.28	0.35	0.31	0.36	0.36
		Buff	0.20	**0.27**	**0.28**	**0.17**	**0.30**	**0.28**	0.25	**0.28**	0.30	0.20	0.30	0.25	**0.22**	**0.25**
Streaming	Lossless	Deflate	0.23	0.85	0.58	0.92	0.37	0.61	0.32	0.51	0.28	0.85	0.42	0.69	0.44	0.54
		LZ4	0.31	0.90	0.72	1.01	0.55	0.78	0.48	0.70	0.42	0.89	0.59	0.82	0.62	0.68
		FPC	0.26	0.91	0.84	0.80	0.76	0.83	0.54	0.76	0.53	0.65	0.66	0.59	0.87	0.69
		Gorilla	0.26	0.92	0.82	0.77	0.77	0.84	0.58	0.33	0.49	0.75	0.68	0.61	0.87	0.67
		$Chimp_{128}$	0.28	0.72	0.55	0.75	0.38	0.55	0.36	0.50	0.32	0.72	0.39	0.58	0.47	0.50
		Elf	0.14	0.53	0.48	0.34	0.23	0.27	0.16	0.36	0.14	0.26	0.18	0.26	0.24	0.28
	Lossy	Serf-Qt	0.07	0.30	0.32	0.16	0.32	0.30	0.13	0.15	0.10	**0.10**	0.16	0.09	0.21	0.18
		Serf-XOR	**0.06**	**0.23**	**0.23**	**0.12**	**0.22**	**0.22**	**0.12**	**0.14**	**0.09**	**0.10**	**0.14**	**0.08**	**0.16**	**0.15**

shows a poor compression ratio (e.g., ≥ 1) when the batch size is less than 128. Hence, we do not report the performance of the three methods in this set of experiments.

7.2.1 Compression Ratio. Table 4 shows the results of compression ratios, from which we have the following observations.

(1) Among all the methods, *Serf* always enjoys the best compression ratio on all datasets. For the batched lossless methods LZ77, Zstd and Snappy, since they are designed for general purpose in large batches, they usually perform worse than advanced floating-point-specialized compressors, let alone in mini-batches. Specifically, compared with the best batched lossless method Zstd, *Serf-XOR* has an average of 75% (i.e., $(0.60 - 0.15)/0.60 = 75\%$) relative improvement in terms of the compression ratio. For the batched lossy methods, as most of them are designed for large batches, their performance is not as good as *Serf* when the batch size is 50. Buff shows the best compression ratio (i.e., 0.25) in this line, but it can only support an error bound with a format of 10^{-k} and *Serf-XOR* still has an average of 40% relative improvement over it. Among the streaming lossless methods, Elf has the best compression ratio (i.e., 0.28) because it employs an erasing technique to increase the trailing zeros tremendously, but it is still much inferior to *Serf*.

(2) *Serf-XOR* has a slightly better compression ratio than *Serf-Qt* (i.e., 0.15 VS 0.18) when $\epsilon_a = 0.001$. *Serf-XOR* employs a set of optimization techniques including data offset and value approximation, all of which contribute to a better compression ratio.

7.2.2 Compression & Decompression Time. Table 5 and Table 6 exhibit the compression time and decompression time of all methods on different datasets, respectively.

(1) Compression Time. In each category, Snappy, Buff, Gorilla and *Serf-Qt* have the least compression time, respectively[4]. We mainly discuss the streaming methods here, since the batched methods cannot be applied to streaming scenarios. Gorilla takes an average compression time of $0.06\mu s$, because it only performs an XORing operation for each record without any complicated logic. Besides, Gorilla encompasses the fewest conditional branches compared with other XORing-based methods, which is beneficial for branch prediction, thus enhancing the efficiency. However, its compression ratio is as bad as 0.67, about 4.47× of *Serf-XOR*. Note that in the IoT data transmission

[4]The results may be inconsistent with those in the paper [45] due to the different experimental settings. In this paper, we aim at streaming compression, so the batch size is set to only 50. However, the paper [45] sets the batch size unlimited.

Table 5. Overall Comparison - Compression Time (μs).

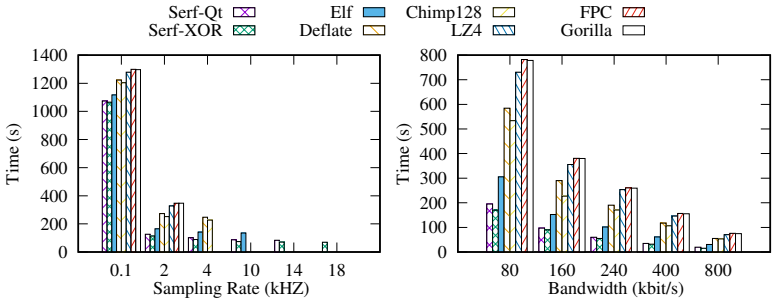
		Dataset	AP	BT	BW	CDT	CT	DT	IR	MT	PM10	SG	SUSA	TD	WS	Avg.
Batched	Lossless	LZ77	0.88	1.40	1.09	1.23	1.09	1.09	1.08	1.06	0.89	0.97	1.05	1.00	1.46	1.10
		Zstd	3.11	10.30	7.04	9.02	5.02	7.14	3.72	5.26	3.41	6.49	5.49	6.71	6.13	6.06
		Snappy	**0.03**	**0.10**	**0.21**	**0.06**	**0.04**	**0.08**	**0.02**	**0.06**	**0.07**	**0.01**	**0.11**	**0.05**	**0.04**	**0.07**
	Lossy	SZ2	20.17	44.54	89.66	26.39	203.05	32.93	17.60	19.67	29.05	16.53	18.99	15.42	23.58	42.89
		Machete	6.00	13.23	10.80	10.86	9.84	11.04	5.99	9.51	5.09	5.06	7.10	6.25	9.39	8.47
		Sim-Piece	7.47	14.39	18.91	17.34	12.06	14.79	10.53	12.34	6.07	5.03	12.20	7.47	12.59	11.63
		Sprintz	1.81	2.11	2.00	2.21	1.89	2.12	2.10	2.01	1.94	1.69	2.01	1.80	2.07	1.98
		Buff	7.10	7.50	8.22	7.25	7.31	7.53	6.69	8.49	7.69	9.38	7.84	4.23	6.92	7.40
Streaming	Lossless	Deflate	13.29	30.22	25.21	28.80	14.04	17.61	10.92	18.53	11.52	27.50	13.71	21.64	14.16	19.01
		LZ4	1.14	1.83	1.22	1.25	0.98	1.04	1.01	1.13	1.04	1.09	1.06	1.13	1.26	1.17
		FPC	**0.13**	0.05	**0.06**	0.19	0.09	**0.05**	0.10	0.16	0.11	0.06	0.07	0.03	0.16	0.10
		Gorilla	0.17	**0.02**	0.07	**0.03**	**0.03**	0.10	**0.02**	**0.06**	**0.01**	**0.02**	**0.06**	**0.01**	**0.12**	**0.06**
		$Chimp_{128}$	0.27	1.31	1.24	1.03	0.43	1.03	0.43	0.11	0.34	0.97	0.50	0.83	0.88	0.72
		Elf	6.92	6.06	6.15	6.56	4.78	5.09	4.75	5.42	4.63	6.00	4.75	4.79	5.37	5.48
	Lossy	Serf-Qt	1.30	1.37	1.12	1.18	1.02	1.05	0.68	1.06	0.32	0.49	1.18	0.58	1.03	0.95
		Serf-XOR	1.37	1.35	1.21	1.17	1.18	1.30	1.11	1.17	0.84	0.88	1.31	0.81	1.33	1.16

Table 6. Overall Comparison - Decompression Time (μs).

		Dataset	AP	BT	BW	CDT	CT	DT	IR	MT	PM10	SG	SUSA	TD	WS	Avg.
Batched	Lossless	LZ77	0.01	**0.02**	**0.02**	**0.01**	**0.03**	0.06	**0.02**	0.04	**0.02**	**0.01**	0.03	**0.02**	0.08	**0.03**
		Zstd	1.07	1.01	1.43	0.79	1.24	1.83	1.46	1.23	1.18	0.75	1.55	1.02	1.24	1.22
		Snappy	**0.002**	0.05	0.12	**0.01**	0.08	0.02	**0.02**	**0.01**	**0.02**	0.03	0.05	**0.02**	**0.02**	**0.03**
	Lossy	SZ2	7.76	3.41	6.26	4.48	6.00	5.38	4.85	4.55	6.56	3.96	5.56	5.40	5.65	5.37
		Machete	0.08	0.06	0.06	0.18	0.07	**0.01**	0.05	0.09	0.07	0.02	**0.01**	0.11	0.06	0.07
		Sim-Piece	2.13	3.12	3.57	2.66	3.17	3.18	2.44	2.82	1.53	1.28	2.36	1.86	2.64	2.52
		Sprintz	1.42	0.91	**0.02**	1.17	0.14	0.69	1.04	1.57	0.34	1.24	0.48	1.23	1.22	0.88
		Buff	3.92	5.14	4.18	5.21	4.73	4.55	2.61	4.88	4.40	5.11	4.79	2.19	3.68	4.26
Streaming	Lossless	Deflate	2.90	3.87	3.48	4.37	3.29	5.13	3.75	4.93	2.94	4.40	3.86	3.96	4.43	3.95
		LZ4	1.17	1.11	1.18	1.01	1.35	1.10	1.13	1.06	1.23	1.04	1.02	1.06	1.12	1.12
		FPC	1.22	1.15	1.19	1.22	1.16	1.24	1.07	1.34	1.08	1.25	1.19	1.12	1.21	1.19
		Gorilla	0.27	1.05	1.12	1.08	0.92	1.07	0.80	0.91	0.70	0.99	0.95	1.00	1.01	0.91
		$Chimp_{128}$	1.07	1.16	1.38	1.04	1.04	1.14	1.03	1.13	1.03	1.06	1.15	1.05	1.19	1.11
		Elf	**0.10**	1.03	1.11	0.82	0.83	1.07	0.59	**0.08**	0.62	0.77	1.04	0.76	1.06	0.76
	Lossy	Serf-Qt	0.75	1.30	1.58	1.21	1.69	1.51	0.61	0.96	0.34	0.38	1.02	0.40	1.24	1.00
		Serf-XOR	0.22	**0.11**	**0.13**	**0.09**	**0.15**	**0.11**	**0.09**	0.19	**0.05**	**0.08**	**0.07**	**0.06**	**0.07**	**0.11**

scenarios, bandwidth is usually the bottleneck. The most competitive streaming competitor in terms of the compression ratio, i.e., Elf, takes more compression time than both *Serf-Qt* and *Serf-XOR*, since the calculation of decimal significand count in Elf is relatively time-consuming. *Serf-XOR* takes slightly more compression time than *Serf-Qt*, as *Serf-XOR* performs more operations (data offset and approximated values search).

(2) Decompression Time. LZ77, Machete, Elf, and *Serf-XOR* exhibit the best decompression efficiency in each category, respectively. We here again focus on the streaming methods. *Serf-XOR* usually performs the best in terms of the decompression time among all the streaming methods. There could be two reasons. First, *Serf-XOR* has the best compression ratio among all methods, resulting in the fewest bits to read. Second, in addition to reading the flags, Elf also needs to read the significand counts and perform a relatively time-consuming recovery operation during decompression. *Serf-XOR* is better than *Serf-Qt*, because for *Serf-XOR* there are many cases where the XORed values are 0. In these cases, *Serf-XOR* only reads the flag bits but does not read other bits (i.e., Lines 4∼5 in Algorithm 2). *Serf-Qt*, however, reads multiple times when counting the number of zeros in the stream.



(a) Different Sampling Rates (b) Different Bandwidths

Fig. 14. Data Transmission Experiments.

(3) In comparison with compression time, the distinction of decompression time among different algorithms is less insignificant. Furthermore, in the IoT data transmission scenarios, the data are usually decompressed in the servers with powerful computing capabilities. That is, data decompression will not be the bottleneck. To this end, in the following experiments, we mainly focus on the compression ratio and compression time.

7.3 Streaming Transmission Experiments

To verify the overall performance of the proposed methods in the streaming data transmission scenarios, we conduct a set of experiments based on our proposed transmission prototype system. We do not compare with the batched compression methods here because they cannot act in a real streaming manner.

We set the sampling rate of the photosensitive sensor in Dofly CC2530 from 0.1kHZ to 18kHZ respectively, and observe the **total time** (including compression time, transmission time and decompression time) using different streaming compression methods when transmitting 100,000 records from the sensor to the server, based on the communication protocol of Zigbee. The observed bandwidth is about 180 kbit/s. As shown in Fig. 14(a), with different sampling rates, *Serf-XOR* always takes the least time, since it has the best compression ratio, contributing to the fewest bits to transmit. When the sampling rate gets larger, the time taken by all methods first drops sharply and then decreases slowly. This is because when the sampling rate is larger than 2kHZ, the bandwidth acts as the bottleneck, causing many records to be blocked in the send queue. When the sampling rate reaches 4kHZ, the methods of LZ4, FPC and Gorilla fail, as the blocked records lead to a memory overflow. If the sampling rate is 10kHZ, Deflate fails. Further, if the sampling rate is 14kHZ, Elf fails. The proposed *Serf-XOR* still works well until the sampling rate is as high as 20kHZ, which proves its robustness.

We also transmit all data shown in Table 3 sequentially from one server to another under different bandwidths. In this set of experiments, we employ the wired network, and the sampling rate and the memory size are not limited. As shown in Fig. 14(b), with the bandwidth increases, the time required for all methods decreases. However, their relative performance remains unchanged, and *Serf-XOR* always performs the best. For instance, when the bandwidth is 240 kbit/s, *Serf-XOR* takes only about 54 seconds, while the most competitive method Elf takes about 2× time (i.e., 101 seconds). *Serf-XOR* performs slightly better than *Serf-Qt* due to its relatively better compression ratio.

These two sets of experiments prove the feasibility and importance of the proposed methods in streaming data transmission with limited bandwidth.

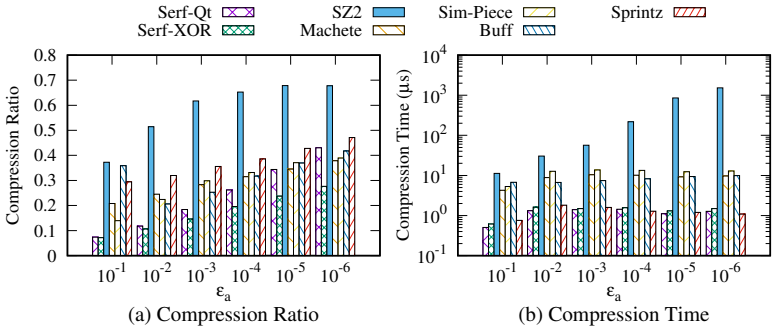


Fig. 15. Performance with Different ϵ_a.

7.4 Performance with Different ϵ_a

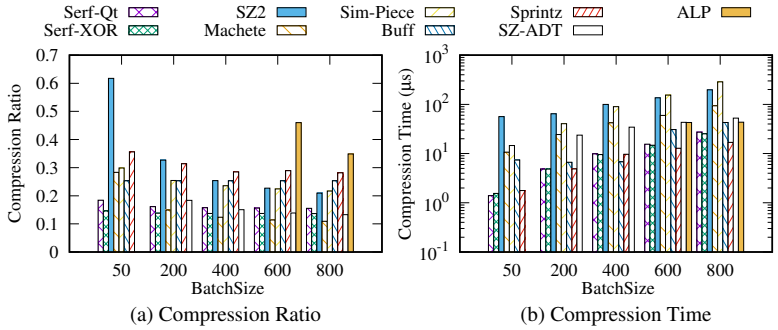
To study the effect of error bound on the performance, we carry out a set of experiments by gradually changing ϵ_a from 10^{-1} to 10^{-6}. We only compare the lossy methods in this set of experiments, because the lossless methods are irrelevant to the error bound. We here still do not compare SZ_ADT and HIRE due to their running exception or poor performance in the batch size of 50.

As shown in Fig. 15(a), as ϵ_a decreases from 10^{-1} to 10^{-6}, the compression ratio of all methods shows an increasing trend, because a tighter error bound means that less information loss is allowed, so we need to store more information to guarantee the fidelity of data. Specifically, for *Serf-Qt*, if the error bound gets smaller, the quantized values tend to be bigger, causing more required bits using Elias gamma coding. For *Serf-XOR*, a stricter error bound means a narrower range for approximated values to choose from, which results in fewer trailing zeros in the XORed values. *Serf-XOR* always enjoys the best compression ratio among all the lossy compression methods. With a smaller ϵ_a, the advantage of *Serf-XOR* over *Serf-Qt* is more obvious in terms of compression ratio. For example, if $\epsilon_a = 10^{-1}$, *Serf-XOR* has a relative compression ratio improvement of 3.5% over *Serf-Qt* (0.071 VS 0.074). When $\epsilon_a = 10^{-6}$, the relative improvement turns out to be as much as 72.3% (0.277 VS 0.43).

The compression time of all methods shows a similar increasing trend with the decreasing of ϵ_a, as shown in Fig. 15(b). There are two possible reasons. First, with a smaller ϵ_a, all methods have a worse compression ratio, leading to more bits to write. Second, a tighter ϵ_a means more validations. For example, *Serf-XOR* should perform more iterations to find the approximated values. *Serf-Qt* , *Serf-XOR* and Sprintz always take less compression time than other lossy methods, since their logics are relatively simple, while other methods perform much more complicated operations, e.g., SZ2 and Machete require to build Huffman trees, which is time-consuming.

7.5 Performance with Different Batch Sizes

One may transmit data using batched compression methods with mini-batches. To investigate if *Serf* still outperforms the batched methods, we conduct a set of experiments with the batch size varying from 50 to 800, where $\epsilon_a = 0.001$ for lossy methods. We do not compare other streaming methods (e.g., Gorilla and Elf) because their performance is almost unaffected by the batch size. For a clear presentation, we do not compare the general compression methods (e.g., Snappy and LZ77), because they usually perform worse than floating-point specific compression methods even if the batch size is 1,000 according to [37]. Note that a larger batch size brings about a longer delay and more memory consumption for the batched methods, while the delay and memory consumption of *Serf* can always be neglectable. Besides, batched methods can leverage the global information in



(a) Compression Ratio (b) Compression Time

Fig. 16. Performance with Different Batch Sizes.

a batch to enhance the compression ratio, but our streaming method *Serf* utilizes only the seen information in a stream. Therefore, this set of experiments is unfair to *Serf* to some extent.

Fig. 16 shows the experimental results, where the compression ratio and compression time are based on the records in a batch. We do not present the results of SZ-ADT and ALP when batch size is smaller than 200 and 600 respectively since they cannot run successfully or manifest a poor performance in our settings. We can see from Fig. 16(a) that, with the increasing of batch size, the batched methods have a better compression ratio, because they can discover more common information in a larger batch. If the batch size is smaller than 200, *Serf-XOR* enjoys the best compression ratio. If the batch size is greater than 400, Machete performs the best, since it adopts many optimizations such as adaptive encoding for mini-batches. However, both *Serf-Qt* and *Serf-XOR* are still better than SZ2, Sim-Piece, Buff, Sprintz and ALP in terms of the compression ratio if the batch size is up to 800.

As shown in Fig. 16(b), with a larger batch size, all methods take more compression time, because more data records need to be processed. However, both *Serf-XOR* and *Serf-Qt* enjoy competitive compression efficiency in all small batch sizes.

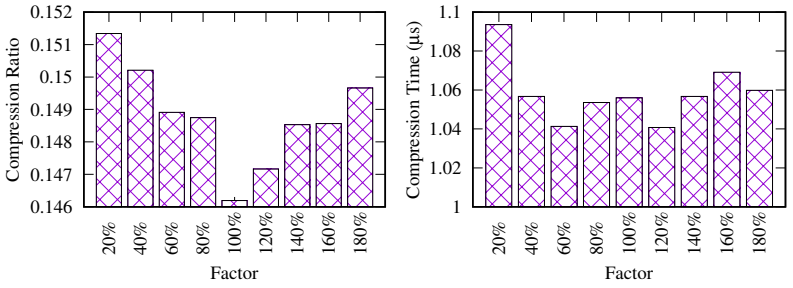
HIRE [9] requires the batch size to be the exponents of 2, so we test it separately with the batch size of 128, 256, 512, 1024, 2048 respectively, obtaining the corresponding compression ratio of 0.79, 0.54, 0.35, 0.24, 0.18 respectively. We can see that even if the batch size is as large as 2048, it is still not comparable to *Serf*.

Overall, even when considering mini-batches, both *Serf-XOR* and *Serf-Qt* are still among the most competitive methods. Therefore, *Serf* also holds potential in data management [34] or data transmission using mini-batched compression.

7.6 Performance with Different Shift Offsets

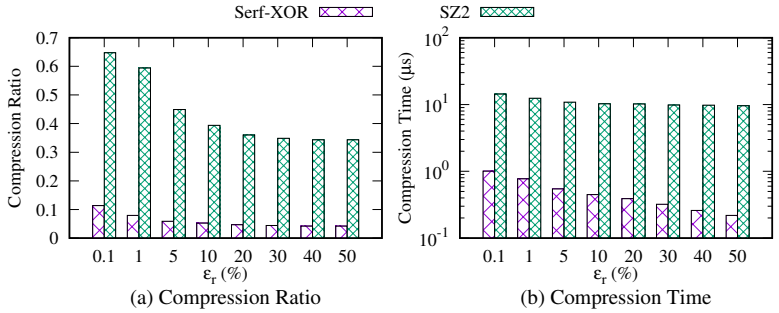
Theorem 1 proves that we can find the best shift offset λ theoretically. To investigate the effects of different values of λ, we perform *Serf-XOR* on all datasets by assigning λ with $20\%\hat{\lambda}$, $40\%\hat{\lambda}$, $60\%\hat{\lambda}$, ..., $180\%\hat{\lambda}$, respectively, where $\hat{\lambda}$ is the theoretically best shift offset. As shown in Fig. 17(a), if λ is set with the theoretically best value (i.e., $\lambda = 100\%\hat{\lambda}$), we can achieve the overall best compression ratio. If the shift offset deviates further from $\hat{\lambda}$, the compression ratio gets worse, since an overlarge shift offset can lead to a lot of information loss, while a too small shift offset cannot guarantee that the shifted values have the same sign and same exponent.

Fig. 17(b) shows that the distribution of the compression time is similar to that of the compression ratio. This is because a better compression ratio contributes to fewer compressed bits, leading to less writing into the stream.



(a) Compression Ratio ($\epsilon_a = 0.001$). (b) Compression Time ($\epsilon_a = 0.001$).

Fig. 17. Performance with Different Shift Offsets.



(a) Compression Ratio (b) Compression Time

Fig. 18. Performance with Different ϵ_r.

7.7 Performance with Different ϵ_r

Apart from the absolute error bound, *Serf-XOR* supports the relative error bound as well. Among the selected competitors, only SZ2 supports the relative error bound. Therefore, in this set of experiments, we compare the performance of *Serf-XOR* and SZ2 in different values of ϵ_r from 0.1%∼50%.

When ϵ_r changes from 0.1% to 50%, the compression ratios of both SZ and *Serf-XOR* first drop and then decrease slightly (shown in Fig. 18(a)), and their compression time shows a similar trend to that of the compression ratio (shown in Fig. 18(b)). A larger relative error bound means a looser fidelity requirement, hence leading to less information to store. For *Serf-XOR*, with a larger ϵ_r, we can find the qualified approximated values more easily. In extreme circumstances, if ϵ_r is large enough, most records can be assigned to their corresponding previous value, respectively, producing many XORed values of 0, which can enhance the compression ratio tremendously. A better compression ratio can usually contribute to less compression time, since fewer bits need to be written. *Serf-XOR* performs much better than SZ2 in terms of both compression ratio and compression time under all ϵ_r. We can see that *Serf-XOR* enjoys a relative compression ratio improvement of 84%∼91.7% over SZ2, and it takes only about 3.3%∼7.4% compression time of SZ2.

7.8 Ablation Study

To verify the effectiveness of each module in *Serf-XOR*, we design a set of experiments by removing each optimization technique from *Serf-XOR*, respectively, and investigate the average performance of each variant on all datasets. The results are shown in Table 7, where $\epsilon_a = 0.001$. Here, "*Serf-XOR*

Table 7. Ablation Study ($\epsilon_a = 0.001$).

***Serf-XOR* modules**	**CR**	**CT (μs)**	**DT (μs)**
Serf-XOR	**0.1462**	**1.1617**	**0.1602**
Serf-XOR w/o Shifter	0.1549 (↑5.95%)	1.4210 (↑22.32%)	0.2880(↑79.78%)
Serf-XOR w/o OptApp	0.1587 (↑8.55%)	1.1717 (↑0.86%)	0.2062(↑28.71%)
Serf-XOR w/o FastApp	0.1462 (↑0%)	2.8706 (↑147.1%)	0.1684(↑5.12%)

Table 8. Performance on the Time Series Benchmark ($\epsilon_a = 0.001$).

Methods	Batched								Streaming							
	Lossless			Lossy					Lossless						Lossy	
	LZ77	Zstd	Snappy	SZ2	Machete	Sim-Piece	Sprintz	Buff	Deflate	LZ4	FPC	Gorilla	Chimp_{128}	Elf	Serf-Qt	Serf-XOR
CR	0.99	1.02	1.01	0.54	0.24	0.28	0.35	**0.20**	0.93	1.03	0.77	0.77	0.74	0.31	**0.09**	0.10
CT (μs)	0.02	6.10	**0.003**	12.49	3.47	4.93	1.06	6.74	21.86	**0.03**	0.10	0.04	0.61	1.52	1.02	1.14
DT (μs)	**0.0004**	0.15	0.0009	3.99	0.01	1.22	0.75	3.27	2.13	**0.02**	0.39	1.04	1.06	0.73	0.95	0.08

w/o Shifter" represents the method that we do not perform the data shift. "*Serf-XOR* w/o OptApp" stands for the basic approximator shown in Fig. 9. "*Serf-XOR* w/o FastApp" indicates that we do not employ Theorem 3 to accelerate the approximated values search.

As shown in Table 7, both Shifter and OptApp contribute to the compression ratio improvement of *Serf-XOR*, since if we remove them separately, the performance of *Serf-XOR* will decrease by as much as 5.95% and 8.55%, respectively. Shifter can also enhance the compression efficiency and decompression efficiency by 22.32% and 79.78%, respectively. This is because Shifter itself contains only simple arithmetic operations, but it can reduce the compressed bits significantly. Although OptApp introduces additional verifications of the bit combination during compression, since it can reduce the compressed bits, if we remove OptApp, the compression time increases instead of decreases. "*Serf-XOR* w/o OptApp" also has higher decompression time, due to more bits to decompress.

FastApp does not affect the compression ratio, which verifies the correctness of Theorem 3 experimentally. Conversely, it improves the compression efficiency by as much as 147.1%.

7.9 Performance on Time Series Benchmark

We also conduct a set of experiments by running different methods on the datasets generated by the standard time series benchmark [4], which currently supports two use cases: *Dev ops* that generates 100 CPU metrics, and *IoT* that simulates data streaming from a set of trucks. Since the data generated by *Dev ops* are mostly integers, we only use the data (the positions of trucks in specific) generated by the use case *IoT*. We regard the position data as two time series (i.e., latitude and longitude), and report the average performance of each method on these two time series. Most parameters of the data generator are set to default values given by its readme file, except that `scale` is set to 1 to maintain the time series characteristics. Here we do not compare ALP, SZ_ADT and HIRE because of their running exception or poor performance in the batch size of 50.

As depicted in Table 8, in terms of the compression ratio, our proposed methods *Serf-Qt* and *Serf-XOR* still perform the best on the benchmark data, while the dictionary-based methods like LZ77, Snappy and LZ4 perform the worst. As the dataset is simulated without exactly equal values in a mini-batch, the dictionary-based methods cannot build a good dictionary, leading to poor compression ratios but high efficiencies due to no any actual compression/decompression action. We also notice that *Serf-Qt* has a slightly better compression ratio than *Serf-XOR*, because the simulated data can be predicted more easily, resulting in small quantized values that can be encoded compactly using Elias gamma coding.

Table 9. Performance for Single Values ($\epsilon_a = 0.001$).

Methods	Batched				Streaming					
	Lossless			Lossy	Lossless				Lossy	
	LZ77	Zstd	Snappy	SZ2	Deflate	LZ4	Chimp_{128}	Elf	Serf-Qt	Serf-XOR
CR	0.83	**0.81**	0.85	0.95	0.76	0.93	0.63	0.54	0.38	**0.32**
CT (μs)	1.169	6.086	**0.072**	33.572	19.096	1.032	**0.718**	4.049	0.723	0.873
DT (μs)	**0.016**	1.309	0.018	4.340	2.771	0.513	0.971	0.682	0.888	**0.053**

7.10 Performance for Single Values

Both *Serf-Qt* and *Serf-XOR* can be easily extended to other precisions. We conduct a set of experiments to investigate the performance of each method on the datasets that can be fully represented by the single precision (i.e., BW, CDT, DT, PM10, SG and WS). Some other methods (e.g., Machete, Sim-Piece and Buff) do not provide an implementation for single values, so we do not compare with them. Table 9 shows the results, from which we can conclude that:

(1) Among all the methods, *Serf-XOR* still has the best compression ratio. Compared with the best streaming competitor Elf, *Serf-XOR* enjoys a relative compression ratio improvement of 40.74%.

(2) The compression/decompression time rankings of most methods are similar to those for double values, respectively.

(3) All methods have much worse compression ratios for single values than for double values, but their compression time and decompression time are similar. Specifically, the compression ratios of *Serf-Qt* and *Serf-XOR* for single values are about twice these for double values, respectively. This is because the numbers of compressed bits for single values are similar to these for double values, but the original sizes of single values are half of double values.

8 Conclusion

This paper proposes the first streaming error-bounded compression algorithm *Serf*, which can be divided into *Serf-Qt* and *Serf-XOR*. Extensive experiments using 13 datasets show the powerful performance of *Serf-XOR*. For example, when the absolute error bound is 0.001, *Serf-XOR* enjoys 40% relative compression ratio improvement over the best competitor Buff with even a higher compression/decompression efficiency. Given a relative error bound, *Serf-XOR* enjoys a relative compression ratio improvement of 82.5%∼87.7% over SZ2, but takes only about 2.3%∼7.0% compression time. In the streaming data transmission scenarios, when the bandwidth is 240 kbit/s, *Serf-XOR* takes only half the time of the best streaming competitor Elf. In future work, we plan to integrate *Serf* into systems like ModelarDB [25–27] or IoTDB [64], e.g., implementing a serializer/deserializer based on *Serf* to accelerate the data transmission among different machines. Analyzing and indexing compressed time series directly [21, 63] are also interesting research directions.

Acknowledgment

This paper is supported by National Natural Science Foundation of China (62202070, 62322601), China Postdoctoral Science Foundation (2022M720567), Fundamental Research Funds for the Central Universities (2024IAIS-QN017), the Independent Research Project of State Key Laboratory of Mechanical Transmission for Advanced Equipment (SKLMT-ZZKT-2024R07), the Excellent Youth Foundation of Chongqing (CSTB2023NSCQJQX0025), and Major Basic Research Project of Shandong Provincial Natural Science Foundation (ZR2024ZD03).

References

[1] 2023. Daily Temperature of Major Cities. Retrieved March 19, 2023 from https://www.kaggle.com/sudalairajkumar/daily-temperature-of-major-cities.

[2] 2023. Financial data set used in INFORE project. Retrieved March 19, 2023 from https://zenodo.org/record/3886895#.Y4DdzHZByM_.

[3] 2023. Historical Weather Data Download. Retrieved March 19, 2023 from https://www.meteoblue.com/en/weather/archive/export/basel_switzerland.

[4] 2023. Time Series Benchmark Suite (TSBS). https://github.com/timescale/tsbs.

[5] 2024. Electric Motor Temperature. Retrieved October 10, 2024 from https://www.kaggle.com/datasets/wkirgsn/electric-motor-temperature.

[6] 2024. PSML: A Multi-scale Time-series Dataset for Machine Learning in Decarbonized Energy Grids (Dataset). Retrieved October 5, 2024 from https://zenodo.org/records/5130612.

[7] 2024. Serf: Streaming Error-Bounded Floating-Point Compression. https://github.com/Spatio-Temporal-Lab/Serf.

[8] Azim Afroozeh, Leonardo X Kuffo, and Peter Boncz. 2023. ALP: Adaptive Lossless floating-Point Compression. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–26.

[9] Bruno Barbarioli, Gabriel Mersy, Stavros Sintos, and Sanjay Krishnan. 2023. Hierarchical residual encoding for multiresolution time series compression. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.

[10] Matěj Bartík, Sven Ubik, and Pavel Kubalik. 2015. LZ4 compression algorithm on FPGA. In *2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*. IEEE, 179–182.

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

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

[13] Kin-Pong Chan and Ada Wai-Chee Fu. 1999. Efficient time series matching by wavelets. In *Proceedings 15th International Conference on Data Engineering (Cat. No. 99CB36337)*. IEEE, 126–133.

[14] Xinyu Chen, Jiannan Tian, Ian Beaver, Cynthia Freeman, Yan Yan, Jianguo Wang, and Dingwen Tao. 2024. FCBench: Cross-Domain Benchmarking of Lossless Compression for Floating-Point Data. *Proceedings of the VLDB Endowment* 17, 6 (2024), 1418–1431.

[15] Y Collet. 2016. Zstd github repository from facebook. Retrieved March 19, 2023 from https://github.com/facebook/zstd.

[16] Sheng Di and Franck Cappello. 2016. Fast error-bounded lossy HPC data compression with SZ. In *2016 ieee international parallel and distributed processing symposium (ipdps)*. IEEE, 730–739.

[17] Frank Eichinger, Pavel Efros, Stamatis Karnouskos, and Klemens Böhm. 2015. A time-series compression technique and its application to the smart grid. *The VLDB Journal* 24 (2015), 193–218.

[18] Peter Elias. 1975. Universal codeword sets and representations of the integers. *IEEE transactions on information theory* 21, 2 (1975), 194–203.

[19] Hazem Elmeleegy, Ahmed Elmagarmid, Emmanuel Cecchet, Walid G Aref, and Willy Zwaenepoel. 2009. Online piece-wise linear approximation of numerical streams with precision guarantees. (2009).

[20] Christos Faloutsos, Mudumbai Ranganathan, and Yannis Manolopoulos. 1994. Fast subsequence matching in time-series databases. *ACM Sigmod Record* 23, 2 (1994), 419–429.

[21] Francesco Fusco, Marc Ph Stoecklin, and Michail Vlachos. 2010. Net-fli: on-the-fly compression, archiving and indexing of streaming network traffic. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 1382–1393.

[22] Jianhua Gao, Weixing Ji, Lulu Zhang, Senhao Shao, Yizhuo Wang, and Feng Shi. 2020. Fast piecewise polynomial fitting of time-series data for streaming computing. *IEEE Access* 8 (2020), 43764–43775.

[23] Google. 2001. Protocol buffers encoding. https://protobuf.dev/programming-guides/encoding/.

[24] Google. 2023. Snappy | A fast compressor/decompressor. Retrieved March 19, 2023 from https://github.com/google/snappy.

[25] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. 2018. Modelardb: Modular model-based time series management with spark and cassandra. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1688–1701.

[26] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. 2021. Scalable model-based management of correlated dimensional time series in modelardb+. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1380–1391.

[27] Søren Kejser Jensen, Christian Thomsen, and Torben Bach Pedersen. 2023. ModelarDB: integrated model-based management of time series from edge to cloud. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems LIII*. Springer, 1–33.

[28] Søren Kejser Jensen, Christian Thomsen, Torben Bach Pedersen, Carlos Enrique Muñiz-Cuza, and Abduvoris Abduvakhobov. 2024. Why Model-Based Lossy Compression is Great for Wind Turbine Analytics. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 5667–5668.

[29] Pu Jiao, Sheng Di, Hanqi Guo, Kai Zhao, Jiannan Tian, Dingwen Tao, Xin Liang, and Franck Cappello. 2022. Toward quantity-of-interest preserving lossy compression for scientific data. *Proceedings of the VLDB Endowment* 16, 4 (2022), 697–710.

[30] William Kahan. 1996. IEEE standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE* 754, 94720-1776 (1996), 11.

[31] Eamonn Keogh, Kaushik Chakrabarti, Michael Pazzani, and Sharad Mehrotra. 2001. Dimensionality reduction for fast similarity search in large time series databases. *Knowledge and information Systems* 3 (2001), 263–286.

[32] Xenophon Kitsios, Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. 2023. Sim-piece: highly accurate piecewise linear approximation through similar segment merging. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1910–1922.

[33] Mona Kumari, Ajitesh Kumar, and Arbaz Khan. 2020. IoT based intelligent real-time system for bus tracking and monitoring. In *2020 International Conference on Power Electronics & IoT Applications in Renewable Energy and its Control (PARC)*. IEEE, 226–230.

[34] Ruiyuan Li, Huajun He, Rubin Wang, Yuchuan Huang, Junwen Liu, Sijie Ruan, Tianfu He, Jie Bao, and Yu Zheng. 2020. Just: Jd urban spatio-temporal data engine. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1558–1569.

[35] Ruiyuan Li, Huajun He, Rubin Wang, Sijie Ruan, Tianfu He, Jie Bao, Junbo Zhang, Liang Hong, and Yu Zheng. 2021. TrajMesa: A Distributed NoSQL-Based Trajectory Data Management System. TKDE (2021), 1–1.

[36] Ruiyuan Li, Zheng Li, Yi Wu, Chao Chen, Tong Liu, and Yu Zheng. 2023. Adaptive Encoding Strategies for Erasing-Based Lossless Floating-Point Compression. *arXiv preprint arXiv:2308.11915* (2023).

[37] Ruiyuan Li, Zheng Li, Yi Wu, Chao Chen, and Yu Zheng. 2023. Elf: Erasing-Based Lossless Floating-Point Compression. *Proceedings of the VLDB Endowment* 16, 7 (2023), 1763–1776.

[38] Tianyi Li, Lu Chen, Christian S Jensen, and Torben Bach Pedersen. 2021. TRACE: Real-time compression of streaming trajectories in road networks. *Proceedings of the VLDB Endowment* 14, 7 (2021), 1175–1187.

[39] Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. 2022. Chimp: efficient lossless floating point compression for time series databases. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3058–3070.

[40] Xin Liang, Sheng Di, Dingwen Tao, Zizhong Chen, and Franck Cappello. 2018. An efficient transformation scheme for lossy data compression with point-wise relative error bound. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 179–189.

[41] Xin Liang, Sheng Di, Dingwen Tao, Sihuan Li, Shaomeng Li, Hanqi Guo, Zizhong Chen, and Franck Cappello. 2018. Error-controlled lossy compression optimized for high compression ratios of scientific datasets. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 438–447.

[42] Xin Liang, Kai Zhao, Sheng Di, Sihuan Li, Robert Underwood, Ali M Gok, Jiannan Tian, Junjing Deng, Jon C Calhoun, Dingwen Tao, et al. 2022. SZ3: A modular framework for composing prediction-based error-bounded lossy compressors. *IEEE Transactions on Big Data* 9, 2 (2022), 485–498.

[43] Jessica Lin, Eamonn Keogh, Stefano Lonardi, and Bill Chiu. 2003. A symbolic representation of time series, with implications for streaming algorithms. In *Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*. 2–11.

[44] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. 2022. On-device training under 256kb memory. *Advances in Neural Information Processing Systems* 35 (2022), 22941–22954.

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

[46] Tao Lu, Yu Zhong, Zibin Sun, Xiang Chen, You Zhou, Fei Wu, Ying Yang, Yunxin Huang, and Yafei Yang. 2023. ADT-FSE: A New Encoder for SZ. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[47] National Ecological Observatory Network (NEON). 2022. 2D wind speed and direction (DP1.00001.001). doi:10.48443/77N6-EH42 Retrieved March 19, 2023 from https://data.neonscience.org/data-products/DP1.00001.001/RELEASE-2022.

[48] National Ecological Observatory Network (NEON). 2022. Barometric pressure (DP1.00004.001). doi:10.48443/ZR37-0238 Retrieved March 19, 2023 from https://data.neonscience.org/data-products/DP1.00004.001/RELEASE-2022.

[49] National Ecological Observatory Network (NEON). 2022. Dust and particulate size distribution (DP1.00017.001). doi:10.48443/RDZ9-XR84 Retrieved March 19, 2023 from https://data.neonscience.org/data-products/DP1.00017.001/RELEASE-2022.

[50] National Ecological Observatory Network (NEON). 2022. IR biological temperature (DP1.00005.001). doi:10.48443/7RS6-FF56 Retrieved March 19, 2023 from https://data.neonscience.org/data-products/DP1.00005.001/RELEASE-2022.

[51] National Ecological Observatory Network (NEON). 2022. Relative humidity above water on-buoy (DP1.20271.001). doi:10.48443/1W06-WM51 Retrieved March 19, 2023 from https://data.neonscience.org/data-products/DP1.20271.001/RELEASE-2022.

[52] Dinh C Nguyen, Ming Ding, Pubudu N Pathirana, Aruna Seneviratne, Jun Li, Dusit Niyato, Octavia Dobre, and H Vincent Poor. 2021. 6G Internet of Things: A comprehensive survey. *IEEE Internet of Things Journal* 9, 1 (2021), 359–383.

[53] Savan Oswal, Anjali Singh, and Kirthi Kumari. 2016. Deflate compression algorithm. *International Journal of Engineering Research and General Science* 4, 1 (2016), 430–436.

[54] Themis Palpanas, Michail Vlachos, Eamonn Keogh, and Dimitrios Gunopulos. 2008. Streaming time series summarization using user-defined amnesic functions. *IEEE Transactions on Knowledge and Data Engineering* 20, 7 (2008), 992–1006.

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

[56] Taha M Rajeh, Zhipeng Luo, Muhammad Hafeez Javed, Fares Alhaek, and Tianrui Li. 2024. A Clustering-Based Multi-Agent Reinforcement Learning Framework for Finer-Grained Taxi Dispatching. *IEEE Transactions on Intelligent Transportation Systems* (2024).

[57] Younes Regragui and Najem Moussa. 2023. A real-time path planning for reducing vehicles traveling time in cooperative-intelligent transportation systems. *Simulation Modelling Practice and Theory* 123 (2023), 102710.

[58] Sijie Ruan, Xi Fu, Cheng Long, Zi Xiong, Jie Bao, Ruiyuan Li, Yiheng Chen, Shengnan Wu, and Yu Zheng. 2021. Filling delivery time automatically based on couriers' trajectories. *IEEE Transactions on Knowledge and Data Engineering* 35, 2 (2021), 1528–1540.

[59] Cyrus Shahabi, Xiaoming Tian, and Wugang Zhao. 2000. TSA-tree: A wavelet-based approach to improve the efficiency of multi-level surprise and trend queries on time-series data. In *Proceedings. 12th International Conference on Scientific and Statistica Database Management*. IEEE, 55–68.

[60] Hagit Shatkay and Stanley B Zdonik. 1996. Approximate queries and representations for large data sequences. In *Proceedings of the twelfth international conference on data engineering*. IEEE, 536–545.

[61] Yang Shi, Xiangyu Zou, Xinyu Chen, Sian Jin, Dingwen Tao, Deng Cai, Yufan Chen, and Wen Xia. 2024. Machete: An Efficient Lossy Floating-Point Compressor Designed for Time Series Databases. In *2024 Data Compression Conference (DCC)*.

[62] 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 *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1129–1139.

[63] Michail Vlachos, Nikolaos M Freris, and Anastasios Kyrillidis. 2015. Compressive mining: fast and optimal data mining in the compressed domain. *The VLDB Journal* 24, 1 (2015), 1–24.

[64] Chen Wang, Xiangdong Huang, Jialin Qiao, Tian Jiang, Lei Rui, Jinrui Zhang, Rong Kang, Julian Feinauer, Kevin A McGrail, Peng Wang, et al. 2020. Apache IoTDB: Time-series database for internet of things. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2901–2904.

[65] Satya Prakash Yadav, Subiya Zaidi, Caio Dos Santos Nascimento, Victor Hugo C de Albuquerque, and Sansar Singh Chauhan. 2023. Analysis and Design of automatically generating for GPS Based Moving Object Tracking System. In *2023 International Conference on Artificial Intelligence and Smart Communication (AISC)*. IEEE, 1–5.

[66] Zehai Yang and Shimin Chen. 2024. MOST: Model-Based Compression with Outlier Storage for Time Series Data. *Proceedings of the ACM on Management of Data* 1, 4 (2024), 1–29.

[67] Jing Yuan, Yu Zheng, Chengyang Zhang, Wenlei Xie, Xing Xie, Guangzhong Sun, and Yan Huang. 2010. T-drive: driving directions based on taxi trajectories. In *Proceedings of the 18th SIGSPATIAL International conference on advances in geographic information systems*. 99–108.

[68] 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 *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1643–1654.

[69] Kai Zhao, Sheng Di, Xin Liang, Sihuan Li, Dingwen Tao, Zizhong Chen, and Franck Cappello. 2020. Significantly improving lossy compression for HPC datasets with second-order prediction and parameter optimization. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*. 89–100.

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

[71] Alireza Zohourian, Sajjad Dadkhah, Euclides Carlos Pinto Neto, Hassan Mahdikhani, Priscilla Kyei Danso, Heather Molyneaux, and Ali A Ghorbani. 2023. IoT Zigbee device security: A comprehensive review. *Internet of Things* (2023), 100791.

Received October 2024; revised January 2025; accepted February 2025