DeXOR: Enabling XOR in Decimal Space for Streaming Lossless Compression of Floating-point Data

Chuanyi Lv*
The State Key Laboratory of Blockchain and Data Security, Zhejiang University
chuanyi.lv@zju.edu.cn

Huan Li†*
The State Key Laboratory of Blockchain and Data Security, Zhejiang University
lihuan.cs@zju.edu.cn

Dingyu Yang*
The State Key Laboratory of Blockchain and Data Security, Zhejiang University
yangdingyu@zju.edu.cn

Zhongle Xie*
School of Software Technology, Zhejiang University
xiezl@zju.edu.cn

Lu Chen
College of Computer Science and Technology, Zhejiang University
luchen@zju.edu.cn

Christian S. Jensen
Department of Computer Science, Aalborg University
csj@cs.aau.dk

ABSTRACT

With streaming floating-point numbers being increasingly prevalent, effective and efficient compression of such data is critical. Compression schemes must be able to exploit the similarity, or smoothness, of consecutive numbers and must be able to contend with extreme conditions, such as high-precision values or the absence of smoothness. We present DeXOR, a novel framework that enables DECIMAL XOR procedure to encode decimal-space longest common prefixes and suffixes, achieving optimal prefix reuse and effective redundancy elimination. To ensure accurate and low-cost decompression even with binary-decimal conversion errors, DeXOR incorporates 1) scaled truncation with error-tolerant rounding and 2) different bit management strategies optimized for DECIMAL XOR. Additionally, a robust exception handler enhances stability by managing floating-point exponents, maintaining high compression ratios under extreme conditions. In evaluations across 22 datasets, DeXOR surpasses state-of-the-art schemes, achieving a 15% higher compression ratio and a 20% faster decompression speed while maintaining a competitive compression speed. DeXOR also offers scalability under varying conditions and exhibits robustness in extreme scenarios where other schemes fail.

PVLDB Reference Format:
Chuanyi Lv, Huan Li, Dingyu Yang, Zhongle Xie, Lu Chen, and Christian S. Jensen. DeXOR: Enabling XOR in Decimal Space for Streaming Lossless Compression of Floating-point Data. PVLDB, 19(5): 849 - 861, 2026.
doi:10.14778/3796195.3796200

PVLDB Artifact Availability:
The source code, data, and other artifacts are available at https://github.com/SuDIS-ZJU/DeXOR.

*Also affiliated with Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, Hangzhou, China.
†Huan Li is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 19, No. 5 ISSN 2150-8097.
doi:10.14778/3796195.3796200

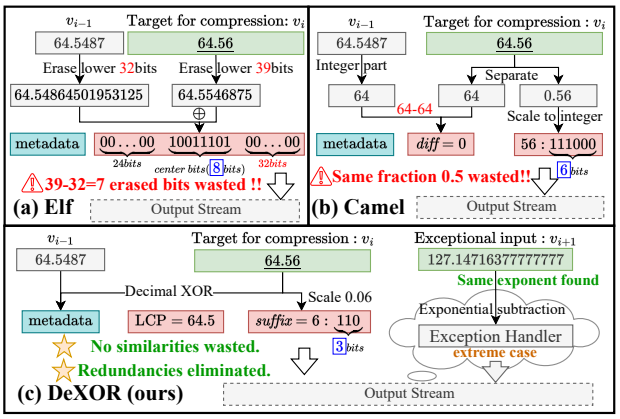


Figure 1: Compressing the target 64.56 with the previous value 64.5487: (a) Elf [33, 35] removes trailing zeros in the binary representation but wastes bits after XOR. (b) Camel [49] targets the integer part of a decimal value, leaving the fractional part unoptimized. (c) DeXOR extracts the longest common decimal prefix (64.5) to exploit smoothness and scales the suffix (from 0.06 to 6) to remove redundancy. It also includes an exception handler based on exponential subtraction for extreme-case target values (127.14716377 ⋯).

1 INTRODUCTION

The rapidly growing Internet of Things [31] emits massive volumes of streaming data, typically in the form of time series. For example, aviation systems emit terabytes per second [23]. Across many domains, including industrial automation, finance, and scientific research, high-precision floating-point data streams require accurate decompression, as even minor inaccuracies can have critical operational consequences. As a result, lossless compression with real-time efficiency is a fundamental and pressing challenge.

To address this challenge, *streaming lossless compression* (SLC) has emerged as a key research focus. The present study targets the common setting where only the previous value is required to compress the current one (Figure 1), addressing scenarios with strict memory constraints or streaming applications where retaining earlier values is infeasible. Existing SLC approaches bifurcate into

two categories: *smoothness-based* and *redundancy-based* schemes. Smoothness-based schemes, such as Gorilla [42] and Chimp [37], leverage the similarity among consecutive values, called *smoothness*, to compress using bitwise operations (e.g., XOR). While effective for sequential patterns, these schemes struggle with the complex storage formats used for floating-point numbers. In contrast, redundancy-based schemes target representational redundancy in floating-point structures, introducing leading or trailing zeros to minimize bit storage. Prominent examples include scaling-to-integer schemes (PDE [27] and ALP [8]). However, such schemes often do not exploit inherent smoothness in the data, causing inferior compression ratios.

Recent proposals, including the Elf series [32–35] and Camel [49], attempt to combine smoothness- and redundancy-based schemes (see Figure 1(a) and (b)). However, this integration causes interference, where redundancy elimination involves format conversion, disrupting smoothness. For example, in the Elf series, mantissa erasure disrupts smoothness by misaligning binary representations with differing precision. In Figure 1(a), 64.56 erases 39 trailing zeros, while 64.5487 erases 32, resulting in 64.56 losing 7 additional trailing zeros after the XOR, leading to redundancy in the compressed output. Next, Camel [49], covered in Figure 1(b), separates integer and fractional components, partially exploiting similarities in the integer parts (64 – 64), while reducing redundancies in the fractional part by scaling (0.56 to 56 for an integer-based representation). However, this scheme does not exploit smoothness in the fractional parts, limiting its compression ratios.

Unlike schemes constrained by binary systems, Camel pioneers smoothness exploitation in the decimal domain, inspiring our design of DeXOR, a unified framework that synergizes smoothness- and redundancy-based techniques via *decimal-space conversion*. Its core is the DECIMAL XOR, which transforms binary floating-point values into decimal-aligned ones, isolating the largest common prefix for smoothness while leaving residuals for redundancy elimination. Consider the target 64.56 in Figure 1(c). Using DECIMAL XOR, the **longest common prefix (LCP)** with the previous value — e.g., *64.5* is shared between 64.56 and 64.5487 — is identified and removed. The residual value (0.06) is then scaled into an integer suffix (6), achieving optimal redundancy elimination.

While DECIMAL XOR is conceptually simple, its processing w.r.t SLC workflows faces three key challenges:

- **C1 (Lossless Reconstruction)**: Binary-to-decimal conversion risks precision loss (see Section 2.2), compromising precise suffix extraction and violating the guarantee of *lossless* reconstruction.
- **C2 (Computational Overhead)**: Prefix extraction for smoothness, as well as suffix processing for redundancy elimination, imports additional computational and metadata costs, increasing time complexity and storage demands.
- **C3 (Robustness in Extreme Cases)**: For high-precision, non-smooth datasets with little redundancy, DECIMAL XOR benefits may decline, just incurring increased time or space costs.

It is essential to address these challenges, DeXOR offers a suite of sophisticated optimizations.

To address Challenge C1, DeXOR ensures accurate prefix and suffix identification through a ***scaled truncation approach with error-tolerant rounding*** (Section 4.2.1). Unlike traditional schemes that convert from binary to decimal formats for string matching, DeXOR computes prefixes and suffixes directly from vanilla binary format. Moreover, an error-tolerant rounding mechanism is introduced to mitigate edge-case risks, ensuring the DECIMAL XOR operation is robust and lossless. This eliminates conversion-induced prefix and suffix identification errors while additionally reducing computational overhead.

To address Challenge C2, DeXOR offers several ***DECIMAL XOR-specific bit management strategies***: (i) DeXOR minimizes prefix storage complexity by encoding the LCP and suffix using minimal coordinate data (i.e., metadata in Figure 1(c)), and a lightweight 2-bit case encoding further enables efficient reuse of these coordinate results (Section 4.2.2). (ii) By identifying the sign consistency between the suffix and prefix; and by assessing the inefficiency of variable-length suffix storage, DeXOR adopts an optimal fixed-bit unsigned suffix encoding design (Section 4.3).

To address Challenge C3, DeXOR includes an ***exception handler*** operating in parallel with the DECIMAL XOR-driven main pipeline, designed expressly to handle high-precision or non-smooth streams. The handler isolates the floating-point exponent and encodes the exponential subtraction (see Figure 1(c)), leveraging smoothness in the data while bypassing data components with minimal benefits (Section 5.1). Given the bounded range, low volatility, and low-bit density of exponential subtraction, DeXOR introduces compact adaptive-length encoding to efficiently store essential information (Section 5.2). The handler can also function as a standalone compressor for datasets known to exhibit exceptional properties.

We evaluate DeXOR using 22 publicly available datasets commonly used in state-of-the-art research [33, 35, 37, 42, 49]. Across all metrics, DeXOR establishes itself as the new leader, achieving 15% higher compression ratios and 20% faster decompression speeds compared to the best-performing competitors, while maintaining comparable compression speeds. On low-precision datasets, DeXOR delivers an impressive 30% improvement in decompression speed and 20+% faster compression speed. It achieves the highest compression ratios across nearly all datasets and exhibits robust performance, maintaining competitive compression even in extreme cases where other schemes fail. Consistent 15%–20% compression gains are impactful as they offer benefits across storage cost [42], network bandwidth [18], query performance [7, 14], and system sustainability [40, 47], thereby delivering substantial benefits.

The main contributions are summarized as follows.

- We revisit existing SLC schemes and provide insights into leveraging smoothness and redundancy with decimal-space operations. Building on these insights, we propose DeXOR, a comprehensive SLC scheme centered around DECIMAL XOR (Sections 2 and 3).
- We equip DeXOR's main pipeline with DECIMAL XOR-specific optimizations, including scaled truncation with error-tolerant rounding for efficient and precise prefix-suffix extraction, as well as different bit management strategies to ensure compact storage of data and metadata (Section 4).
- We present an exception handler to complement DECIMAL XOR in extreme settings involving high-precision and non-smooth streams. The handler stabilizes exponents and employs adaptive encoding, maintaining high compression ratios even where other schemes fall short (Section 5).

- We report on extensive experiments involving pertinent datasets and baselines, offering evidence that DeXOR is capable of superior compression ratios and decompression speeds, offers scalability under varying conditions, and achieves robustness in extreme scenarios (Section 6).

2 PRELIMINARIES

2.1 Numerical Data Representation

In practice, numerical values are most commonly represented as floating-point numbers rather than integers, with the **IEEE754 standard** [4] being the *de facto* format. Moreover, computer systems must store each numerical value v_i using a binary representation ω_i, most often in 64-bit double-precision form:

$$f_{\text{IEEE754}} : v_i \rightarrow \omega_i = \underbrace{sign}_{1\ bit} \ || \ \underbrace{exp}_{11\ bits} \ || \ \underbrace{fraction}_{52\ bits}. \quad (1)$$

The original decimal value v_i can be reconstructed as:

$$v_i \approx sign \times 2^{(exp-bias)} \times (1 + fraction), \quad (2)$$

where *sign* indicates polarity; *exp* is the stored exponent; *bias* is a predefined offset that maps negative exponents to non-negative values for compact storage (the actual exponent is $exp - bias$); and *fraction* represents the significand as a fraction, which, when added to 1, forms the complete significand value. This format ensures compact and efficient representation across platforms. This study adopts double-precision (with $bias = 1023$ in Equation 2) by default due to its higher accuracy. Nevertheless, the proposed compression techniques can be applied readily to single-precision values, with a similar format (8-bit *exp* and 23-bit *fraction*) but $bias = 127$.

Rounding Errors in Binary-decimal Conversions. Limited bits in IEEE754 introduce rounding errors during binary-decimal conversions in Equation 2. For example, when storing a decimal value $v_i = 88.1479$, it may be reconstructed as $v_i' = 88.14790000000000702\cdots$, causing a negligible error (e.g., the rounding error is below $2^{(exp-bias)-52}$ for 64-bit precision). While typically insignificant, rounding errors can accumulate in typical compression schemes involving binary-decimal conversions [8, 27, 49]; thus, care is needed to maintain accuracy. We discuss how to combat this issue in Section 4.2.

2.2 Problem Formulation

Following prior studies [33, 35, 37, 42, 49], we assume that the compression system processes a stream of incoming numerical values, denoted as $v_1, v_2, \cdots$. Such a stream may originate from a subscription to an existing time series, or it may encompass random data emitted by multiple sources. Timestamps, if present, are excluded from consideration in our framework, as they are typically handled separately using specialized compression techniques, such as *Delta-of-delta* compression [42]. Instead, we focus exclusively on the compression of numerical values. As discussed, such input data is generally represented as bit streams, defined as follows.

Definition 1 (Bit Stream). *An input bit stream is denoted as $\omega_1\omega_2\cdots$, where each ω_i is the IEEE754 binary representation of a numerical value v_i from the decimal space.*

We define our problem as follows.

Problem Statement (Streaming Lossless Compression, SLC). *Given an input bit stream, without assuming temporal smoothness, streaming compression is a function $C : \omega_{i-N}\cdots\underline{\omega_i} \rightarrow \omega_i'$ that transforms a target $\underline{\omega_i}$ into a compressed binary representation ω_i' using a buffer of N preceding values. A streaming lossless compression ensures a reverse mapping $\mathcal{D} : \omega_{i-N}'\cdots\omega_i' \rightarrow \underline{\omega_i}$, enabling exact recovery of $\underline{\omega_i}$ from the compressed bit stream (ω_i' and its N preceding ones).*

Our problem statement adopts the univariate streaming setting, thus aligning with prevailing state-of-the-art methods [8, 35, 37, 42]. For high-dimensional datasets (e.g., vector data), each dimension can be compressed independently and assigned to a separate computational thread, enabling straightforward parallelization.

This study addresses SLC for $N = 1$, where compression and decompression involve only the preceding value. This minimal context renders it more challenging to achieve effective compression. When $N > 1$, the larger buffer enables the use of sliding windows (compressing each current using its previous N) or truncated windows (compressing N data in a mini-batch). As shown in Section 6.6, our proposal outperforms schemes using larger buffers in terms of compression ratio and/or efficiency.

2.3 Rethinking Existing SLC Schemes

In numerical data compression, **variable-length encoding** [42] reduces storage by removing unnecessary bits, such as **leading** or **trailing zeros**, resulting from fixed assignments. Referring to the example in Table 1, v_2's IEEE754 representation $\omega(88.1479)$ has one leading zero, which can be omitted. The remaining necessary bits, the **center bits**, start and end with a $\boxed{1}$ bit.

To reconstruct the original value, auxiliary bits (i.e., metadata) are needed to record the length and (start) position of the center bits, with little opportunity to compress these bits. Thus, reducing the **center bit length (CBL)** often becomes a bottleneck when trying to achieve higher compression ratios. To address this, prior approaches [8, 11, 35, 37, 49] introduce preprocessing steps, leveraging properties such as *data smoothness* [11, 35, 37, 49] or *numerical precision redundancy* [8, 35, 49], to shorten the CBL. We refer to these preprocessing steps as **converters**, with Table 1 summarizing common types.

Type-① *Smoothness-based Converters* leverage the smoothness of streaming data, where adjacent values often are similar, a property particularly prevalent in time series processing [28–30, 36].

Example 1. *Methods like Gorilla [42] and Chimp [37] use XOR operations (e.g., $\omega_2 \oplus \omega_1$) to highlight similarities, yielding many leading zeros that reduce the CBL from 63 to 39 (see row 2 in Table 1).*

Camel [49] separates decimal values into integer and fractional parts. For $v_2 = 88.1479$, the integer part 88 is subtracted from the previous integer part and stored (e.g., $88 - 88 = 0$), while the fractional part 0.1479 is mapped into a smaller range using modulus 2^{-4} (corresponding to the 4 decimal places of v_2) and scaled by 10^4. This reduces the center bits to 8 bits but adds a 4-bit overhead for indexing (row 3 in Table 1).

Type-② *Redundancy-based Converters* target IEEE754 specifically and reduce representational redundancy by utilizing knowledge of data precision.

Table 1: Examples of approaches in converters (assuming the previous value $v_1 = 88.1537$ and the current value $v_2 = 88.1479$): leading zeros, center bits, and trailing zeros are highlighted. CBL includes additional cost, e.g., the decimal separation method requires a mapping operation, resulting in an index cost of 4 bits, i.e., (index = 0010) (see row 3 in Table 1).

Type	Approaches	Conversion	Binary Representation (after conversion)	CBL
-	Original IEEE754	$\omega(88.1479)$	0 10000000101011000001001011101110011000110001111110001010000101	63
①	XOR (Gorilla [42] and Chimp [37])	$\omega(88.1479) \oplus \omega(88.1537)$	00000000000000000000000 10100001000010010000100110011000100111 0	39
	Decimal separation (Camel [49])	$(88.1479-88)\ \%2^{-4}\times10^{4}$	00000000000000000000000000000000000000000000000000000000 11100101 ⟹ (index=0010)	12
②	Erasure (Elf [35])	88.14788818359375	0 10000000101011000001001011101110 00000000000000000000000000000000	31
	Scaling to integers (ALP [8])	$88.1479 \times10^{6} \times 10^{-2} = 881479$	00000000000000000000000000000000000000000000000000 11011110111111100111	20
③	Decimal XOR (**ours**)	$(\underline{88.1479} \diamond 88.1537) = 479$	00000000000000000000000000000000000000000000000000000000000000 111011111	9

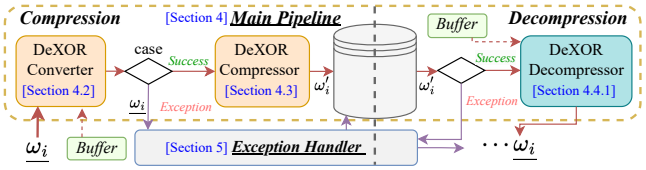


Figure 2: The "main pipeline" features a DECIMAL XOR-based converter and a specialized compressor/decompressor, accompanied by an "exception handler" for extreme cases.

Example 2. *Elf [35] erases unnecessary mantissas, introducing more trailing zeros while preserving the original decimal precision (stored separately) that can round decimal values back (*88.14788 $\cdots$ *in row 4 of Table 1 is rounded up to* 88.1479 *if 4 decimal places are known).*

ALP [8] scales floating-point numbers into integers via $v' = v \times 10^{e} \times 10^{-f}$*, where* $\langle e, f \rangle$ *are chosen carefully to avoid rounding errors. This scaling adds leading zeros, optimizing the CBL to 20 for* 88.1479 *(row 5 in Table 1).*

Joint Utilization of Smoothness and Redundancy. Most SLC converters exploit either smoothness or redundancy, but combining their approaches introduces challenges:

- ***Smoothness Disruption***: Precision-based transformations in type-② can disrupt smoothness in type-①. For instance, the Elf series schemes subtract minimal values based on precision, altering binary representations and reducing similarity to prior values. Similarly, ALP scales values independently, distorting coordinate systems (e.g., 66.101 ⟶ 66101 and 66.1 ⟶ 661).
- ***Partial Solutions***: Camel preserves smoothness in integer components but ignores fractional similarities (e.g., 88.1479 and 88.1537 share 88.1, but only 88 is retained). Elf attempts to combine erasure (type-②) with XOR (type-①), but inconsistencies in CBL after erasure cause inefficiencies in XOR. Here, smoothness is inter-value continuity. In Elf, variable-length bit erasure is applied to each value based on its precision; the subsequent XOR then aligns the lengths to the longer, causing smoothness loss.

In this study, we propose **the first type-③ converter**, which combines the strengths of smoothness- and redundancy-based converters. Our converter is enabled by a decimal-space XOR-like operation that maximizes the exploitation of smoothness (*88.1* shared between 88.1479 and 88.1537) while preserving the remaining portion (88.1479 − 88.1 = 0.0479) for precision-based transformation. As shown in row 6 of Table 1, the converter, named DECIMAL XOR, achieves a shorter CBL than do existing SLC converters.

3 OVERALL FRAMEWORK OF DEXOR

The proposed framework, illustrated in Figure 2, compresses each floating-point number ω_i (in IEEE754 format) through a *main pipeline* (Section 4) built around the DECIMAL XOR technique, supplemented by an *exception handler* (Section 5) for extreme cases.

Main Pipeline: The input is first processed by the DeXOR converter (Section 4.2) to eliminate precision redundancy while exploiting smoothness. This step generates a 2-bit case code for execution status during decompression. A special case code, 11, triggers a switch to the exception process, while other cases proceed with metadata storage and suffix encoding. The suffix, with its significantly reduced information capacity, is then passed to the DeXOR compressor (Section 4.3) for specialized variable-length encoding.

Exception Handler: Data handled poorly by DeXOR (e.g., 16+ decimals) is redirected to the exception handler, in its original IEEE754 format via an *in-line conditional branch*. This module identifies the most stable components (e.g., exponent with high smoothness) for subtraction and applies adaptive storage of the subtraction result to reduce the storage overhead. The exception handling process is consistent with the global input structure, enabling it to perform compression independently when necessary.

The framework integrates a DECIMAL XOR-centered pipeline with a plug-in exception-handling module, to achieve robust performance across diverse datasets. It dynamically determines whether to process data through the main pipeline or the exception handler, optimizing compression for all cases. Thus, prior knowledge of a data stream can allow direct entry into the exception handler (see Section 5.3), reducing the conversion overhead and further improving the compression ratio for high-precision data.

4 MAIN PIPELINE IN DEXOR

4.1 Definition of DECIMAL XOR

As discussed in Section 2.3, leveraging both smoothness in binary representations and precision redundancy in decimal values presents challenges. This inspires us to consider from another perspective: floating-point numbers also exhibit smoothness in decimal space. This suggests that it is possible to apply XOR-like operations first, to strip away the common parts in decimal space, and then handle the redundancy in the residual. Hence, we propose **DECIMAL XOR**, a floating-point data conversion method within a global-aligned decimal coordinate system. It separates the handling of common decimal prefixes and the rest, addressing smoothness and redundancy independently.

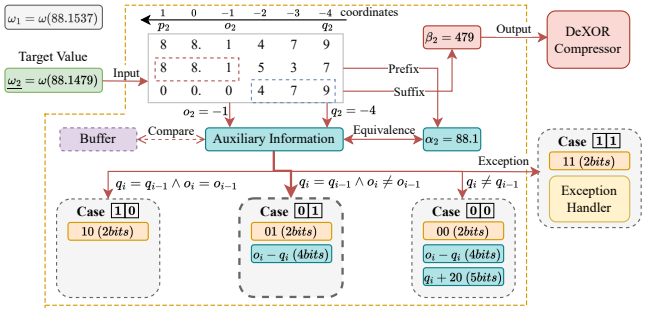


Figure 3: The DeXOR converter keeps coordinates as auxiliary information to replace the extracted prefix, while routing the suffix to the compressor for variable-length encoding.

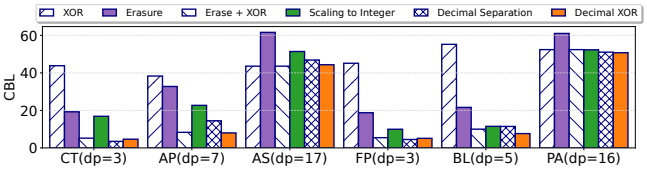


Figure 4: CBL (metadata overhead excluded) reported for *time-series datasets*: City-temp (CT), Air-pressure (AP), and Air-sensor (AS), and *non-time-series datasets*: Food-Price (FP), Blockchain-tr (BL), and POI-lat (PA). Datasets are sorted by data precision (dp) per category.

A decimal value v_X can be represented by a finite segment on its coordinate range, i.e., $v_X = x_p x_{p-1} \cdots x_{q+1} x_q$, where $p \in \mathbb{Z}$ is the **head coordinate** (highest decimal place) and $q \in \mathbb{Z}$ is the **tail coordinate** (lowest decimal place). We have $v_X = \sum_{j=q}^{p} x_j \times 10^j$. The **data precision** dp is defined as the length of meaningful decimal digits, i.e., $dp = p - q + 1$. Referring to Figure 3, for the target value $v = 88.1479$, we have $p = 1$, $q = -4$, and $dp = 1 - (-4) + 1 = 6$.

Given $\underline{v_X}$ the target and v_Y, represented by their decimal digits x_j and y_j ($j \in \mathbb{Z}$), we define the asymmetric DECIMAL XOR as:

$$\underline{v_X} \diamond v_Y = \left(\sum_{j=-\infty}^{\infty} \left(\underline{x_j} \diamond y_j\right) \cdot 10^j \right) \cdot 10^{(-q)}, \tag{3}$$

where $x_j \diamond y_j$ denotes the DECIMAL XOR operation applied to the individual digits x_j and y_j, and is defined as:

$$\underline{x_j} \diamond y_j = \begin{cases} 0 & \text{if } \forall k \in [j, \infty)(x_k = y_k) \\ \underline{x_j} & \text{otherwise} \end{cases} \tag{4}$$

To put it simply, we have $\underline{88.1479} \diamond 88.1537 = 0.0479 \times 10^4 = 479$. This leads to a CBL as small as 9 (see type-③ in Table 1). To further verify its efficacy, we conduct a pilot experiment comparing different converters using six representative benchmark datasets, including both time-series and non-time-series ones (see dataset descriptions in Section 6.1). Figure 4 reports the average CBLs after conversion. By design the CBL captures the **intrinsic compression potential** of each converter and excludes any metadata overhead.

We draw the following key observations: 1) While XOR (type-①) is insensitive to increases in dp, the resulting CBL remains consistently poor (CBL $\geq$ 38). 2) Methods involving precision redundancy (type-① and type-②) degrade in performance as dp increases, with the erasure method nearly failing when dp exceeds 15 bits (CBL $>$ 60); the scaling-to-integers approach performs worse on time-series data compared to non-time-series data as it fails to leverage the smoothness. 3) DECIMAL XOR achieves the best CBL reduction in most cases; this is attributed to its successful integration of the strengths of type-① and type-② methods. Interestingly, DECIMAL XOR performs similarly to XOR in high-dp scenarios (AS and PA); these exceptional cases will be analyzed and addressed in Section 5.

4.2 The DeXOR Converter

While DECIMAL XOR offers significant advantages, implementing it in SLC scenarios is non-trivial. The compression system operates on binary IEEE754 data, requiring costly *binary-decimal conversions* to identify similar prefixes in the decimal space. Moreover, conversion errors can compromise accurate prefix and suffix detection. To resolve these issues, we propose a novel approach in Section 4.2.1, called scaled truncation with error-tolerant rounding, which enables lightweight and precise prefix-suffix identification. Furthermore, Section 4.2.2 introduces an efficient method for storing metadata associated with identified prefixes and suffixes.

4.2.1 Prefix and Suffix Identification. Let α be the **longest common prefix (LCP)** shared between $\underline{v_X}$ and v_Y. We have $v_X - \alpha = \beta'$, where β' is a residual value. Let β be the suffix after DECIMAL XOR and we have $\underline{v_X} \diamond v_Y = \beta = \beta' \times 10^{-q}$ where q is the tail coordinate of v_X. We use $o \geq q$ to denote the LCP coordinate associated with α in v_X. Given the current value $\underline{v_X} = 88.1479$ and previous value $v_Y = 88.1537$ in Figure 3, we have $\alpha = 88.1$, $o = -1$, and $\beta = 479$.

To determine the prefix α and the suffix β, two coordinates are required: (1) the LCP coordinate o, as $\alpha = \sum_{j=o}^{\infty} x_j \times 10^o$; and (2) the tail coordinate q, as $\beta = \beta' \times 10^{-q} = (v_X - \sum_{j=o}^{\infty} x_j \times 10^o) \times 10^{-q}$.

However, v_X and its decimal space coordinates are unknown as compression systems take in only binary values. A naive method is to reconstruct a binary value ω_X into its approximate decimal representation v'_X using Equation 2, convert v'_X to a `String` for prefix-suffix matching with `String`(v'_Y), and then revert the `String` results back to numerical values for further processing. This is inefficient due to the overhead of multiple type conversions, and can fail to accurately identify prefixes and suffixes due to IEEE754 conversion errors ($v'_X \approx v_X$), as discussed later in this section.

To address this issue, we propose a lightweight approach to determine o and q using simple scaling and truncation operators. The approach is formalized through the following two lemmas.

Lemma 1 (LCP Coordinate Judgment). *Any common prefix α' between v_X and v_Y satisfies $\alpha' = \sum_{j=l}^{\infty} x_j \times 10^j =$*

$$\text{trunc}(v_X \times 10^{-l}) \times 10^l = \text{trunc}(v_Y \times 10^{-l}) \times 10^l, \tag{5}$$

where $l \geq q$ is the ending coordinate that indicates α' and trunc$(\cdot)$ *denotes the* truncation *function. The LCP coordinate o is the smallest l that satisfies this condition: $\nexists\, o' < o(\text{trunc}(v_X \times 10^{-o'}) \times 10^{o'} = \text{trunc}(v_Y \times 10^{-o'}) \times 10^{o'})$.*

We omit the proof for its simplicity.

Example 3. *For LCP coordinate $o = -1$ of* 88.1479, trunc$(88.1479 \times 10^1) \times 10^{-1} = 88.1 =$ trunc$(88.1537 \times 10^1) \times 10^{-1}$. *However, for $o' = -2$,* trunc$(88.1479 \times 10^2) \times 10^{-2} = 88.14 \neq 88.15 =$ trunc$(88.1537 \times 10^2) \times 10^{-2}$. *For $o' = 0$, the common prefix is* 88, *but not the longest.*

Lemma 1 suggests that we can iterate from the tail coordinate q to find the smallest coordinate satisfying Equation 5. However, since q is unknown, it must be determined first.

Lemma 2 (Tail Coordinate Judgment). *A coordinate q is the tail coordinate of v_X ($v_X \neq 0$) if and only if:*

$$\begin{aligned}&\left(\text{trunc}(v_X \times 10^{-q}) = v_X \times 10^{-q}\right)\\ &\quad \wedge \left(\text{trunc}(v_X \times 10^{-(q+1)}) \neq v_X \times 10^{-(q+1)}\right)\end{aligned} \tag{6}$$

Proof. ***Sufficiency***: if q is the tail coordinate, the scaled value $v_X \times 10^{-q}$ must be an integer, i.e., $\text{trunc}(v_X \times 10^{-q}) = v_X \times 10^{-q}$. Also, for $q+1$, the scaled value $v_X \times 10^{-(q+1)}$ cannot be an integer, as a smaller scaling introduces a fractional part: $\text{trunc}(v_X \times 10^{-(q+1)}) \neq v_X \times 10^{-(q+1)}$. ***Necessity***: if q satisfies the condition in Equation 6, q is the rightmost coordinate such that $v_X \times 10^{-q}$ is an integer. Thus, q must be the tail coordinate. □

Example 4. *For $v_2 = 88.1479$ with tail $q = -4$: $\text{trunc}(v_2 \times 10^{-(-4)}) = 881479 = v_2 \times 10^{-(-4)}$ whereas $\text{trunc}(v_2 \times 10^{-(-3)}) = 88147 \neq 88147.9 = v_2 \times 10^{-(-3)}$.*

Lemma 2 gives the condition for determining q, but its possible range is too large for an iterative search. However, regularities in input data streams can simplify this. For instance, in the AP dataset, adjacent values often share the same tail coordinate ($q_i = q_{i-1}$) 89% of the time. This observation allows us to reuse the previous tail coordinate q_{i-1} as the initial point for the tail coordinate search.

So far, we have not addressed **IEEE754 conversion errors**. Instead of obtaining the exact values v_X and v_Y, we work with their approximations v'_X and v'_Y. Moreover, scaling operations like $v'_X \times 10^{-j}$ (see Equations 5 and 6) introduce additional errors as 10^{-j} is also represented as a floating-point number. As a result, conditions like $\text{trunc}(v'_X \times 10^{-q}) = v'_X \times 10^{-q}$ in Lemma 2 may fail due to minor rounding errors. To address this, we introduce a small tolerance Δ, typically set to 10^{-6}. If $|\text{trunc}(v'_X \times 10^{-q}) - v'_X \times 10^{-q}| < \Delta$, we treat the condition as satisfied. The same tolerance is applied to the equality condition in Lemma 1.

Example 5. *Assume $v_2 = 88.1479$ with its reconstructed value $v'_2 = 88.14790\cdots7\cdots$. Scaling introduces small rounding errors, causing exact equality to fail: $v'_2 \times 10^4 = 881479.0\cdots1\cdots \neq 881479 = \text{trunc}(v'_2 \times 10^4)$. As a result, $v'_2.q$ may differ from the actual $v_2.q = -4$. However, the difference $|v'_2 \times 10^4 - \text{trunc}(v'_2 \times 10^4)| = 0.0000000001\cdots < \Delta$. By allowing this tolerance, we ensure that $v'_2.q = v_2.q = -4$.*

The scaled truncation with error-tolerant rounding is described in Algorithm 1. Here, q_i is determined via a locality-aware heuristic starting from q_{i-1}, converging within two steps. If $|v'_i \times 10^{-q_{i-1}} - \text{trunc}(v'_i \times 10^{-q_{i-1}})| < \Delta$ then $q_i \geq q_{i-1}$, and a forward search is performed until Lemma 2 is met. Otherwise, a backward search finds the correct q_i. And the search for o_i starts from q_i and continues until the first common prefix is identified.The longest common prefix α_i and the suffix β_i are extracted for processing (lines 4–5).

4.2.2 Efficient Metadata Storage. Figure 3 depicts the procedure after identifying the LCP α_i and suffix β_i. The prefix α_i is excluded from storage, as it can be derived using the LCP coordinate o_i and the previous value v'_{i-1}. Storing o_i is far more space-efficient than storing α_i. Conversely, β_i is passed to the DeXOR compressor for variable-length encoding (detailed in Section 4.3). Note that q_i must

Algorithm 1 `get_prefix_and_suffix` (current value ω_i, previous tail coordinate q_{i-1})

1: $v'_i \leftarrow$ `binary_to_decimal`(ω_i) ▷ Equation 2
2: Tail coordinate $q_i \leftarrow$ `get_tail`(v'_i, q_{i-1})
3: LCP coordinate $o_i \leftarrow$ `get_LCP`(v'_i, q_i)
4: Prefix $\alpha_i \leftarrow \lfloor v'_i \times 10^{-o_i} \rfloor \times 10^{o_i}$
5: Suffix $\beta_i \leftarrow$ `round`$((v'_i - \alpha_i) \times 10^{-q_i})$
6: **return** $q_i, o_i, \alpha_i, \beta_i$

be stored to scale β_i back to its decimal value in decompression, e.g., $479 \times 10^{-4} = 0.0479$. Thus, both o_i and q_i are stored as metadata using auxiliary bits.

Assuming the decimal values fall within a coordinate range: $-20 \leq q \leq p \leq 11$ [1], q_i and o_i each require 5 bits to cover the interval of 31. To minimize the sign overhead, an offset of 20 is added to make q_i and o_i non-negative. Furthermore, instead of storing o_i, we store $(o_i - q_i)$, which typically falls within $[0, 15]$ and only requires 4 bits. If $(o_i - q_i) > 15$, the value v_i is treated as high precision and directed to the exception handler in Section 5.

Importantly, o_i and q_i (and $(o_i - q_i)$) can often be reused without storage, as they frequently match the values of the previous element. A two-bit control code is used to indicate the following reuse cases:

- **Case** `1|0` **(**$q_i = q_{i-1} \wedge o_i = o_{i-1}$**)**: Only 2 bits are needed to indicate reuse of both q_{i-1} and o_{i-1}.
- **Case** `0|1` **(**$q_i = q_{i-1} \wedge o_i \neq o_{i-1}$**)**: In addition to the control code, 4 bits are used to store $(o_i - q_i)$, while 5 bits for q_i are saved.
- **Case** `0|0` **(**$q_i \neq q_{i-1}$**)**: 2 bits are used for the control code, 5 bits for q_i, and 4 bits for $(o_i - q_i)$.

We do not distinguish whether $o_i = o_{i-1}$ when $q_i \neq q_{i-1}$, as $(o_{i-1} - q_{i-1})$ is rarely reusable in such cases. Instead, the remaining **Case** `1|1` serves as the entry point for the exception handler.

Example 6. *Referring to Figure 3, the binary stream after the DeXOR converter is* `0|1|0|0|1|1`. *The first two bits are the control code, and* `0|0|1|1` *indicates $o_i - q_i = -1 - (-4) = 3$. Five bits for $(q_i + 20)$ are saved due to the reuse of $(q_{i-1} + 20)$.*

4.3 The DeXOR Compressor

The DeXOR converter extracts the suffix β_i, the most informative part of the value. Since β_i is an integer, its binary representation often contains leading zeros. For simplicity, we refer to the binary representation of β_i without leading zeros as the **binary suffix**. For example, the binary suffix of $\beta_2 = 479$ is `111011111` (see Table 1). The DeXOR compressor then reduces the storage overhead of the binary suffix using variable-length encoding.

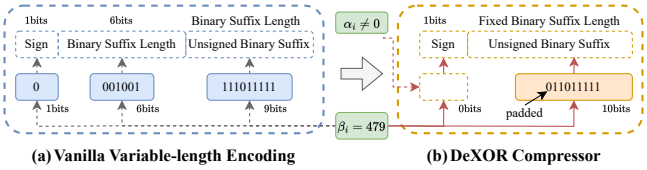
Figure 5(a) illustrates the vanilla variable-length encoding: 1 bit for the sign, 6 bits for the length of the unsigned binary suffix (up to 63 bits), and the required bits for the suffix itself. For example, `111011111` uses 1 + 6 + 9 = 16 bits. To optimize this process, the DeXOR compressor employs the following two strategies.

4.3.1 Optimized Sign Management. Using Lemma 3, we reduce the overhead of sign storage:

Lemma 3 (Sign Consistency). $\forall j, k \in \mathbb{Z},\ x_j \times x_k \geq 0 \implies \alpha \cdot \beta \geq 0$.

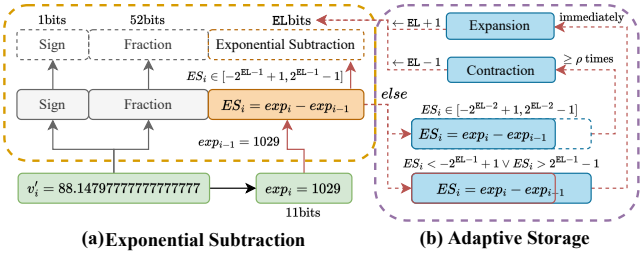
Lemma 3 implies when $\alpha_i \neq 0$, sign of β_i matches that of α_i.

[1] This range is sufficient in practical real-world applications such as geographical information systems and scientific computing [23].



(a) Vanilla Variable-length Encoding (b) DeXOR Compressor

Figure 5: The optimization strategies in the DeXOR compressor. The sign overhead can be waived except $\alpha_i = 0$.



(a) Exponential Subtraction (b) Adaptive Storage

Figure 6: The data flow of the exception handler ($exp_i = 1029$ with the predefined $bias = 1023$); EL is an adjustable variable.

Since α_i is encoded via o_i, the sign of β_i (and its binary suffix) can be omitted. However, in the rare case where $\alpha_i = 0$, the sign of β_i must still be stored in 1 bit.

4.3.2 Optimized Binary Suffix Length. Given β_i as an integer, the length of its unsigned binary suffix is $\ell_i = \lceil \log_2(\text{abs}(\beta_i)+1) \rceil$ where $\text{abs}(\cdot)$ denotes the absolute value. E.g., $\beta_2 = 479$ needs $\ell_2 = 9$ bits to store the unsigned binary suffix. Indeed, the recorded tail coordinate q_i and the LCP coordinate o_i can approximately determine the value β_i, such that $\text{abs}(\beta_i) \in [10^{\delta-1}, 10^{\delta})$ where $\delta = o_i - q_i$. E.g., if $\beta_2 = 479$, then $\beta_2 < 10^{(-1-(-4))} = 10^3$. This means the maximum required bits to represent β_2 is $\bar{\ell}_2 = \lceil \log_2(10^3) \rceil = 10$. By using a fixed length $\bar{\ell}_i = \lceil \log_2(10^\delta) \rceil$, we avoid storing ℓ_i (which requires 6-bit storage as this length is up to 63 bits). For cases where the unsigned binary suffix β_i requires fewer bits than the fixed length, leading zeros are padded to match $\bar{\ell}_i$.

Overall, the optimized scheme incurs 4 bits to store $\delta = o_i - q_i$ (see Section 4.2.2) and $\bar{\ell}_i$ bits for the suffix. In contrast, the original scheme requires 6 bits for ℓ_i and ℓ_i bits for the suffix. The following lemma formalizes the advantage of the fixed-length approach:

Lemma 4 (Fixed Bit Allocation for Unsigned Binary Suffix)**.** *For any $\beta_i \in \mathbb{Z}$, fixed allocation of $\bar{\ell}_i$ bits achieves better compression than variable allocation of ℓ_i, i.e., $\mathbb{E}[(4+\bar{\ell}_i)] < \mathbb{E}[(6+\ell_i)]$.*

We provide a proof sketch for $\mathbb{E}[(\bar{\ell}_i - \ell_i - 2)] < 0$. The derivation conditions on $[10^{\delta-1}, 10^{\delta})$. With $2^{j-1} < 10^{\delta-1} \leq 2^j < 2^{j+1} < 2^{j+2} < 10^\delta$, we have $\delta \in [(j-1)\log_{10} 2 + 1, j \log_{10} 2 + 1]$. We distinguish two cases: the probability $\mathbb{P}_1\{2^{j+3} > 10^\delta\} \approx 0.6781$, and $\mathbb{P}_2\{2^{j+3} \leq 10^\delta\} \approx 0.3219$. We know that $\bar{\ell}_i$ is constant over $[10^{\delta-1}, 10^\delta)$; ℓ_i is a piecewise function, the expectation $\mathbb{E}_1 = \mathbb{E}(\bar{\ell}_i \mid 2^{j+3} > 10^\delta) - \mathbb{E}(\ell_i \mid 2^{j+3} > 10^\delta) - 2 < (j+3) - (j+\frac{7}{6}) - 2 = -\frac{1}{6}$. Similarly, $\mathbb{E}_2 = j + 2 - \mathbb{E}(\ell_i \mid 2^{j+3} \leq 10^\delta) < -\frac{1}{7}$. Finally, $\mathbb{E}[(\bar{\ell}_i - \ell_i - 2)] = \mathbb{E}_1 \times \mathbb{P}_1 + \mathbb{E}_2 \times \mathbb{P}_2 < -\frac{1}{6} \times \mathbb{P}_1 - \frac{1}{7} \times \mathbb{P}_2 \approx -0.159 < 0$.

Example 7. *Referring to Figure 5, the DeXOR compressor reduces the storage for the suffix $\beta_2 = 479$ from 16 bits (vanilla design: 1 bit for sign + 6 bits for $\ell_2 = 9$ and $\ell_2 = 9$ bits for β_2) to 10 bits (0 bit for $\alpha_i \neq 0$ + $\bar{\ell}_2 = 10$ fixed bits allocated for β_2 in Figure 5(b)). Even after accounting for the 4 bits overhead for δ already recorded in the DeXOR converter, the optimization yields a net improvement of 2 bits.*

4.4 Accelerated Decompression

The decompression process extracts and parses segments from the compressed bit stream. For each new value, the DeXOR decompressor reads a two-bit control code(see Section 4.2). If the exception handler is not triggered, q_i and o_i are reused or derived from the metadata bits: q_i scales back the original suffix β'_i, while o_i identifies the LCP α_i. The final value is reconstructed by combining α_i and β'_i. When the control code is 10 and $q_i = q_{i-1} \wedge o_i = o_{i-1}$, we simply reuse the previous LCP ($\alpha_i = \alpha_{i-1}$), avoiding additional computation and accelerating the decompression. In Section 6.2, we examine the efficiency of this scheme.

5 THE DEXOR EXCEPTION HANDLER

Referring back to Figure 4, all converters, including DECIMAL XOR, perform poorly on high-precision datasets (e.g., AS and PA), often matching or underperforming compared to smoothness-based XOR converters. This is because high-precision data reduces precision redundancy, weakening the performance of redundancy-based techniques, including DECIMAL XOR.

Example 8. *High-precision datasets exhibit minimal smoothness and precision redundancy, leading to longer suffixes in DeXOR main pipeline. E.g., with $\delta = o_i - q_i = 16$, the required suffix length $\bar{\ell}_i$ is 54 bits, exceeding the 52 bits used by the fraction in the IEEE754 format (see Equation 1). Such data leaves precision-based methods such as Elf with minimal opportunities for erasure and pushes ALP to overflow.*

While XOR converters rely solely on smoothness and remain relatively robust, their performance is still suboptimal for high-precision data (CBL $\geq$ 38) due to limited inherent smoothness. To address this, the proposed exception handler focuses on enhancing smoothness utilization specifically for high-precision cases. As shown in Figure 6(a), the exception handler targets the exponent (*exp*) in the IEEE754 format. It performs subtraction on adjacent exponents, $exp_i - exp_{i-1}$ (red line), to enable data reuse. The *sign* and *fraction* fields are retained as-is (gray lines) due to their limited smoothness. To optimize storage further, an adaptive storage mechanism dynamically adjusts storage space for the exponential subtraction based on the streaming data. The subtraction and adaptive storage are detailed in Sections 5.1 and 5.2, respectively.

5.1 Exponential Subtraction

In the IEEE754 representation (see Table 1), the fraction part *fraction* typically lacks smoothness, while the exponent *exp* contains more stable and meaningful bits.

We summarize the key properties of the exponent stored in the IEEE754 representation: (1) *wide range of shared values*: A single exponent value represents all real numbers within a given range. For example, if $exp - bias = 6$ (see Equation 2), it represents all values in $[2^6, 2^7) = [64, 128)$, regardless of precision. (2) *low volatility*: exponent values exhibit infrequent changes along the stream. (3) *dense*

lower bits: the lower bits of an exponent's binary representation are compact because exponents are all integers.

Given these properties, the exponent is ideal for compression and forms the foundation for the adaptive storage strategy detailed in Section 5.2. Subtraction is used instead of XOR between consecutive exponents, as subtraction often yields shorter CBLs in high-precision datasets. For high-precision data, the IEEE754 sign bit and fraction are stored in their original binary form, while compression focuses exclusively on the exponent, the most cost-effective component for capturing smoothness.

From these findings, we define **exponential subtraction** as $ES_i = exp_i - exp_{i-1}$. This result is then processed by the adaptive storage module. The required length to store ES_i is defined as: $\ell_i^e = 1 + \lceil log_2(\text{abs}(ES_i) + 1) \rceil$. Unlike ℓ_i defined in Section 4.3.2, ℓ_i^e accounts for 1 extra bit to handle the sign, as ES_i can be negative.

Example 9. *Consider the high-precision number* $88.147977\cdots$ *(11 trailing sevens) in Figure 6, where* $exp_i - bias = 6$*. Typically, the preceding value* exp_{i-1} *equals* exp_i*, resulting in* $ES_i = 0$*. This yields a required bit length of* $\ell_i^e = 1 + \log_2(\lceil 0 + 1 \rceil) = 1$ *for* $ES_i = 0$*. In contrast, encoding* exp_i *with variable-length encoding requires 4 bits (1 bit for the sign and 3 bits for* $(exp_i - bias) = 6$*).*

5.2 Adaptive Storage of Subtraction Results

Compressing only the exponent offers limited savings, as the exponent requires just 11 bits. Additionally, a 4-bit metadata overhead is needed to represent the length of ES_i, which ranges from 1 to 11 bits, further reducing compression ratios.

Example 10. *Consider compressing three exponential subtraction results:* $ES_1 = 3$*,* $ES_2 = 1$*, and* $ES_3 = 1$*. They originate from four distinct exceptional values, which may not be contiguous in the original sequence. Their required lengths are* $\ell_1^e = 3$ *and* $\ell_2^e = \ell_3^e = 2$*. With a fixed metadata approach, 4 bits are needed for each* ℓ_i^e*, plus 53 bits for the residual sign and fraction, resulting in a fixed overhead of 57 bits per entry. The total storage cost is:* $\ell_1^e + \ell_2^e + \ell_3^e + 57 \times 3 = 178$ *bits.*

We propose an adaptive storage mechanism that dynamically adjusts the storage length EL based on historical data, fitting ES_i *elastically*. The storage length EL is maintained as a local variable in the main memory during de/compression. EL varies from 1 to 12 bits, sufficient to cover any $ES_i \in [-2047, 2047]$[2].

Given the current EL, the range $[0, 2^{\text{EL}}-1]$ is used to store integers (i.e., ES_i), with the maximum representable value $2^{\text{EL}} - 1$ reserved to indicate overflow (described below), reducing the usable range by 1. Since ES_i can be negative, a bias of $2^{\text{EL}-1} - 1$ is added to offset the sign of ES_i. The actual stored value becomes: $ES_i + 2^{\text{EL}-1} - 1$, and the valid range of the original ES_i is $[-(2^{\text{EL}-1} - 1), 2^{\text{EL}-1} - 1]$.

After storing the current ES_i, EL is dynamically adjusted for the next ES_{i+1} using two operations, as shown in Figure 6(b):

- **Expansion**: If ES_i is outside the valid range defined by EL, the current EL-length space is filled with $\boxed{1}$s. This is an **overflow** case that triggers the storage of the 64-bit IEEE754 representation. In response, EL is increased by 1 for the next entry ES_{i+1} (hopefully addressing potential overflow).
- **Contraction**: If ES_i consistently fits within a smaller range ($ES_i \in [-(2^{\text{EL}-2} - 1), 2^{\text{EL}-2} - 1]$) for more than ρ consecutive times (with interruptions resetting the count), EL is reduced by 1 (but never below 1), as a smaller EL is likely sufficient. A small ρ risks premature contraction, while a large ρ delays adjustment. We use $\rho = 8$ by default, and Section 6.4 analyzes its impact on high-precision datasets. Note that EL remains unchanged if the ρ-parameterized condition is not met.

[2]This range is required for IEEE754 `double` format, where $exp_i \in [0, 2047]$ [4].

Example 11. *Continuing from Example 10, using adaptive storage, we attempt to store* $ES_1 = 3$ *with an initial length of* $\text{EL} = 1$*. Since* $ES_1 > 2^{\text{EL}} - 1$*, it triggers overflow (all bits in* EL *are* $\boxed{1}$*). The total storage required is 65 bits (1 for* EL *to indicate overflow + 64-bit original IEEE754 representation). After the overflow,* EL *is expanded to 2, which is sufficient to store* $ES_2 \le 2^{2-1} - 1$*. This results in a storage cost of 55 bits (*$\text{EL} = 2$ *to store* ES_2 *+ 53 bits for residual sign and fraction). Similarly,* ES_3 *also requires 55 bits, as no further expansion is needed. The total storage for these three results is* 175 *bits.*

Although this method requires a warm-up phase, EL quickly stabilizes at the optimal value ($\text{EL} \rightarrow \ell_i^e$), allowing the scheme to approach the theoretical compression limit with no overhead. Additionally, the gain of Example 11 over Example 10 becomes more pronounced as the overhead is amortized across more ES_i.

The only drawback is that the maximum compression gain is capped at 11 bits. However, in most extreme cases with $\delta = 16$, the main pipeline requires at least 56 bits, increasing with precision. In contrast, the adaptive storage consistently achieves 56-bit storage (the pipeline's lower bound) regardless of precision, even for high-precision data. Given this, our exception handling process is triggered when $\delta = o_i - q_i > 15$, with q_i and o_i provided by the converter (see Section 4.2), for maximum compression ratios.

5.3 Extended Use of the Exception Handler

The exception handler not only addresses inefficiencies in the main pipeline but also handles exceptions caused by rounding errors in binary-decimal conversions (see Section 2.2). In Section 4.2, we introduce a tolerance for errors within $\Delta = 10^{-6}$, eliminating most rounding inaccuracies. However, two exceptions remain:

(1) **Misclassification of Correct Results**: For example, a value 19.0000005 with a very small residual may be incorrectly treated as 19 when calculating the tail coordinate q, resulting in $q = 0$ instead of its true value $q = -7$. This causes lower coordinate bits to be discarded, and the decompressed value becomes 19.
(2) **Rounding Errors During Decompression**: When reconstructing values using $v_i' = \alpha_i + \beta_i \times 10^{q_i}$, rounding errors, from the scaling operation, may occur even if α_i, β_i, and q_i are correct, leading to slight discrepancies from the original data.

While these errors are rare ($< 0.01\%$), they still require special handling. The exception handler operates entirely in binary, ensuring reliability and enabling full restoration of the original binary representation. Though it cannot match the high compression ratio of the main pipeline, it guarantees decompression accuracy with negligible impact on overall compression due to the rarity of anomalies. The corresponding condition is given as $v_i' \neq \alpha_i + \beta_i \times 10^{q_i}$. This is combined with the condition $o_i - q_i > 15$ (Section 5.2) to immediately identify exceptional cases after Algorithm 1.

Typically, dataset precision is known and consistent, making exception cases predictable. If the precision is known or consecutive exceptions exceed a threshold, we skip storing the case code and

Table 2: Overall comparisons of ACB (↓), compression speed (unit: MB/s, ↑), and decompression speed (unit: MB/s, ↑). Data precision $dp = p - q + 1$ reflects the average information per value. Datasets are divided into low-dp ($dp \leq 7$) (marked in blue) and high-dp (in red). Camel [49] works losslessly only in low-dp datasets.

	Datasets	Time-Series Datasets with Ascending dp														Non-Time-Series Datasets with Ascending dp								GEOMEAN	
		WS	PM	CT	IR	DPT	SUSA	SUK	SDE	AP	BM	BW	BT	BP	AS	FP	EVC	SSD	BL	CA	CO	PA	PO	FULL	low-dp
ACB	Gorilla	52.77	32.12	46.34	39.73	53.71	43.85	35.65	46.18	45.93	49.28	59.79	58.71	53.10	53.06	32.26	59.33	37.24	44.82	63.83	68.02	61.94	68.31	49.09	46.82
	Chimp	51.94	33.31	45.75	43.36	56.69	49.52	43.31	52.18	56.76	54.58	60.13	59.96	58.59	58.33	34.05	57.99	34.99	44.57	61.71	64.20	60.75	64.60	51.18	48.31
	Elf	16.44	13.12	20.78	18.09	27.38	24.78	24.25	27.10	36.81	36.12	39.33	39.81	43.92	62.73	17.82	24.20	18.24	23.84	36.53	40.19	64.39	69.45	29.72	23.18
	Elf+	14.27	9.70	18.35	14.91	24.32	21.86	22.82	25.29	33.37	34.61	37.80	37.21	40.97	63.69	17.30	21.58	16.24	21.21	34.56	39.49	65.34	70.36	27.31	20.57
	Camel	10.36	10.79	11.47	11.59	**12.09**	11.45	**10.18**	13.58	/	/	/	/	/	/	/	**13.85**	20.01	19.97	31.94	34.41	/	/	/	14.86
	DeXOR	**10.35**	**7.12**	**11.33**	**8.01**	13.22	**9.71**	11.59	**12.22**	**14.87**	**19.47**	**30.67**	**29.25**	**25.89**	**52.28**	**12.38**	14.13	**13.27**	**15.00**	**24.76**	**26.53**	**57.86**	**58.70**	**17.82**	**12.70**
Com. Speed	Gorilla	10.06	34.37	9.77	34.42	3.55	18.19	30.80	17.69	28.78	2.97	3.50	25.95	21.16	23.17	**22.68**	6.81	11.66	30.46	25.45	7.41	7.65	**12.16**	13.94	14.95
	Chimp	13.20	35.01	10.58	15.13	16.25	35.63	34.60	**26.57**	31.80	2.25	5.35	20.34	**28.19**	16.57	21.51	**25.50**	0.56	31.92	26.26	30.09	**23.01**	7.91	**15.69**	17.47
	Elf	10.94	13.36	**35.11**	36.87	**18.11**	36.20	18.45	22.77	32.79	2.89	**6.67**	19.96	1.58	**26.90**	4.14	5.07	1.12	34.45	4.98	10.02	9.90	6.37	10.93	13.53
	Elf+	17.01	10.85	11.16	39.70	17.63	39.74	31.35	26.23	**36.51**	**4.12**	4.77	**33.40**	20.86	1.22	21.01	6.15	7.82	**38.34**	29.13	**30.79**	19.16	4.81	15.14	19.92
	Camel	8.46	0.47	6.28	15.25	2.98	9.61	10.44	4.21	/	/	/	/	/	/	/	2.75	0.34	3.66	21.46	3.97	/	/	/	4.23
	DeXOR	**23.86**	**37.08**	32.88	**51.57**	16.74	**49.63**	**37.71**	19.60	33.22	3.61	4.65	21.44	13.49	7.45	15.89	6.77	**19.00**	21.82	19.93	17.49	2.14	1.86	14.94	**24.05**
Decom. Speed	Gorilla	60.61	78.96	52.20	72.09	7.76	62.94	48.26	44.95	58.26	8.37	37.97	48.03	**53.97**	52.49	29.12	21.72	40.45	66.49	54.81	48.25	42.46	**40.66**	41.39	44.82
	Chimp	64.87	**81.76**	**69.16**	73.51	8.29	46.08	21.46	58.53	52.25	**26.50**	34.56	52.79	47.65	43.73	**71.59**	33.08	24.80	**70.72**	54.92	59.81	50.23	37.58	44.41	44.21
	Elf	60.92	69.30	21.75	67.49	26.59	28.57	57.27	23.17	36.11	6.14	27.19	51.41	47.86	51.22	56.38	24.52	15.01	37.78	37.70	7.35	43.99	25.98	31.86	30.97
	Elf+	42.92	44.18	56.77	68.25	33.23	53.84	58.62	51.62	53.78	5.36	5.75	36.37	28.64	48.47	10.05	11.26	1.04	49.29	51.56	48.81	9.67	26.43	25.90	33.11
	Camel	**95.15**	75.00	38.54	95.04	35.78	32.68	83.70	35.83	/	/	/	/	/	/	/	**67.42**	12.51	36.22	56.90	48.56	/	/	/	48.28
	DeXOR	48.28	47.15	63.86	**105.21**	**56.43**	**84.58**	**87.13**	**91.15**	**92.65**	8.89	**40.85**	**68.78**	26.22	**62.16**	62.48	27.25	**82.46**	42.06	**73.83**	**64.61**	**59.51**	24.59	**53.12**	**63.29**

default to using the exception handler alone for compression. This trims a few percent of bits and nearly doubles the compression speed on high-precision, non-time-series datasets, yet the full DeXOR still achieves the best compression on time-series sets. As this "prior-knowledge" mode yields an unfair advantage, Section 6 disables it and reports only on the precision-agnostic configuration.

6 EXPERIMENTAL STUDY

6.1 Experimental Settings

Datasets. We employ one synthetic and 21 real-world datasets from previous studies [33, 35, 49]: 14 time-series and 8 non-time-series sets kept in original order (see Table 2). The latter lack timestamps, and their irregularity challenges smoothness-based converters (type-①); AS (synthetic, noisy) stresses *redundancy-based* converters (type-②); PA/PO supply high-precision geospatial coordinates that strain both types. These datasets contain sequences of up to several hundred million points, with numerical precision (dp) from 3 to 17, value ranges vary from narrow (e.g., [0.01, 1.45]) to broad (e.g., [13,068, 532,323]), and domains spanning meteorology, finance, ecology, mobility, blockchain, and geospatial analysis [3].

Baselines and Metrics. We compare DeXOR with several state-of-the-art SLC schemes, including Gorilla [42], Chimp [37], Camel [49], and the Elf series (Elf [35] and its variant Elf+ [33]). To broaden the comparison, Section 6.6 also covers shemes that use sliding or truncated windows: ALP [8], Chimp_{128} [37], SElf∗ [34], and Elf∗ [34]. To provide a more intuitive measure of compression ratios, we adopt the ***Average Compression Bits* (ACB)** metric from ALP [8]. Compression ratio equals ACB/64 (the IEEE754 representation length). We assess the efficiency in terms of ***compression*** and ***decompression speed***, measured in megabytes per second (MB/s).

Settings and Implementation. Experiments run on an Ubuntu server (Intel Xeon Gold 6348, 512 GB RAM). All SLC schemes are Java-based and `Docker`-containerized for reproducibility and resource isolation; the pipeline supports timestamp-driven streaming. System-level IoTDB [43] integration is reported in Section 6.7.

[3]The resources, statistics, processing scripts for datasets, and DeXOR source code, as well as the appendix, are publicly available at https://github.com/SuDIS-ZJU/DeXOR.

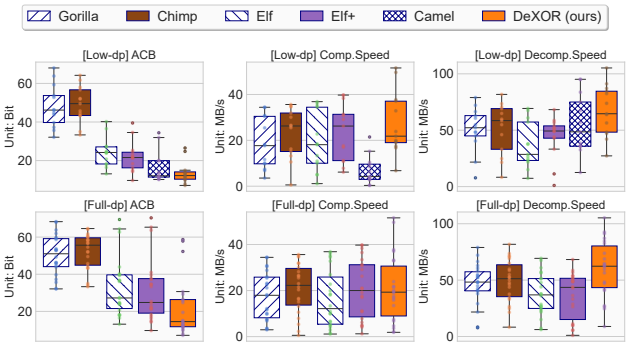


Figure 7: Box-plot visualization of Table 2 under low-dp (top) and full-dp (bottom); boxes show IQR as median, whiskers span 1.5×IQR, and points mark individual dataset outcomes.

6.2 Overall Performance

Table 2 and Figure 7 present the results across the three metrics.

Average Compression Bits. DeXOR consistently delivers the best ACB overall, trailing only Camel by ≤ 2 bits on three low-dp sets (DPT, SUK, EVC), where Camel employs lossy truncation targeting $q \in [-4, -1]$. This specialization, however, incurs accuracy loss or overflow on high-dp data (red) and outlier series (FP). Gorilla and Chimp yield ≥ 30 and generally above 50 bits via XOR and variable-length encoding strategies; Elf and Elf+ exhibit reduced ACBs (≤ 30 and even ≤ 10 in some cases) on low-dp but degrade on high-dp (AS, PA, PO). Overall, only DeXOR (alongside Camel) sustains ACB≤ 20 in most cases, compressing low-dp inputs to ≤ 25% of their original size, and reaching as low as 12.5% at best.

Compression Speed. DeXOR stays within 5% of the fastest baseline (Chimp) and doubles the throughput of its closest ratio rival, Camel. On CA it lags because unstable tail coordinates trigger recomputation (Algorithm 1); yet in the low-dp blue zone ($dp \leq 7$ and $q \in [-4, -1]$) it outruns Elf+ by 21% and Camel by 4.7×. XOR schemes (Chimp, Gorilla) minimize CPU; Elf and Elf+ balance I/O; DeXOR's dynamic heuristics (see Algorithm 1) favour regular series, with only rare irregulars (IR, CA) incurring overhead.

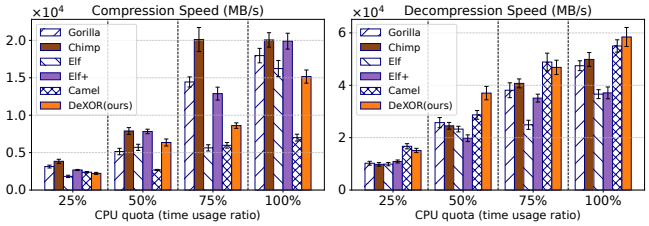


Figure 8: Scalability vs. CPU quotas. Bars show mean over 10 runs, with error bars indicating one standard deviation.

Decompression Speed. DeXOR leads on 13/22 datasets, outpacing Camel by 31% and Gorilla by 41% on average; it remains fastest even where its compression lags (IR, CA) and doubles state-of-the-art throughput on SSD. The only exception is a few low-*dp* sets (e.g., WS) where DeXOR runs at roughly half the speed of Camel.

This superior decompression performance stems from two factors. First, DeXOR reads only the stored coordinate, so unstable patterns do not slow decompression. Second, reusing the previous LCP α_i cuts decompression work by ~50% versus full recomputes (Section 4.4). On low-*dp* datasets (PM, EVC) I/O savings and reusable steps are limited, narrowing the advantage slightly, yet DeXOR remains markedly faster on high-*dp* data.

6.3 Scalability Studies

We assess the scalability of all schemes along two axes — (1) *CPU-quota constraints* and (2) *data-sampling variability*:

CPU-quota constraints (Figure 8). We throttle CPU from 25% to 100% and record speeds. Most schemes scale linearly; XOR-based schemes (Gorilla, Chimp) accelerate fastest at 50–75%. DeXOR improves steadily: second for decompression at 25%, first for both compression and decompression at 50–100%.

Data-sampling variability (Figure 9). We also assess robustness by compressing representative time-series datasets of distinct precisions, under two sampling strategies: (1) *continuous sampling*, which preserves order, and (2) *random sampling*, which disrupts it.

Overall, DeXOR consistently achieves the lowest ACB with minimal variance across both sampling strategies. Random sampling significantly impacts the smoothness of streaming data. XOR-based methods like Gorilla are highly sensitive to smoothness, achieving the poorest ACB at 60% random sampling in the AP dataset. Elf and Elf+ demonstrate better tolerance than simple XOR-based methods. Decimal-aware methods (Camel and DeXOR) perform well on CT and DPT datasets, but Camel suffers a significant drop in ACB (10 ~ 15 bits lost) under random sampling on CT and DPT, while DeXOR remains stable. On WS, the Elf series suffers from disrupted correlations. In contrast, DeXOR maintains ACB stability by effectively utilizing data regularity (Section 4.2.2).

6.4 Parameter Sensitivity Analysis

Section 5.2 introduces a hyperparameter ρ, which acts as a threshold for determining when to contract the bit length EL of the recorded exponential subtraction. Optimizing ρ can slightly improve compression ratios in high-*dp* datasets, such as AS, PA, and PO.

As shown in Figure 10, the three dashed lines represent the scenario where contraction never occurs ($\rho \to +\infty$), resulting in significant redundancy and the worst ACB. Conversely, unconditional contraction ($\rho = 0$) performs better, indicating that uncontrolled expansion has a more detrimental impact than failures caused by premature contraction. This observation suggests favoring smaller ρ values. The effect of ρ on ACB follows a consistent trend across datasets: ACB decreases rapidly to a minimal value and then grows very slowly. For the PA dataset, the ACB stabilizes once $\rho \geq 1$ and only begins to rise noticeably at $\rho \geq 9$. In contrast, for the PO dataset, $\rho = 0$ outperforms $\rho \geq 1$, suggesting frequent attenuation of exponential fluctuations, making $\rho = 0$ a locally optimal choice.

Based on these findings, we set $\rho = 8$ as the default value for all datasets. This choice is likely optimal for PA and PO, while the difference from the local optimum for AS is less than 0.01 bits. In most cases, this default setting suffices.

Table 3: Ablation study with best ACB results in bold. Δ indicates ACB changes, with green for gains and red for losses. Gray sections indicate deactivation of the exception handler.

	Dataset	DeXOR	w/o Excep.		w/o DEC. XOR		w/o Both	
		ACB ↓	ACB ↓	Δ (%)	ACB ↓	Δ (%)	ACB ↓	Δ (%)
Time-Series (TS)	WS	**10.35**	**10.35**	0.00	10.73	-3.67	10.73	-3.67
	PM	**7.12**	**7.12**	0.00	8.72	-22.47	8.72	-22.47
	CT	**11.33**	**11.33**	0.00	14.02	-23.74	14.02	-23.74
	IR	**8.01**	**8.01**	0.00	13.47	-68.16	13.47	-68.16
	DPT	**13.22**	**13.22**	0.00	18.18	-37.52	18.18	-37.52
	SUSA	**9.71**	**9.71**	0.00	19.06	-96.29	19.06	-96.29
	SUK	**11.59**	**11.59**	0.00	21.16	-82.57	21.16	-82.57
	SDE	**12.22**	**12.22**	0.00	21.68	-77.41	21.68	-77.41
	AP	**14.87**	**14.87**	0.00	27.68	-86.15	27.68	-86.15
	BM	**19.47**	**19.47**	0.00	29.33	-50.64	29.33	-50.64
	BW	**30.67**	**30.67**	0.00	32.21	-5.02	32.21	-5.02
	BT	**29.25**	**29.25**	0.00	31.73	-8.48	31.73	-8.48
	BP	**25.89**	**25.89**	0.00	34.35	-32.68	34.35	-32.68
	AS	**52.28**	53.13	-1.63	55.93	-6.98	64.70	-23.76
Non-TS	FP	**12.38**	**12.38**	0.00	16.99	-37.24	16.99	-37.24
	EVC	**14.13**	**14.13**	0.00	14.77	-4.53	14.77	-4.53
	SSD	**13.27**	**13.27**	0.00	17.20	-29.62	17.20	-29.62
	BL	**15.00**	**15.00**	0.00	18.62	-24.13	18.62	-24.13
	CA	**24.76**	**24.76**	0.00	24.97	-0.85	24.97	-0.85
	CO	26.53	26.53	0.00	**26.19**	**1.28**	**26.19**	**1.28**
	PA	57.86	63.97	-10.56	**57.61**	**0.43**	64.70	-11.82
	PO	58.7	65.05	-10.82	**58.49**	**0.36**	65.24	-11.14
	Average	**21.76**	22.36	-2.78	26.05	-19.74	27.08	-24.46

6.5 Ablation Studies

We conduct a module-wise ablation analysis to evaluate the contribution of each component while always retaining the DeXOR compressor. We examine: (1) *w/o Excep.*, which stores extreme-case data directly in IEEE754 format without exception handling; and (2) *w/o DEC. XOR*, which skips eliminating the LCP (Section 4.2) but still stores metadata and the scaled suffix, including $\delta = (o_i - q_i)$, q_i and $\beta_i = v_i \times 10^{-q_i}$. Results are summarized in Table 3.

Effect of the Exception Handler. This plug-in module is primarily triggered for high-*dp* datasets, as indicated by the gray sections in Table 3. While its activation is limited to specific cases, it contributes greatly to ACB performance on datasets like AS, PA, and PO. Without this module, DeXOR experiences a slight drop in performance on the time-series dataset AS and suffers notable degradation on non-time-series datasets such as PA and PO—sometimes performing worse than the original 64-bit IEEE754 representation. However, we confirm that DeXOR remains competitive w.r.t. all baselines in terms of ACB, even without the exception handler. In contrast, after ablating this module, any drawback on low-precision datasets must stem from tolerance failure (see Section 4.2.1), yet the observed value is zero; therefore, tolerance errors are negligible.

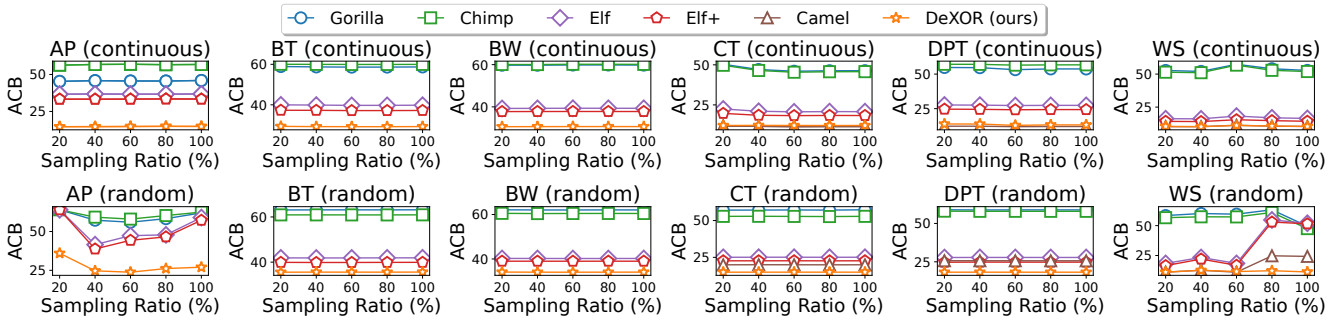


Figure 9: Scalability test on six representative time-series datasets ('continuous' means temporal order preserved while 'random' means contextual continuity disrupted). Camel does not work in high-*dp* datasets AP, BT, and BW.

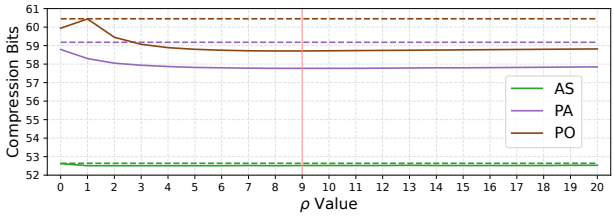


Figure 10: The impact of ρ on ACB for DeXOR. Dashed lines represent the cases when $\rho \to +\infty$.

Effect of DECIMAL XOR. Removing DECIMAL XOR results in severe performance degradation, reducing DeXOR to a scaling-to-integer scheme. As reported in Table 3, this leads to an average ACB loss of 19.74% across all datasets. However, DECIMAL XOR introduces overhead for specific datasets (e.g., PA, PO, CO) when misaligned scales result in front-padding with zeros, increasing required precision. For example, processing $\underline{8.81479} \diamond 88.1479$ as the reference could result in the LCP coordinate $(o_i = 2) > (p_i = 0)$, rendering $\delta = (o_i - q_i = 7) > dp = (p_i - q_i + 1 = 6)$[4]. This reflects a trade-off in our converter: while reusing the previous suffix avoids storing it explicitly, misalignment can occasionally demand higher precision.

Effect of Both Modules. When both modules are removed, DeXOR essentially retains only the functions of metadata reuse (Section 4.2.2) and the compressor (Section 4.3). Even so, Table 3 shows that the remaining components achieve competitive compression ratios compared to Elf, Elf+, and Camel (cf. Table 2).

Two notable observations emerge: (1) On the CO dataset, the stripped-down version of DeXOR achieves slightly better performance than the full version (+1.28%), due to misalignment overhead caused by rare data cases. A similar effect is observed on PA and PO, though the losses are offset by the removal of the exception handler. (2) DECIMAL XOR reduces exceptions by eliminating the LCP, allowing some high-*dp* data to fall outside the bounds of exceptional cases. For example, on the AS dataset, ablation deteriorates ACB by -23.76%, far worse than the combined impact of individual ablations ((-1.63%) + (-6.98%)). This underscores a key advantage of DeXOR: its ability to perform post-judgment of exceptions, enabling more accurate handling of high-*dp* yet smooth data.

[4]Consider a simplified case: $\underline{v_X} = 8.8$ and $v_Y = 88.1$ have different head coordinates $(v_X.p = 0) \neq (v_Y.p = 1)$, implying LCP $\alpha = 0$ at $o = 2 = \max(v_X.p, v_Y.p) + 1$. This pads the suffix to 088 in DECIMAL XOR, which, while correct, is inefficient.

Table 4: Comparison with larger buffer schemes.

Metrics (geomean)	Chimp_{128}	SElf∗	ALP	Elf∗	DeXOR (ours)
AVG. Comp. Bits ↓	31.37	19.55	52.97	**17.55**	17.82
Comp. Speed (MB/s) ↑	12.65	14.54	2.07	**16.06**	14.94
Decomp. Speed (MB/s) ↑	40.27	49.69	38.73	23.41	**53.12**

Table 5: ACB (↓) under pre-processing strategies: original, zero-mean (-Z), and standardization (-S) on four datasets.

Dataset	Gorilla	Chimp	Elf	Elf+	DeXOR (ours)
CT	46.34	45.75	20.78	18.35	**11.33**
CT-Z	47.95	**47.66**	52.51	53.51	57.10
CT-S	**51.48**	52.04	54.97	55.95	55.94
AS	53.06	58.33	62.73	63.69	**52.28**
AS-Z	**51.00**	58.39	60.57	61.56	53.44
AS-S	**53.21**	58.43	62.74	63.68	54.34
FP	32.26	34.05	17.82	17.30	**12.38**
FP-Z	**29.79**	34.69	37.53	38.53	56.05
FP-S	**35.99**	40.90	43.60	44.56	42.20
PA	61.94	60.75	64.39	65.34	**57.86**
PA-Z	65.77	63.69	66.48	67.44	**59.11**
PA-S	67.40	63.89	68.27	69.24	**59.10**

6.6 SLC with Buffering or Pre-processing

Larger Buffer. Our main setting (Section 2.2) restricts access to only the immediately preceding values. In contrast, some schemes leverage larger buffers — e.g., sliding windows (Chimp_{128} [37], SElf∗ [34]) or mini-batches (ALP [8], Elf∗ [34]) — to improve compression. Because these methods require more memory, we report them separately rather than mixing them with the primary baselines in Table 2. Each scheme's sensitivity to buffer size and its target scenarios differ. To ensure fairness, we use each's recommended buffer sizes: $N = 128$ for Chimp_{128}, $N = 1024$ for ALP, and $N = 1000$ for SElf∗ and Elf∗. As shown in Table 4, despite competing against buffer-heavy schemes, DeXOR achieves the fastest decompression and ranks second in both compression ratio and speed, trailing Elf∗ by only 0.27 bits and 1.12 MB/s.

Pre-processing Methods. Zero-mean normalization and 0-1 standarization reduce smoothness, increase entropy, and inject irreversible loss with extra runtime cost: e.g., $50.0 \to -0.0431\cdots$ expands precision requirements that strain each SLC scheme. As Table 5 shows, these pre-processing methods hurt precision-redundancy schemes including DeXOR, while simple XOR schemes (Gorilla, Chimp) benefit. The impact on DeXOR is milder for high-precision, irregular datasets (AS and PA), where it stays competitive. Therefore,

Table 6: All results are produced within Apache IoTDB Tsfile engine, evaluated on three heterogeneous real-world datasets: CT (time-series), FP (non-time-series), and PA (high-precision non-time-series).

Secondary Compression	Algorithm	Average Compression Bits (↓)				Compression Throughput (MB/s, ↑)				Query Latency (ms/1k, ↓)			
		CT	FP	PA	Avg.	CT	FP	PA	Avg.	CT	FP	PA	Avg.
Uncompressed	Sprintz	51.94	52.01	61.85	55.27	1.344	4.160	3.136	2.880	0.67	0.188	0.132	0.330
	Gorilla	58.17	40.28	67.57	55.34	6.016	5.888	**4.096**	**5.205**	0.174	**0.073**	**0.090**	0.112
	DeXOR	**12.91**	**13.96**	**59.45**	**28.78**	**6.144**	**6.016**	3.072	4.949	**0.093**	0.083	0.106	**0.094**
Lz4	Sprintz	37.57	34.93	60.95	44.48	4.032	5.696	3.264	4.331	0.105	0.118	0.127	0.116
	Gorilla	29.00	27.97	66.70	41.22	5.120	5.760	**4.480**	**5.120**	0.095	**0.081**	0.101	0.092
	DeXOR	**11.56**	**12.83**	**58.54**	**27.64**	**5.952**	**5.824**	3.008	4.928	**0.079**	0.086	**0.090**	**0.085**
Snappy	Sprintz	36.70	34.17	60.72	43.86	3.840	5.632	**4.032**	4.501	0.119	0.099	0.091	0.103
	Gorilla	30.13	28.89	66.44	41.82	5.120	**6.208**	3.968	5.099	0.140	**0.077**	**0.080**	0.098
	DeXOR	**11.69**	**12.83**	**58.32**	**27.61**	**6.272**	5.760	3.328	**5.120**	**0.073**	0.092	0.092	**0.085**

Table 7: Compression and query performance of Gorilla and DeXOR in IoTDB on reprsentative vector datasets.

Metric	Algorithm	Datasets		
		SIFT	WR	WW
Average Compression Bits (↓)	Gorilla	19.65	42.84	43.03
	DeXOR	**12.99**	**13.47**	**13.11**
Compression Throughput (MB/s, ↑)	Gorilla	13.510	4.933	6.958
	DeXOR	**18.493**	**11.167**	**14.724**
Query Latency (ms/1k, ↓)	Gorilla	0.017	0.014	0.005
	DeXOR	**0.0002**	**0.010**	**0.004**

we recommend avoiding transformations that inflate precision or introduce loss before DeXOR; if normalization is required, it should be applied post-DeXOR's decompression for better performance.

6.7 Real-World Time Series Database Testing

Apache IoTDB's TsFile [43] has a pluggable encoding layer enabling DeXOR integration without system changes. On datasets (CT, FP, PA), we compare DeXOR against IoTDB's Gorilla and Sprintz schemes, each paired with Lz4 [3] or Snappy [6] for secondary compression. Table 6 shows that DeXOR consistently outperforms both (up to 4.5×), retains ingestion throughput, and answers 1,000 single-value queries faster; the secondary compressors Lz4 and Snappy yield only $< 2\%$ extra space reductions and negligible runtime overhead and are thus optional for DeXOR.

Support for Vector Data Compression. While the main study covers 22 one-dimensional sequences, vectors are also used widely. To determine whether DeXOR can be reused as-is, we pick two representative sets: SIFT [2] (10k vectors, 128-dim computer vision) and Wine-Quality [15] (4,898 vectors, 11-dim red/white wines—WR, WW), with each dimension compressed independently. A query retrieves a vector record, and the latency is measured over 1,000 queries. As shown in Table 7, without vector-specific redesign, DeXOR still achieves $\sim 3.5\times$ better compression, $\sim 2\times$ higher throughput, and much lower latency versus dimension-wise Gorilla, indicating the utility of DeXOR for vector data.

7 RELATED WORK

General-purpose Compression. LZ4 [3], LZMA [1], Snappy [6], XZ [5], and Brotli [9] exploit redundancy via Huffman and dictionaries, achieving better compression on static data but incurring decompress-recompress cycles that hinder real-time use.

Streaming Lossy Compression embeds data-cleaning techniques [17, 26, 39] and error-bounded mechanisms [12, 16, 32, 38, 48] to trade precision for compression ratio. Though we target lossless compression, our techniques readily extend to lossy regimes.

Learned Compression [16, 19, 24, 39, 48] predicts values and encodes high-precision stochastic residuals, so most are lossy (e.g., SZ [16], MOST [48]). NeaTS [19] stands out by providing a *configurable* switch, enabling both lossy and lossless compression. Strongly model-dependent and hardware-demanding, these techniques are promising but need further exploration.

Lossless Compression has been developed for **streaming data** [8, 10, 11, 13, 20–22, 24, 25, 27, 33–35, 37, 41, 42, 45, 46, 50] and **floating-point data** [8, 27, 33–35, 42]. Compressing streaming data often utilizes contextual statistics. Batch schemes like ALP [8] and Elf∗ [34] employ off-line techniques, including entropy and dictionary coding, trading higher latency for better compression. For floating-point data, IEEE754 encoding issues are mitigated by techniques like XOR [42] and scaling-to-integer [8]. DeXOR extends these techniques to support integer compression and includes a configurable exception handler to adapt online to diverse datasets.

Vector Data Compression. Low-dimensional vectors can be compressed dimension-wise with SIMD parallelism [8]; Compressing high-dimensional vectors (e.g., DeltaPQ [44]) relies on quantization. Compressing sequential values in a vector is intuitive (see our implementation in Section 6.7), while exploiting inter-vector smoothness for lossless compression remains an open problem.

8 CONCLUSION

We presents DeXOR, a streaming lossless compression scheme for floating-point data that applies lightweight XOR-like conversions in decimal space. By exploiting common decimal prefixes and suffixes, it unifies smoothness- and redundancy-based techniques while mitigating IEEE754 conversion errors, anomalies, and high-precision cases. DeXOR delivers state-of-the-art performance and flexibility for decimal-system numerical compression.

Clear directions exist for building a DeXOR family for diverse scenarios: (1) Batch-mode DeXOR: Employing advanced parallelism (e.g., SIMD vectorization) for high-performance, native vector data compression. (2) Model-based lossless DeXOR: Refining DECIMAL XOR for compressing prediction residuals from lightweight time-series forecasting models. (3) Lossy DeXOR: Creating powerful, error-bounded variants adapted to specific data conditions.

ACKNOWLEDGMENTS

This work was supported by the NSFC Grant No. 62402420, the "Pioneer" R&D Program of Zhejiang Province (No. 2024C01021), and the NSFC Grant No. 62472377.

REFERENCES

[1] 2001. LZMA compression/decompression in golang. https://code.google.com/archive/p/lzma/.
[2] 2010. ANN_SIFT10K (siftsmall). http://corpus-texmex.irisa.fr/
[3] 2013. LZ4: Extremely Fast Compression algorithm. https://github.com/lz4/lz4.
[4] 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84.
[5] 2022. XZ Utils. https://github.com/tukaani-project/xz.
[6] 2023. Snappy: A fast compressor/decompressor. https://github.com/google/snappy.
[7] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating Compression and Execution in Column-Oriented Database Systems. In *SIGMOD*. 671–682.
[8] Azim Afroozeh, Leonardo X. Kuffo, and Peter Boncz. 2023. ALP: Adaptive Lossless floating-Point Compression. *SIGMOD* 1, 4 (2023), 1–26.
[9] Jyrki Alakuijala, Andrea Farruggia, Paolo Ferragina, Eugene Kliuchnikov, Robert Obryk, Zoltan Szabadka, and Lode Vandevenne. 2018. Brotli: A general-purpose data compressor. *TOIS* 37, 1 (2018), 1–30.
[10] Noura Alghamdi, Liang Zhang, Huayi Zhang, Elke A. Rundensteiner, and Mohamed Y. Eltabakh. 2020. ChainLink: Indexing Big Time Series Data For Long Subsequence Matching. In *ICDE*. 529–540.
[11] Davis Blalock, Samuel Madden, and John Guttag. 2018. Sprintz: Time Series Compression for the Internet of Things. *IMWUT* 2, 3 (2018), 1–23.
[12] Shubham Chandak, Kedar Tatwawadi, Chengtao Wen, Lingyun Wang, Juan Aparicio Ojea, and Tsachy Weissman. 2020. LFZip: Lossy Compression of Multivariate Floating-Point Time Series Data via Improved Prediction. In *DCC*. 342–351.
[13] Giacomo Chiarot and Claudio Silvestri. 2023. Time Series Compression Survey. *Comput. Surveys* 55, 10 (2023), 1–32.
[14] ClickHouse. 2025. Compression in ClickHouse. https://clickhouse.com/docs/data-compression/compression-in-clickhouse.
[15] Paulo Cortez, A. Cerdeira, F. Almeida, T. Matos, and J. Reis. 2009. Wine Quality. https://archive.ics.uci.edu/dataset/186/wine+quality
[16] Sheng Di and Franck Cappello. 2016. Fast Error-Bounded Lossy HPC Data Compression with SZ. In *IPDPS*. 730–739.
[17] Eugene Fink and Harith Suman Gandhi. 2011. Compression of time series by extracting major extrema. *JETAI* 23, 2 (2011), 255–270.
[18] Google. 2018. The Need for Mobile Speed: How Mobile Latency Impacts Publisher Revenue. Google Research Whitepaper.
[19] Andrea Guerra, Giorgio Vinciguerra, Antonio Boffa, and Paolo Ferragina. 2025. Learned Compression of Nonlinear Time Series with Random Access. In *ICDE*. 1579–1592.
[20] Adrián Gómez-Brandón, José R. Paramá, Kevin Villalobos, Arantza Illarramendi, and Nieves R. Brisaboa. 2021. Lossless compression of industrial time series with direct access. *Computers in Industry* 132 (2021), 103503.
[21] Huamin Chen, Jian Li, and P. Mohapatra. 2004. RACE: time series compression with rate adaptivity and error bound for sensor networks. In *MASS*. 124–133.
[22] Sang-Ho Hwang, Kyung-Min Kim, Sungho Kim, and Jong Wook Kwak. 2023. Lossless Data Compression for Time-Series Sensor Data Based on Dynamic Bit Packing. *Sensors* 23, 20 (2023), 8575.
[23] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. 2017. Time series management systems: A survey. *TKDE* 29, 11 (2017), 2581–2600.
[24] Nan Jiang, Qingping Xiang, Hongzhi Wang, and Bo Zheng. 2023. Time series compression based on reinforcement learning. *Information Sciences* 648 (2023), 119490.
[25] Abdelouahab Khelifati, Mourad Khayati, and Philippe Cudre-Mauroux. 2019. CORAD: Correlation-Aware Compression of Massive Time Series using Sparse Dictionary Coding. In *Big Data*. 2289–2298.
[26] Lucie Klus, Roman Klus, Elena Simona Lohan, Carlos Granell, Jukka Talvitie, Mikko Valkama, and Jari Nurmi. 2021. Direct Lightweight Temporal Compression for Wearable Sensor Data. *Sensors Letters* 5, 2 (2021), 1–4.
[27] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *SIGMOD* 1, 2 (2023), 1–26.
[28] Zhichen Lai, Huan Li, Dalin Zhang, Yan Zhao, Weizhu Qian, and Christian S. Jensen. 2024. E2USD: Efficient-yet-effective Unsupervised State Detection for Multivariate Time Series. In *WWW*. 3010–3021.
[29] Zhichen Lai, Dalin Zhang, Huan Li, Christian S. Jensen, Hua Lu, and Yan Zhao. 2023. LightCTS: A Lightweight Framework for Correlated Time Series Forecasting. *Proc. ACM Manag. Data* 1, 2 (2023), 125:1–125:26.
[30] Zhichen Lai, Dalin Zhang, Huan Li, Dongxiang Zhang, Hua Lu, and Christian S. Jensen. 2024. ReCTSi: Resource-efficient Correlated Time Series Imputation via Decoupled Pattern Learning and Completeness-aware Attentions. In *KDD*. 1474–1483.
[31] Huan Li, Hua Lu, Christian S. Jensen, Bo Tang, and Muhammad Aamir Cheema. 2022. Spatial data quality in the Internet of Things: Management, exploitation, and prospects. *CSUR* 55, 3 (2022), 1–41.
[32] 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. *SIGMOD* 3, 3 (2025), 1–27.
[33] Ruiyuan Li, Zheng Li, Yi Wu, Chao Chen, Songtao Guo, Ming Zhang, and Yu Zheng. 2023. Erasing-based lossless compression method for streaming floating-point time series. *ArXiv* abs/2306.16053 (2023).
[34] Ruiyuan Li, Zheng Li, Yi Wu, Chao Chen, Tong Liu, and Yu Zheng. 2025. Adaptive Encoding Strategies for Lossless Floating-Point Compression. *IoTJ* (2025), 1–1.
[35] Ruiyuan Li, Zheng Li, Yi Wu, Chao Chen, and Yu Zheng. 2023. Elf: Erasing-Based Lossless Floating-Point Compression. *PVLDB* 16, 7 (2023), 1763–1776.
[36] Xiao Li, Huan Li, Hua Lu, Christian S. Jensen, Varun Pandey, and Volker Markl. 2023. Missing Value Imputation for Multi-attribute Sensor Data Streams via Message Propagation. *PVLDB* 17, 3 (2023), 345–358.
[37] Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. 2022. Chimp: efficient lossless floating point compression for time series databases. *PVLDB* 15, 11 (2022), 3058–3070.
[38] Chunwei Liu, Hao Jiang, John Paparrizos, and Aaron J Elmore. 2021. Decomposed bounded floats for fast compression and queries. *PVLDB* 14, 11 (2021), 2586–2598.
[39] Jinxin Liu, Petar Djukic, Michel Kulhandjian, and Burak Kantarci. 2024. Deep Dict: Deep Learning-based Lossy Time Series Compressor for IoT Data. In *ICC*. 1–6.
[40] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. 2022. Operational Characteristics of SSDs in Enterprise Storage Systems: A Large-Scale Field Study. In *USENIX*. 165–180.
[41] Ningting Pan, Peng Wang, Jiaye Wu, and Wei Wang. 2018. MTSC: An Effective Multiple Time Series Compressing Approach. In *DEXA*. 267–282.
[42] 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. *PVLDB* 8, 12 (2015), 1816–1827.
[43] Chen Wang, Xiangdong Huang, Jialin Qiao, Tian Jiang, Lei Rui, Jinrui Zhang, Rong Kang, Julian Feinauer, Kevin A. McGrail, Peng Wang, Diaohan Luo, Jun Yuan, Jianmin Wang, and Jiaguang Sun. 2020. Apache IoTDB: time-series database for internet of things. *PVLDB* 13, 12 (2020), 2901–2904.
[44] Runhui Wang and Dong Deng. 2020. DeltaPQ: lossless product quantization code compression for high dimensional similarity search. *PVLDB* 13, 13 (2020), 3603–3616.
[45] Tianrui Xia, Jinzhao Xiao, Yuxiang Huang, Changyu Hu, Shaoxu Song, Xiangdong Huang, and Jianmin Wang. 2024. Time series data encoding in Apache IoTDB: comparative analysis and recommendation. *VLDBJ* 33, 3 (2024), 727–752.
[46] Lei Yan, Jiayu Han, Runnan Xu, and Zuyi Li. 2021. Model-Free Lossless Data Compression for Real-Time Low-Latency Transmission in Smart Grids. *TSG* 12, 3 (2021), 2601–2610.
[47] Xianjun Yang, Xiaofeng Tao, Eryk Dutkiewicz, Xiaojing Huang, Y. Jay Guo, and Qimei Cui. 2013. Energy-Efficient Distributed Data Storage for Wireless Sensor Networks Based on Compressed Sensing and Network Coding. *IEEE TWC* 12, 10 (2013), 5087–5099.
[48] Zehai Yang and Shimin Chen. 2023. MOST: Model-Based Compression with Outlier Storage for Time Series Data. *SIGMOD* 1, 4 (2023), 1–29.
[49] Yuanyuan Yao, Lu Chen, Ziquan Fang, Yunjun Gao, Christian S. Jensen, and Tianyi Li. 2024. Camel: Efficient Compression of Floating-Point Time Series. *SIGMOD* 2, 6 (2024), 1–26.
[50] Xinyang Yu, Yanqing Peng, Feifei Li, Sheng Wang, Xiaowei Shen, Huijun Mai, and Yue Xie. 2020. Two-Level Data Compression using Machine Learning in Time Series Database. In *ICDE*. 1333–1344.