A MORE TECHNICAL DETAILS

A.1 Data Ranges and Examples of `float` and `double` Numbers in IEEE754 Representation

In the IEEE754 standard, floating-point representation consists of three main components: the sign bit, the exponent bits, and the significand (or fraction) bits. Please refer back to Section 2.1 for the definition of IEEE754 floating-number representation. The ranges and interpretations of the exponent bits differ for ***single precision (32 bits)*** and ***double precision (64 bits)***, as presented in Table 5.

Table 5: Structure and value ranges of the IEEE754 standard.

Precision	*sign*	*exp*	*fraction*	Total	Absolute Value Range
`float` (*bias* = 127)	1 bit	8 bits	23 bits	32 bits	$[2^{-149}, 2^{128} - 2^{104}]$
`double` (*bias* = 1023)	1 bit	11 bits	52 bits	64 bits	$[2^{-1074}, 2^{1024} - 2^{971}]$

For `single` numbers, *exp* is 8 bits, with a range $[1, 254]$, while for `double` numbers, *exp* is 11 bits, with a range $[1, 2046]$. When the exponent falls within these ranges, the floating-point number is in *normalized* form. In normalized form, the actual exponent value is adjusted by a bias: 127 for single precision and 1023 for double precision. Thus, the actual exponent field is within $[-126, 127]$ for `single` and $[-1022, 1023]$ for `double`.

When *exp* is equal to 0, the floating-point number is in *denormalized* (or subnormal) form. In single precision, *exp* = 0 indicates that the number is less than 2^{-126}, and its value is represented as:

$$v_i = 2^{-126} \times fraction. \tag{7}$$

In denormalized form, the significand is represented directly without the implicit leading $\boxed{1}$, which is used in normalized form. In this sense, the smallest value for *fraction* is 2^{-23} as *fraction* occupies 23 bits in `single`. Therefore, the smallest denormalized number in `single` is $v_i = 2^{-126} \times 2^{-23} = 2^{-149}$, while the smallest normalized number in `single` is 2^{-126}, with the fraction part equals to (1+0). If the absolute value of a number is smaller than this minimum normalized value, it is considered to be 0.

Taking single precision as an example, when *exp* = 0, it implies that the number is less than 2^{-126}, at which point $v_i = 2^{-126} \times fraction$. At this juncture, the magnitude of the number solely depends on the significand, and *fraction* no longer requires the addition of 1. Therefore, the smallest single-precision floating-point number is 2^{-149}, and when the absolute value of a certain number is smaller than it, it will be regarded as 0. Correspondingly, when *exp* = 255, depending on the significand, it is classified into two scenarios: when *fraction* = 0, the number is infinity (Inf); and when *fraction* ≠ 0, the number is not a number (NaN).

Example 12. *In single-precision format, a numerical value* $v_i = 1.25$ *is represented as* $1.25 = (+1) \times 2^0 \times 1.25$*. Thus, the bit of sign is 0 (positive), exp is 127 (with bias = 127 added), and fraction is 0.25 (with 1 subtracted). To convert them into binaries, we have sign as* $(0)_2$*, exp as* $(127)_{10} = (01111111)_2$*, and a 23-digit significand of* $(0.25)_{10} = (0.01)_2$*. Since fraction must be within* $[0, 1)$*, we can always omit the first zero and have* $(0.01)_2$ *stored as* $01\{0\cdots\}_{21\ bits}$ *rather than* $001\{0\cdots\}_{20\ bits}$*.*

A.2 Extreme Case Faced by SLC Schemes

The high-precision dataset poses the most formidable challenge to existing schemes, as exemplified in Table 6.

Table 6: High precision data.

No.	Value	*sign*	*exp*	*fraction*
1	v_1 : 0.6163339365209309	0	01111111110	0011101110010000000111110010100110001011011011111111
2	v_2 : 0.6163339365216953	0	01111111110	0011101110010000000111110010100110001101000111100100
3	v_3 : 0.6163339365216972	0	01111111110	0011101110010000000111110010100110001101000111110101
4	v_4 : 0.9241518389309976	0	01111111110	1101100100101010011011100000100101111110011010001111
5	v_5 : 0.92	0	01111111110	1101011100001010001111010111000010100011110101110001
6	$v_1 \oplus v_2$	0	00000000000	0000000000000000000000000000000000000110011100011011
7	$v_3 \oplus v_4$	0	00000000000	1110001010111010011100010010000011110011011101111010
8	$v_4 \oplus v_5$	0	00000000000	0000111000100000010100110111100111011101101111111110
9	$v_5' = 0.9140625$	0	01111111110	1101010000000000000000000000000000000000000000000000
10	$v_4 \oplus v_5'$	0	00000000000	0000110100101010011011100000100101111110011010001111

Example 13. *Table 6 provides examples of high-precision data. In these cases, most values include 16 decimal places, which prevents Elf from achieving significant erasure. The only successful erasure by Elf occurs with* $v_5 = 0.92$*, resulting in* $v_5' = 0.9140625$ *(cf. line 9). However, the subsequent XOR operation* $v_4 \oplus v_5'$ *performed by Elf (rows 9 and 10) negates the impact of the earlier erasure.*

When ALP processes the values by converting them into integers, such as transforming v_1 *into* 6163339365209309*, the result occupies 54 binary bits in a signed integer format. Moreover, since ALP does not always obtain the optimal value for each record, the actual transformed values may become larger, ultimately exceeding the range that a 64-bit signed integer can represent, leading to exceptions. As shown in line 6* ($v_1 \oplus v_2$)*, XOR-based algorithms can deliver satisfactory results; however, in other cases similar to line 6, their performance is less effective.*

A.3 Detailed Framework and Algorithms

As a supplement to the Figure 2, a more detailed framework is illustrated in Figure 10. The binary input ω_i is first processed by the DeXOR converter (Algorithms 1 and 2):

Algorithm 2 `Metadata_Storage` (binary output stream *out*, current and previous tail coordinates q_i, q_{i-1}, current and previous LCP coordinates o_i, o_{i-1}, prefix α_i, suffix β_i, value v_i' and binary ω_i)

```
1: $\delta \leftarrow o_i - q_i$
2: if $\delta > 15 \vee v_i' \neq \alpha_i + \beta_i \times 10^{-q}$ then
3:     $out.\text{write}(3, 2)$                ▷ Exception Case $\boxed{1}\boxed{1}$
4:     Exception Handler($\omega_i$)
5: else if $q_i = q_{i-1} \wedge o_i = o_{i-1}$ then
6:     $out.\text{write}(2, 2)$                ▷ Case $\boxed{1}\boxed{0}$
7: else if $ed_i = q'$ then
8:     $out.\text{write}(1, 2)$                ▷ Case $\boxed{0}\boxed{1}$
9:     $out.\text{write}(\delta, 4)$
10: else
11:     $out.\text{write}(0, 2)$               ▷ Case $\boxed{0}\boxed{0}$
12:     $out.\text{write}(\delta, 4)$
13:     $out.\text{write}(q_i + 20, 5)$        ▷ $q_i \in [-20, 11]$
```

The inputs are then immediately processed by Algorithm 1 (see Section 4.2.1). Then Algorithm 2 serves as a post-processing step, which includes the detection of exceptions (line 2 of Algorithm 2, as described at the end of Section 5.3) and efficient metadata storage (see Section 4.2.2).

We utilize a 2-bit case code to classify which scenario has been entered, encompassing three successful cases ($\boxed{0}\boxed{0}$, $\boxed{0}\boxed{1}$, $\boxed{1}\boxed{0}$) and one for exceptions ($\boxed{1}\boxed{1}$).

For those successful cases, the suffix β_i from the results of Algorithm 1 must also be stored. By employing Lemma 3 (Sign Consistency) and the optimized binary suffix lengths (Section 4.3.2), DeXOR achieves efficient outcomes via compressor (Algorithm 3).

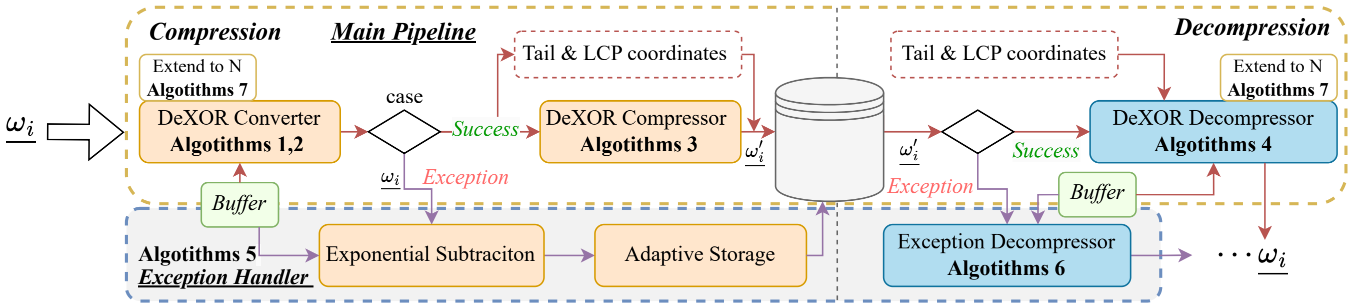

Figure 10: The processing pipeline consists of a vanilla design with a DECIMAL XOR-based converter (reverter) and a specialized compressor (decompressor). An Exception Handler (gray section) is included to process failure cases.

Algorithm 3 DeXOR_Compressor (out, δ, prefix α_i, suffix β_i)

```
1: $\bar{\ell}_i \leftarrow \lceil \delta \times \log_2(10) \rceil$        ▷ optimized binary suffix length
2: if $\alpha_i = 0$ then        ▷ using Lemma 3 (Sign Consistency)
3:     $out$.write $Bit(\beta_i < 0)$
4: $out$.write(abs($\beta_i$), $\bar{\ell}_i$)
```

Algorithm 4 Decompressor (binary stream from storage stream in, values in global Buffer $v'_{i-1}, \alpha_{i-1}, q_{i-1}, o_{i-1}$)

```
1: $code \leftarrow in$.readInt(2)
2: if $code = 3$ then        ▷ Exception Case $\boxed{1}\boxed{1}$
3:     return $Exception\_Decompressor()$
4: else
5:     $\alpha_i, q_i, o_i \leftarrow$ DeXOR_Reverter($in$)
6:     $v'_i \leftarrow$ DeXOR_Decompressor($in, \alpha_i, o_i, q_i$)
7:     Buffer $\leftarrow v'_i, \alpha_i, q_i, o_i$
8:     $\omega_i \leftarrow$ decimal_to_binary($v'_i$)
9:     return $\omega_i$
10: function DeXOR_Reverter($in$, $code$)
11:     if $code = 0$ then
12:         $q_i \leftarrow in$.readInt(5) $- 20$
13:     else
14:         $q_i \leftarrow q_{i-1}$
15:     if $code < 2$ then
16:         $o_i \leftarrow q_i + in$.readInt(4)
17:         $\alpha_i \leftarrow$ trunc($v'_{i-1} \times 10^{-o_i}$) $\times 10^{o_i}$
18:     else
19:         $o_i \leftarrow o_{i-1}$
20:         $\alpha_i \leftarrow \alpha_{i-1}$        ▷ Acceleration of reconstruction
21:     return $\alpha_i, q_i, o_i$
22: function DeXOR_Decompressor($in, \alpha_i, o_i, q_i$)
23:     if $\alpha_i = 0$ then
24:         $sign \leftarrow in.readBoolean()? - 1 : 1$
25:     else
26:         $sign \leftarrow \alpha_i < 0? - 1 : 1$
27:     $\bar{\ell}_i \leftarrow \lceil \delta \times \log_2(10) \rceil$        ▷ $\delta = k_i - q_i$
28:     $\beta_i \leftarrow sign \times in.readLong(\bar{\ell}_i)$
29:     $v'_i \leftarrow \alpha_i + \beta_i \times 10^{q_i}$
30:     return $v'_i$
```

The efficient metadata storage we mentioned can not only save storage space during compression but can also be utilized to accelerate the computational process of decompression (Section 4.4, Algorithm 4). For example, the computation of reconstruction of the prefix α_i (line 20 of Algorithm 4) can be omitted in the Case $\boxed{1}\boxed{0}$. This is because when this case occurs ($q_i = q_{i-1} \land o_i = o_{i-1}$), the following conclusion can be drawn:

$$\alpha_{i-1} = \alpha_i = \text{trunc}(v_{i-1}\times 10^{-o_{i-1}})\times 10^{o_{i-1}} = \text{trunc}(v_i \times 10^{-o_i})\times 10^{o_i}. \tag{8}$$

In the event of an exception, the storage of metadata (the tail coordinate q_i and the LCP coordinate o_i) is skipped, and the process directly proceeds to the Exception Handler (see Section 5). The complete Exception handler module and its decompressor are presented in Algorithm 5 and Algorithm 6.

Algorithm 5 Exception_Handler (out, ω_i, exp_{i-1} in buffer, EL and $step$ are global variables)

```
1: $exp_i \leftarrow (\omega_i >> 52) \& 0x7FF$
2: $ES_i \leftarrow exp_i - exp_{i-1}$        ▷ Exponential Subtraction
3: $bias \leftarrow 2^{EL-1} - 1$
4: if $ES_i \in [-bias, bias]$ then
5:     $out$.write($ES_i + bias$, EL)
6:     $out$.write($\omega_i < 0$)        ▷ 1-bit sign
7:     $out$.write($\omega_i$, 52)        ▷ 52-bit fraction
8:     if $ES_i \in [-\frac{bias-1}{2}, \frac{bias-1}{2}]$ then
9:         $step$ += 1
10:    else
11:        $step \leftarrow 0$
12:    if EL $> 1 \land step \geq \theta$ then        ▷ Contraction
13:        EL −= 1
14:        $step \leftarrow 0$
15: else
16:     $out$.write($bias$, EL)
17:     $out$.write($\omega_i$, 64)
18:     $step \leftarrow 0$
19:     if EL $< 10$ then        ▷ Expansion
20:         EL += 1
21: return $ES_i$
```

Algorithm 6 Exception_Decompressor (binary input stream in and the previous exceptional exponent exp_{i-1} in buffer, EL and $step$ are global variables)

```
1: $ES_i \leftarrow in.read$(EL)
2: $bias \leftarrow 2^{EL-1} - 1$
3: if $ES_i \in [-bias, bias]$ then
4:     $exp_i \leftarrow exp_{i-1} + ES_i$
5:     $sign \leftarrow in$.readInt(1)
6:     $\omega_i \leftarrow sign||exp_i||in.readLong(52)$
7:     $contract(ES_i$, EL, $step)$
8: else
9:     $\omega_i \leftarrow in.readDouble(64)$
10:    $exp_i \leftarrow get\_exp(\omega_i)$
11:    $step = 0$
12:    $expand(ES_i$, EL$)$
13: return $\omega_i$
```

A.4 Algorithm of Extension to Higher Buffer Size

Another attempt involves an extended buffer size N strategy analogous to Chimp$_{128}$ but tailored specifically for DeXOR. What we need to do is to update the original results of LCP coordinates in Algorithm 1 and the reverter during decompression (Algorithm 4) with the method presented in Algorithm 7.

Algorithm 7 `Extension_N` (Buffer size N, `Buffer`:$v'_{i-N}, \cdots, v'_{i-1}$, current value v'_i, tail coordinate q_i)

```
1:  ···
2:  LCP coordinate $o_i \leftarrow$ get_LCP($v'_i, q_i$)            ▷ line 3 of Algorithm 1
3:  $o_i \leftarrow$ update_LCP($v'_i, o_i, N$)
4:  ···
5:  $out.$write($2$ or $1$ or $0, 2$)                             ▷ line 6/8/11 of Algorithm 2
6:  $out.$write($id_i - 1, \lceil log_2 N \rceil$)
7:  ···
8:  function update_LCP($v'_i, o_i, N$)                          ▷ Continue get_LCP
9:      $id_i \leftarrow 1$
10:     for $j = 2$ to $N$ do
11:         while trunc($v'_i \times 10^{-(o_i-1)}$) $\times 10^{(o_i-1)}$ = trunc($v'_{i-j} \times 10^{-(o_i-1)}$) $\times 10^{(o_i-1)}$ do
12:             $o_i \leftarrow o_i - 1$
13:             $id_i \leftarrow j$
14:     return $o_i, id_i$
15: function DeXOR_Reverter_N($in, q_i, code, N$)               ▷ Replace DeXOR_Reverter (line 10 of Algorithm 4)
16:     $id_i \leftarrow in.$readInt($\lceil log_2 N \rceil$) + 1
17:     if $code = 0$ then
18:         $q_i \leftarrow in.$readInt($5$) − 20
19:     else
20:         $q_i \leftarrow q_{i-1}$
21:     if $code < 2$ then
22:         $o_i \leftarrow q_i + in.$readInt($4$)
23:     else
24:         $o_i \leftarrow o_{i-1}$
25:     $\alpha_i \leftarrow [v'_{i-id_i} \times 10^{-o_i}] \times 10^{o_i}$        ▷ Acceleration fails
26:     return $\alpha_i, q_i, o_i$
```

The primary distinction from the original scheme lies in decompression. Under this extension, Equation 8 fails to hold, thereby precluding the acceleration of the reconstruction of the prefix α_i (line 25 of Algorithm 7 and line 20 of Algorithm 4).

In fact, we have supplemented our work with experiments on N across different sizes and demonstrated that our algorithm achieves the most optimal compression when it is not extended ($N = 1$). In the vast majority of cases, the extension does not yield better results than the original DeXOR due to the overhead associated with recording the index of the buffer.

A.5 Algorithm of Metadata Calculation

Algorithm 8 `get_tail` (current value v'_i, previous tail coordinate q_{i-1})

```
1:  if $v'_i = 0$ then
2:      return 0
3:  $q_i \leftarrow q_{i-1}$
4:  if $|v'_i \times 10^{-q_i}$ − trunc($v'_i \times 10^{-q_i}$)$| < \Delta$ then          ▷ Forward search
5:      while |trunc($v'_i \times 10^{-(q_i+1)}$) − $v'_i \times 10^{-(q_i+1)}| < \Delta$ do   ▷ Using Lemma 2
6:          $q_i \leftarrow q_i + 1$
7:  else                                                            ▷ Backward search
8:      $q_i \leftarrow q_i - 1$
9:      while |trunc($v'_i \times 10^{-q_i}$) − $v'_i \times 10^{-q_i}| \geq \Delta$ do
10:         $q_i \leftarrow q_i - 1$
11: return $q_i$
```

Algorithm 9 `get_LCP` (current value v'_i, current tail coordinate q_i)

```
1:  $o_i \leftarrow q_i$
2:  while $[v'_i \times 10^{-o_i}] \times 10^{o_i} \neq$ trunc($v'_{i-1} \times 10^{-o_i}$) $\times 10^{o_i}$ do ▷ Using Lemma 1
3:      $o_i \leftarrow o_i + 1$
4:  return $o_i$
```

Following Algorithm 1 in Section 4.2, Algorithms 8 and 9 provide full pseudo-code implementations of finding the `tail` and `LCP` procedures.

R1.D6 (3)

In Algorithm 8, the tail coordinate q_i is certified using Lemma 2. To accelerate the search, we exploit the *regularity* property (Appendix C.2), whereby successive q_i values exhibit strong locality. Initialization from q_{i-1} (line 3) typically resolves q_i within two iterations. To prevent over-estimation from rounding, we consider $\mathrm{trunc}(v'_i \times 10^{-(q_i+1)})$ and $v'_i \times 10^{-q_i}$ as equivalent whenever their difference is below the tolerance Δ (line 5, Section 4.2.1); occasional deviations are corrected by the Exception Handler. By contrast, Algorithm 9 enforces exact equality (line 2) to guarantee the correctness of the longest common prefix.

B SUPPORTING THEORY AND PROOFS

B.1 Loss of Smoothness in Elf

We provide a formula to quantify the **smoothness loss** after conversion:

$$\hat{q} = \min(v_X.q, v_Y.q), \tag{9}$$

$$S(v_X, v_Y) = \mathrm{CBL}\left((v_X \times 10^{-\hat{q}}) \oplus (v_Y \times 10^{-\hat{q}})\right), \tag{10}$$

The smoothness loss is then defined as:

$$\mathrm{Loss} = S(\hat{v}_X, \hat{v}_Y) - S(v_X, v_Y). \tag{11}$$

Certain algorithms, such as Elf, introduce precision-related errors that disrupt smoothness. For Elf, the loss can be expressed as:

$$\mathrm{Loss(Elf)} = S(v_X - \delta_{v_X}, v_Y - \delta_{v_Y}) - S(v_X, v_Y). \tag{12}$$

δ_{v_X} denotes the discrepancy between the value after erasure and the original value, which may be represented as: $\delta_{v_X} = sign \times 2^{(exp-bias)} \times fraction_{(\text{lower } g(v_X) \text{ bits})}$.

Lower $g(v_X)$ bits are erased. A concept defined by the original author, Elf, but can be described in the language of this paper as: $g(v_X) = 52 - (\lceil (-q) log_2(10) \rceil + exp - bias)$.

When XOR operations are applied, precision mismatches occur. Specifically, when $g(v_X) \neq g(v_Y)$, it follows that $\delta_{v_X} \neq \delta_{v_Y}$. As a result, the Elf algorithm can disrupt up to: abs $\left(g(v_X) - g(v_Y)\right)$ bits of already-erased CBL.

Although Elf attempts to combine redundancy elimination and smoothness exploitation, examples illustrate that performing XOR after erasure leads to poor results. Reversing the order — erasure after XOR — is equally problematic. Moreover, precision becomes uncontrollable after XOR operations, making such optimizations challenging to implement effectively.

B.2 Zero Loss of DECIMAL XOR Converter

We formally prove that the DECIMAL XOR converter incurs no loss of smoothness.

Lemma 5 (Zero Loss of DECIMAL XOR). *The smoothness loss of DECIMAL XOR satisfies:* $\forall \underline{v_X}, v_Y \in \mathbb{R}, Loss(\underline{v_X} \diamond v_Y) = 0$.

PROOF. According to the preconditions defined in Section 4.2.1, the DECIMAL XOR operation produces:

$$\hat{v}_X = \underline{v_X} \diamond v_Y = v_X - \alpha, \quad \hat{v}_Y = v_Y - \alpha, \tag{13}$$

where α denotes the shared prefix between v_X and v_Y.

Let $\hat{q} = \min(v_X.q, v_Y.q)$, we now compute the smoothness metric $S(\hat{v}_X, \hat{v}_Y)$:

$$\begin{aligned} S(\hat{v}_X, \hat{v}_Y) &= (\hat{v}_X \times 10^{-\hat{q}}) \oplus (\hat{v}_Y \times 10^{-\hat{q}}) \\ &= (v_X \times 10^{-\hat{q}} - \alpha) \oplus (v_Y \times 10^{-\hat{q}} - \alpha) \\ &= \Big((v_X \times 10^{-\hat{q}}) \oplus \gamma\Big) \oplus \Big((v_Y \times 10^{-\hat{q}}) \oplus \gamma\Big), \end{aligned}$$

where $\gamma = -\alpha \times 10^{-\hat{q}}$.

Since the $\oplus$ operation is associative and γ cancels out due to shared prefixes, we have:

$$S(\hat{v}_X, \hat{v}_Y) = (v_X \times 10^{-\hat{q}}) \oplus (v_Y \times 10^{-\hat{q}}) = S(v_X, v_Y). \tag{14}$$

Thus, the smoothness loss is:

$$\text{Loss}(v_X \diamond v_Y) = S(\hat{v}_X, \hat{v}_Y) - S(v_X, v_Y) = 0. \tag{15}$$

This completes the proof. □

B.3 Proof of Fixed Bit Allocation for Unsigned Binary Suffix

In Section 4.3, we introduce Lemma 4: 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., $\mathbb{E}[(4 + \bar{\ell}_i)] < \mathbb{E}[(6 + \ell_i)]$. Its detailed proof is provided as follows:

PROOF. Given: $\delta = o_i - q_i \in \mathbb{N}$, $\text{abs}(\beta_i) \in [10^{\delta-1}, 10^{\delta})$. $\ell_i = \lceil \log_2(\text{abs}(\beta_i) + 1) \rceil$, $\bar{\ell}_i = \lceil \log_2(10^{\delta}) \rceil$. We aim to prove: $\mathbb{E}[(\bar{\ell}_i + 4)] < \mathbb{E}[(6 + \ell_i)] \iff \mathbb{E}[(\bar{\ell}_i - \ell_i - 2)] < 0$.

We divide the range of $\text{abs}(\beta_i)$ by powers of 2. Let $j \in \mathbb{N}$ be the smallest integer such that $2^j > 10^{\delta-1}$, thus: $2^{j-1} \le \mathbf{10^{\delta-1}} < 2^j < 2^{j+1} < 2^{j+2} < \mathbf{10^{\delta}} < 2^{j+4}$. The relationship between 2^{j+3} and 10^{δ} gives two complementary cases:

Case (1) $2^{j+3} > 10^{\delta}$ with possibility $\mathbb{P}_1$; fixed allocation $\bar{\ell}_i = \lceil \log_2(10^{\delta}) \rceil = j + 3$ whereas

$$\ell_i = \begin{cases} j & \text{if } \text{abs}(\beta_i) \in [10^{\delta-1}, 2^j), \\ j+1 & \text{if } \text{abs}(\beta_i) \in [2^j, 2^{j+1}), \\ j+2 & \text{if } \text{abs}(\beta_i) \in [2^{j+1}, 2^{j+2}), \\ j+3 & \text{if } \text{abs}(\beta_i) \in [2^{j+2}, 10^{\delta}). \end{cases}$$

Calculating the expectation of case (1): $\mathbb{E}_1 = \mathbb{E}(\bar{\ell}_i + 4 \mid 2^{j+3} > 10^{\delta}) - \mathbb{E}(\ell_i + 6 \mid 2^{j+3} > 10^{\delta}) = j + 1 - \mathbb{E}(\ell_i \mid 2^{j+3} > 10^{\delta})$.

$$\begin{aligned} \mathbb{E}(\ell_i \mid 2^{j+3} > 10^{\delta}) &= \frac{(j+3)(10^{\delta} - 2^{j+2})}{10^{\delta} - 10^{\delta-1}} + \frac{(j+2)(2^{j+2} - 2^{j+1})}{10^{\delta} - 10^{\delta-1}} \\ &\quad + \frac{(j+1)(2^{j+1} - 2^j)}{10^{\delta} - 10^{\delta-1}} + \frac{j(2^j - 10^{\delta-1})}{10^{\delta} - 10^{\delta-1}} \\ &= j + \frac{3(10^{\delta} - 2^{j+2}) + 2(2^{j+2} - 2^{j+1}) + (2^{j+1} - 2^j)}{10^{\delta} - 10^{\delta-1}} \\ &= j + \frac{3 \times 10^{\delta} - 2^{j+2} - 2^{j+1} - 2^j}{10^{\delta} - 10^{\delta-1}} \\ &= j + 3 - \frac{2^{j+2} + 2^{j+1} + 2^j - 3 \times 10^{\delta-1}}{10^{\delta} - 10^{\delta-1}} \\ &> j + 3 - \frac{2^{j+2} + 2^{j+1} + 2^j - 3 \times 2^{j-1}}{2^{j+2} - 2^j} \\ &= j + 3 - \frac{(8+4+2-3) \times 2^{j-1}}{(8-2) \times 2^{j-1}} \\ &= j + 3 - \frac{11}{6} = j + \frac{7}{6} \end{aligned}$$

$\implies \mathbb{E}_1 = j + 1 - \mathbb{E}(\ell_i | 2^{j+3} > 10^{\delta}) < -\frac{1}{6}$.

Calculating possibility $\mathbb{P}_1$ of case (1): $\mathbb{P}_1\{2^{j+3} > 10^{\delta}\} = \mathbb{P}\{(j+3)\log_{10} 2 > \delta\}$,

we know the condition $2^j \ge 10^{\delta-1} \implies j \log_{10} 2 + 1 \ge \delta$,

and we know $2^{j-1} \le 10^{\delta-1} \implies (j-1)\log_{10} 2 + 1 \le \delta$,

so $\delta \in [(j-1)\log_{10} 2 + 1, j \log_{10} 2 + 1]$

$$\mathbb{P}\{(j+3)\log_{10} 2 > \delta\} = \frac{(j+3)\log_{10} 2 - (j-1)\log_{10} 2 - 1}{\log_{10} 2}$$

$$= \frac{4\log_{10} 2 - 1}{\log_{10} 2} \implies \mathbb{P}_1\{2^{j+3} > 10^{\delta}\} = 4 - \log_2 10 \approx 0.6781.$$

Case (2) $2^{j+3} \le 10^{\delta}$ with possibility $\mathbb{P}_2\{2^{j+3} \le 10^{\delta}\} = 1 - \mathbb{P}_1 \approx 0.3219$; fixed allocation $\bar{\ell}_i = \lceil m \log_2 10 \rceil = \lceil \log_2(2^{j+4}) \rceil = j + 4$ whereas

$$\ell_i = \begin{cases} j & \text{if } \text{abs}(\beta_i) \in [10^{\delta-1}, 2^j), \\ j+1 & \text{if } \text{abs}(\beta_i) \in [2^j, 2^{j+1}), \\ j+2 & \text{if } \text{abs}(\beta_i) \in [2^{j+1}, 2^{j+2}), \\ j+3 & \text{if } \text{abs}(\beta_i) \in [2^{j+2}, 2^{j+3}), \\ j+4 & \text{if } \text{abs}(\beta_i) \in [2^{j+3}, 10^{\delta}). \end{cases}$$

The expectation of this case: $\mathbb{E}_2 = \mathbb{E}(\bar{\ell}_i + 4 \mid 2^{j+3} < 10^{\delta}) - \mathbb{E}(\ell_i + 6 \mid 2^{j+3} < 10^{\delta}) = j + 2 - \mathbb{E}(\ell_i \mid 2^{j+3} < 10^{\delta})$.

$$\begin{aligned} \mathbb{E}(\ell_i \mid 2^{j+3} < 10^{\delta}) &= j + \frac{4 \times 10^{\delta} - 2^{j+3} - 2^{j+2} - 2^{j+1} - 2^j}{10^{\delta} - 10^{\delta-1}} \\ &= j + 4 - \frac{2^{j+3} + 2^{j+2} + 2^{j+1} + 2^j - 4 \times 10^{\delta-1}}{10^{\delta} - 10^{\delta-1}} \\ &> j + 4 - \frac{2^{j+3} + 2^{j+2} + 2^{j+1} + 2^j - 4 \times 2^{j-1}}{2^{j+3} - 2^j} \\ &= j + 4 - \frac{(16+8+4+2-4) \times 2^{j-1}}{(16-2) \times 2^{j-1}} \\ &= j + \frac{13}{7} \end{aligned}$$

$\implies \mathbb{E}_2 = j + 2 - \mathbb{E}(\ell_i | 2^{j+3} < 10^{\delta}) < \frac{1}{7}$.

Finally, we have the overall expectation $\mathbb{E}[(\bar{\ell}_i - \ell_i - 2)] = \mathbb{E}_1 \times \mathbb{P}_1 + \mathbb{E}_2 \times \mathbb{P}_2 < -\frac{1}{6} \times \mathbb{P}_1 + \frac{1}{7} \times \mathbb{P}_2 \approx -0.067 < 0$. □

Table 7: Dataset characteristics, with numerical values in 'value range' rounded to two decimal places.

	Dataset	Sequence Length	*dp*	Value Range	Source Domain
Time-Series	WS (Wind speed)	199,570,396	3	[0.01, 1.45]	Meteorology
	PM (PM10-dust)	222,911	3	[0.001, 150]	Air Quality
	CT (City-temp)	2,905,887	3	[-99, 100.2]	Urban Climate
	IR (IR-bio-temp)	380,817,839	3	[-1.69, 2.71]	Biomedical IR Sensing
	DPT (Dewpoint-temp)	5,413,914	4	[34.38, 98.88]	Meteorology
	SUSA (Stocks-USA)	374,428,996	5	[13.98, 250.64]	Finance
	SUK (Stocks-UK)	115,146,731	5	[145.51, 6087.5]	Finance
	SDE (Stocks-DE)	45,403,710	6	[3.85, 147.63]	Finance
	AP (Air-pressure)	137,721,453	7	[86.99, 89.26]	Meteorology
	BM (Bird-migration)	17,964	7	[4.03, 58.52]	Ecology
	BW (Basel-wind)	124,079	8	[0, 79.99]	Meteorology
	BT (Basel-temp)	124,079	9	[-12.52, 36.7]	Meteorology
	BP (Bitcoin-price)	7,116	9	[13068.28, 532323.17]	Cryptocurrency
	AS (Air-sensor)	8,664	17	[0.18, 75.8]	IoT / Air Quality
Non-Time-Series	FP (Food price)	2,050,638	3	[0.17, 5833333]	Economics
	EVC (Vehicle-charge)	3,395	3	[0, 23.68]	E-mobility
	SSD (SSD-bench)	8,927	4	[0.57, 14600]	Storage Benchmark
	BL (Blockchain-tr)	231,031	5	[0, 273.54]	Blockchain
	CA (City-lat)	41,001	6	[-54.93, 81.72]	Geospatial / GIS
	CO (City-lon)	41,001	7	[-179.59, 179.37]	Geospatial / GIS
	PA (POI-lat)	424,205	16	[-1.56, 1.45]	Geospatial / POI
	PO (POI-lon)	124,205	16	[3.14, 3.14]	Geospatial / POI

C MORE EXPERIMENTAL DETAILS

C.1 Dataset Description

Our study encompasses 22 cross-domain datasets, following the benchmark design adopted in multiple prior studies [9, 33, 46]. The collection includes 14 time-series and 8 non-time-series datasets. As summarized in Table 7, these datasets exhibit diverse characteristics: sequence lengths vary from only a few thousand to hundreds of millions of points; numerical precision (*dp*) ranges from 3 to 17; value ranges span from narrow intervals (e.g., $[0.01, 1.45]$) to extremely wide domains (e.g., $[13{,}068, 532{,}323]$); and the sources cover domains such as meteorology, finance, ecology, mobility, blockchain, and geospatial analysis. Together, they provide a representative and challenging evaluation workload.

R1.W5 &D5

Most of the benchmark datasets are collected from real-world workloads, broadly reflecting the data streams encountered in streaming lossless compression scenarios. For completeness, we briefly introduce their application contexts and provide download sources as follows.

R2.W1 (1) W1

Time-Series Datasets:

- City-temp (CT)[7]: Temperature records from major cities worldwide, collected by the University of Dayton.
- NEON Datasets: A collection of five datasets from various sensors, published by the National Ecological Observatory Network (NEON):
  - Air-pressure (AP)[8],
  - Dewpoint-temperature (DPT)[9],
  - IR-bio-temperature (IR)[10],
  - PM10-dust (PM)[11],
  - Wind-speed (WS)[12].
- Stock Exchange Datasets[13]: Exchange price data from three countries:
  - UK (Stocks-UK, SUK),
  - USA (Stocks-USA, SUSA),
  - Germany (Stocks-DE, SDE).
- Meteoblue Datasets[14]: Historical weather data for Basel, Switzerland, including:
  - Wind speed (Basel-wind, BW),
  - Temperature (Basel-temp, BT).
- InfluxDB Datasets[15]: A set of datasets from various domains, including:
  - Air-sensor (AS),
  - Bird-migration tracking (BM),
  - Bitcoin-price (BP).

Non-Time-Series Datasets:

- Electric vehicle charging sessions (EVC)[16].
- Global food prices (FP) for December 2020[17].
- Bitcoin transaction values (Blockchain transactions, BL)[18] for a specific day.
- Storage disk benchmarking scores (SSD)[19].
- Geographic city coordinates[20], which include:
  - Cartesian coordinates (CA),
  - Longitude and latitude coordinates (CO).
- Position of Interest (POI)[21] radian coordinates extracted from Wikipedia parsing, including:
  - Angular coordinates (PA),
  - Polar coordinates (PO).

C.2 Regularity Analysis: Case Proportions and Efficiency

Based on the reuse relationships between the tail coordinates q_i, the LCP coordinates k_i, and their historical values q_{i-1} and k_{i-1}, we categorize the scenarios into four distinct cases. We conduct experiments across all existing datasets and calculate the average proportion of each case. The results are shown in Figure 11, which reports the average proportions of these cases across all datasets.

Referring to Figure 11, it is evident that **Case** $\boxed{1}\boxed{0}$ (see Section 4.2.2), which achieves the maximum efficiency by occupying only 2 bits, accounts for the largest proportion, at 38.55%. The second most significant case, **Case** $\boxed{0}\boxed{1}$ (occupying 6 bits), constitutes 33.41%. Together, these two dominant cases cover the majority of the scenarios.

The remaining two cases contribute a combined proportion of only 28.04%. Notably, the scenario where $q_i \neq q_{i-1} \wedge k_i = k_{i-1}$,

[7] 2023. Daily Temperature of Major Cities. https://www.kaggle.com/sudalairajkumar/daily-temperature-of-major-cities

[8] 2022.Barometric pressure. https://data.neonscience.org/data-products/DP1.00004.001/RELEASE-2022

[9] 2022. Relative humidity above water on the buoy. https://data.neonscience.org/data-products/DP1.20271.001/RELEASE-2022

[10] 2022. IR biological temperature. https://data.neonscience.org/data-products/DP1.00005.001/RELEASE-2022

[11] 2022. Dust and particulate size distribution. https://data.neonscience.org/data-products/DP1.00017.001/RELEASE-2022

[12] 2022. 2D wind speed and direction. https://data.neonscience.org/data-products/DP1.00001.001/RELEASE-2022

[13] 2020. Financial data set used in the INFORE project. https://zenodo.org/records/3886895#.Y4DdzHZByM_

[14] 2023. Basel Historical Weather Data. https://www.meteoblue.com/en/weather/archive/export/basel_switzerland

[15] 2013. Scalable datastore for metrics, events, and real-time analytics. https://github.com/influxdata/influxdb2-sample-data

[16] 2023. Electric Vehicle Charging Dataset. https://www.kaggle.com/datasets/michaelbryantds/electric-vehicle-charging-dataset

[17] 2021. Global Food Prices Database (WFP). https://data.humdata.org/dataset/wfp-food-prices

[18] 2023. Bitcoin Transactions. https://gz.blockchair.com/bitcoin/transactions/

[19] 2022. SSD and HDD Benchmarks. https://www.kaggle.com/datasets/alanjo/ssd-and-hdd-benchmarks

[20] 2023. World City. https://www.kaggle.com/datasets/kuntalmaity/world-city

[21] 2023. Points of Interest POI Database. https://www.kaggle.com/datasets/ehallmar/points-of-interest-poi-database

Table 9: Comparison with schemes with prior precision judgment (Exception Handler only).

Dataset	AS		PA		PO	
Algorithm	DeXOR	Exception	DeXOR	Exception	DeXOR	Exception
ACB	**52.28**	54.12	57.86	**56.10**	58.70	**56.86**
Comp. Speed	**7.45**	4.83	2.14	**8.16**	1.86	**10.20**
Decomp. Speed	**62.16**	22.63	**59.51**	48.61	24.59	**53.81**

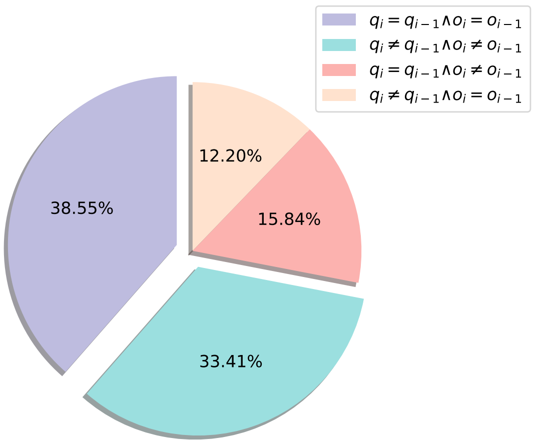

Figure 11: Proportion of reuse cases described in Section 4.2.2.

despite its multiplexing effect (occupying 7 bits), represents the smallest proportion, at merely 12.20%. To optimize efficiency, we merge these cases into **Case** $\boxed{0}\boxed{0}$, where $q_i \neq q_{i-1}$. This merging allows us to implement an optimal allocation scheme while reserving **Case** $\boxed{1}\boxed{1}$ for the Exception Handler module (see Section 5).

C.3 Comparisons of Exponential Subtraction and Exponential XOR

To validate the effectiveness of XOR and subtraction operations in managing high-precision datasets, we conducted experiments on three datasets: AS, PA, and PO. The results of these experiments are presented in Table 8.

As shown in Table 8, the overall efficacy of the two approaches is comparable. However, the subtraction method tends to yield lower CBLs in certain cases, particularly for the PO dataset. Specifically, while XOR performs better on PA, subtraction excels on PO, with both methods performing equally well on AS.

Table 8: CBL generated by different operations.

Operation	AS	PA	PO
$exp_i \oplus exp_{i-1}$	**0.02**	**0.51**	2.23
$exp_i - exp_{i-1}$	**0.02**	0.65	**1.14**

C.4 Compression with Prior Precision Judgment

As discussed in Section 5.3, we explore additional applications of the Exception Handler module. To address irregular or non-smooth data, we introduce a *mode-switching* mechanism: when anomalies exceed a threshold, the core DeXOR pipeline is bypassed, and both `LCP` and `tail` computations are skipped in favor of direct delegation to the Exception Handler.

R2.W2 (3) 2b

R2.W2 (4) 2c

To validate the effectiveness of this mechanism, we conduct supplementary experiments on three high-precision datasets: AS, PA, and PO. The results, presented in Table 9, compare the performance of DeXOR with the Exception Handler used independently.

The results reveal several important insights:

- **Compression Ratio**: The Exception Handler alone achieves marginal gains in compression ratio for certain datasets (e.g., PA and PO), primarily due to the omission of the 2-bit case code. However, when excluding this gain, the compression ratio of the Exception Handler is consistently inferior to that of the complete DeXOR. This gap is particularly pronounced in the time-series dataset AS, where the main process of DeXOR demonstrates significantly better performance by effectively reducing precision.
- **Compression and Decompression Speed**: The Exception Handler exhibits considerable improvements in compression speed and modest gains in decompression speed. For non-time-series datasets, such as PA and PO, the Exception Handler is notably faster. However, it struggles with time-series data, such as AS, where the main process retains its advantage.

For typical workloads such as AS, the existing heuristics for `LCP` and `tail` already exploit the strong regularity of contiguous data, yielding significant speed-ups. Further refinements would bring only minor gains with limited practical impact.

R2.W2 (4) 2c

These results suggest that the Exception Handler, when used independently, is best suited for high-precision, non-time-series datasets where precision reduction in the main pipeline is rarely needed. Although the Exception Handler can offer speed advantages, the main DeXOR process consistently delivers superior compression effectiveness on time-series data.

R2.W1 (3) 1b

C.5 Discussions on Inclusion or Exclusion of the Exception Handler in Extreme Cases

This section further validates the effectiveness of the Exception Handler on high-precision datasets: AS, PA, and PO. As illustrated in Figure 12, we compare DeXOR against an ablated version without the Exception Handler. These datasets are characterized by virtually no representational redundancy, making them challenging for compression algorithms.

R2.W1 &1a

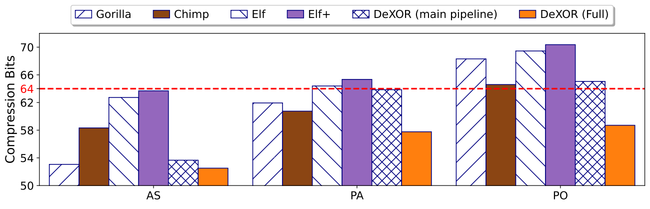

Figure 12: Comparison of DeXOR with and without the Exception Handler module. The red line ($y = 64$) indicates the average bit occupancy of `double` precision data without compression.

For such datasets, only algorithms that fully exploit temporal smoothness, such as XOR-based compressors like Gorilla and Chimp/Chimp$_{128}$, achieve any level of compression. In contrast,

Table 10: A detailed comparison of the DeXOR with other SLC schemes that have higher buffer sizes. Unlike DeXOR, which references only a single previous value in the stream, Chimp$_{128}$ and SElf∗ utilize sliding windows, while ALP and Elf∗ rely on truncated windows (mini-batches), referencing significantly more previous values.

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	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
	Chimp$_{128}$	13.96	13.21	25.28	16.39	27.70	17.56	26.97	21.37	47.59	37.27	48.07	33.83	55.61	58.35	24.41	24.86	17.13	36.31	51.77	55.02	60.53	64.42	31.37	24.14
	SElf∗	11.61	8.40	12.12	8.82	15.20	10.51	11.08	11.94	16.30	21.80	32.90	30.82	29.56	50.57	13.25	17.13	13.56	17.77	29.51	34.40	59.36	65.23	19.55	14.16
	ALP	25.87	62.06	50.48	42.95	51.50	41.47	41.97	48.21	48.30	53.01	59.54	61.97	58.71	71.32	49.77	61.43	47.79	49.73	56.04	53.16	80.57	81.72	52.97	47.66
	Elf∗	**8.80**	**6.35**	**10.89**	**7.51**	**12.83**	**8.71**	**9.21**	**10.97**	**14.45**	19.88	**30.55**	**28.88**	**27.07**	**49.22**	12.41	15.09	13.32	16.45	27.57	32.86	**57.62**	63.15	**17.55**	**12.34**
Comp. Speed	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
	Chimp$_{128}$	18.86	26.44	9.57	23.60	4.68	42.96	32.22	**30.16**	31.88	3.58	**14.26**	**24.91**	13.77	1.73	5.42	10.36	1.05	**34.46**	**30.41**	**29.54**	**9.41**	**4.85**	12.65	16.62
	SElf∗	15.81	28.63	26.85	29.26	16.48	**60.10**	**39.06**	17.54	30.53	**3.78**	3.77	20.33	**47.89**	3.79	**20.13**	7.26	**24.73**	20.33	22.89	19.12	1.57	1.94	14.54	22.53
	ALP	0.82	0.90	2.48	1.96	2.11	2.97	5.13	1.53	3.37	0.67	1.47	5.24	7.82	4.37	1.60	0.17	2.44	3.34	3.68	3.77	1.62	1.76	2.07	1.87
	Elf∗	19.04	30.42	**34.17**	51.20	**20.08**	53.67	36.06	16.74	32.74	3.51	4.88	17.97	40.40	**8.56**	16.42	**10.39**	17.82	23.41	18.75	19.67	2.18	2.54	**16.06**	**24.34**
Dec. Speed	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**
	Chimp$_{128}$	**75.87**	**79.22**	14.58	70.45	11.26	60.12	55.77	52.42	56.42	6.94	34.28	51.93	**52.02**	25.02	67.52	**43.03**	20.40	**64.15**	44.63	55.45	50.31	**38.86**	40.27	42.97
	SElf∗	32.56	33.19	55.52	81.44	53.35	70.14	69.83	84.81	106.25	8.09	**41.69**	**87.06**	33.93	61.26	**75.22**	29.35	62.06	54.13	77.76	42.97	56.28	21.26	49.69	54.30
	ALP	39.80	22.74	32.64	64.32	26.71	43.11	43.02	53.06	**152.04**	6.69	32.31	58.89	18.49	**67.95**	38.59	20.73	71.81	35.97	61.13	57.12	48.67	20.31	38.73	41.03
	Elf∗	40.67	31.17	**64.35**	5.24	**57.07**	76.88	65.65	60.82	130.65	6.60	33.19	74.21	2.93	5.31	73.74	1.23	6.15	46.54	**78.70**	2.62	51.41	19.63	23.41	23.14

algorithms like Camel, Elf/Elf+, and the ablated DeXOR incur overheads that outweigh their compression gains.

The extreme PO dataset, which lacks both precision redundancy and temporal smoothness, poses a significant challenge. In this scenario, all algorithms fail except DeXOR with its Exception Handler. The Exception Handler adapts by capturing the largest residual similarities at minimal cost. Even in the theoretical worst-case scenario, where consecutive values differ by more than 2^{1023}, the Exception Handler incurs at most a one-bit penalty.

Under realistic conditions, the Exception Handler consistently secures the best compression ratios for high-precision datasets. Its ability to adaptively handle extreme cases ensures that DeXOR remains effective, even on datasets with minimal redundancy. These results highlight the critical role of the Exception Handler in achieving superior performance on challenging high-precision data.

R2.W1 (2) 1a

Robustness of DeXOR Main Pipeline. According to the results of this ablation study, the DeXOR main pipeline shows no inherent weakness on those high-precision or non-smooth data. Before our work, no algorithm explicitly addressed these cases, and all state-of-the-art methods degraded notably. While all baselines deteriorate on datasets like AS, PA, and PO, the DeXOR main pipeline (without the Exception Handler) still delivers the most competitive compression ratios — achieving lower bit-budgets than Elf and Elf+, and outperforming Chimp on AS and Gorilla on PO. Although precision-agnostic schemes like Gorilla and Chimp may occasionally surpass DeXOR on isolated traces, DeXOR achieves a consistent 4×–5× advantage over them on low-precision datasets, which dominate real-world workloads.

Our design therefore provides a comprehensive solution: the DeXOR main pipeline already outperforms most baselines even without special handling, and for extreme cases we provide an Exception Handler plugin. This handler is not a separate scheme but a lightweight extension of the DeXOR main pipeline, ensuring that DeXOR remains a comprehensive and robust solution with stable compression ratios across diverse workloads.

C.6 More Details of Higher Buffer Size Schemes

In Table 10, we present a detailed comparison of DeXOR and those schemes augmented with a higher buffer size (each scheme uses its recommended buffer size reported in the original paper: $N = 128$ for Chimp$_{128}$, $N = 1024$ for ALP, and $N = 1000$ for SElf∗ and Elf∗).

In terms of the geometric mean measures, DeXOR demonstrates the fastest decompression speed while securing the second-highest compression ratio and speed, trailing only Elf∗. When compared with the streaming version of Elf∗, namely SElf∗, DeXOR outperforms it across all metrics.

Furthermore, while Elf∗ holds an advantage in compression ratio for time-series data, DeXOR consistently delivers a **superior compression ratio for non-time-series data**, showcasing its versatility and effectiveness.

C.7 More Details of Applying Pre-processing Methods

We begin by noting that zero-mean normalization and 0-1 standardization typically degrade lossless compression. By altering the original numeric distribution, they often destroy local smoothness and increase entropy, leading to worse compression ratios. Moreover, these steps introduce **irreversible information loss** and **additional runtime cost**. As a result, prior studies (e.g., Elf [33], ALP [9], Camel [46]) resort to compressing on raw input values. In this study, we opt to adopt the same setting to pursue state-of-the-art compression performance.

R1.W2 &D2

To provide a complete picture, we evaluated both pre-processing steps, namely zero-mean normalization (zero-mean) and standardization (Stdz.), on four representative datasets (CT, AS, FP, PA). As shown in Table 11, normalization and standardization basically degrade the compression ratios. In contrast, **DeXOR achieves the best results on the original inputs**, confirming that compression is most effective without pre-processing.

We exclude Camel from detailed comparisons because its lossless mode only supports values with up to four decimal digits. After pre-processing, all datasets exceed this limit, causing the algorithm to silently fall back to *lossy* compression. Additionally, its fixed bit-width for integer deltas cannot handle the large residuals in FP, leading to overflows. Although pre-processing narrows the residual range and avoids crashes, the results remain lossy. Note that we

Table 11: Average Compression Bits (ACB, lower is better) under different input pre-processing strategies. We compare raw inputs (origin) against commonly used pre-processing steps, namely zero-mean normalization (zero-mean) and standardization (Stdz.), across four representative datasets: CT (low-precision time-series), AS (high-precision time-series), FP (low-precision non-time-series), and PA (high-precision non-time-series).

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

Table 12: Compression and query performance comparison between Gorilla and DeXOR within IoTDB across three vector datasets (SIFT, WR, WW). Metrics include average compression bits (lower is better), compression throughput (higher is better), and query latency per 1,000 queries (lower is better).

Metric	Algorithm	Dataset		
		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**

have not yet mentioned the floating-point precision loss introduced by pre-processing itself, which would affect all lossless schemes we evaluated.

The degradation primarily results from **inflated precision**: for example, 50.0 in FP becomes −0.043129525552165315 after standardization, pushing most values into conservative encodings and collapsing the compression ratio relative to raw-data DeXOR. This penalty is most severe for methods exploiting precision redundancy, while smoothness-based encodings (e.g., Gorilla) are less affected — indeed, Gorilla slightly outperforms DeXOR on AS. Such cases, however, are rare, and extra pre-processing yields at most 1.28 bits of improvement. Thus, applying DeXOR directly to unprocessed data is both efficient and robust.

For strict streaming scenarios, applying normalization or standardization is often impractical, as these require prior knowledge of the mean or min–max values. Even if pre-processed input exists, the significant performance gap before and after pre-processing suggests that an inverse normalization or standardization step could still enable state-of-the-art methods, such as DeXOR, to achieve superior compression ratios.

C.8 Visualization of Overall Comparison Results

R1.W4 &D4 (2)

The quantitative findings of our main experiment are consolidated in Table 2, detailed in Section 6.2, and visualized as a box-plot in Figure 13. To ensure fairness, the evaluation datasets were stratified into two precision-based cohorts: low-*dp* and full-*dp*. This is necessary because Camel's lossless mode operates only on reduced-precision corpora; thus, Camel is not included in the full-*dp* comparison (second row of Figure 13).

As shown in Figure 13, DeXOR consistently achieves the best compression ratio with the smallest inter-quartile range, indicating stable performance across datasets. Moreover, it delivers the highest decompression throughput while maintaining compression speed that is highly competitive with other methods. Indeed, the highest compression speed among all tested cases is achieved by DeXOR.

C.9 Evaluations with Real-World Time Series Databases (TSDB)

We integrated DeXOR into TsFile, the columnar storage engine of the open-source TSDB — Apache IoTDB [40]. This database is purpose-built for edge-to-cloud scenarios, sustaining millions of samples per second with sub-second ingestion latency and offering high compression ratios through TsFile's hybrid page–chunk layout, adaptive encodings, and time-partitioned indices. Our DeXOR plugs into its adaptive encoding and inherits runtime codec selection without extra orchestration. All empirical results in single-node are documented in Table 13.

R2.W1 (1) W1 (3) 1c&1d

We evaluate DeXOR on three real-world datasets—CT, FP, and PA (all listed in Table 11 alongside the synthetic AS dataset). The three datasets are classic and heterogeneous: CT (low-precision, time-series), FP (low-precision, non-time-series), and PA (high-precision, non-time-series). Table 13 contrasts DeXOR with IoTDB-embedded Gorilla and Sprintz (the two dominant baselines), and further investigates the effect of cascading general-purpose compressors (e.g., Lz4 [3], Snappy [6]) on the DeXOR-compressed payload.

R4.W2 &D2

Empirical evidence reveals that DeXOR shrinks the data volume to merely 20%-50% of that achieved by Gorilla, while sustaining comparable compression throughput and delivering a markedly lower average query latency. This query latency is defined as the total milliseconds required to complete 1, 000 single-value queries. Owing to DeXOR's inherently elevated compression ratio, the incremental space reduction afforded by the secondary, general-purpose stage is limited. Conversely, the overhead introduced by this additional compaction phase is so negligible that it occasionally precipitates a modest acceleration in both ingestion throughput and query responsiveness.

Support for Vector Data Compression within IoTDB. Through deep integration with IoTDB, we observed that vector data exhibit strong dimension-wise separability. Accordingly, we implemented a dimension-oriented DeXOR compression strategy within IoTDB, with results summarized in Table 12. Since most baselines lack vector-specific designs, we compare against the widely adopted Gorilla. We evaluate on two benchmarks: SIFT [2] (10k vectors, 128

R1.W1 &D1 (2)

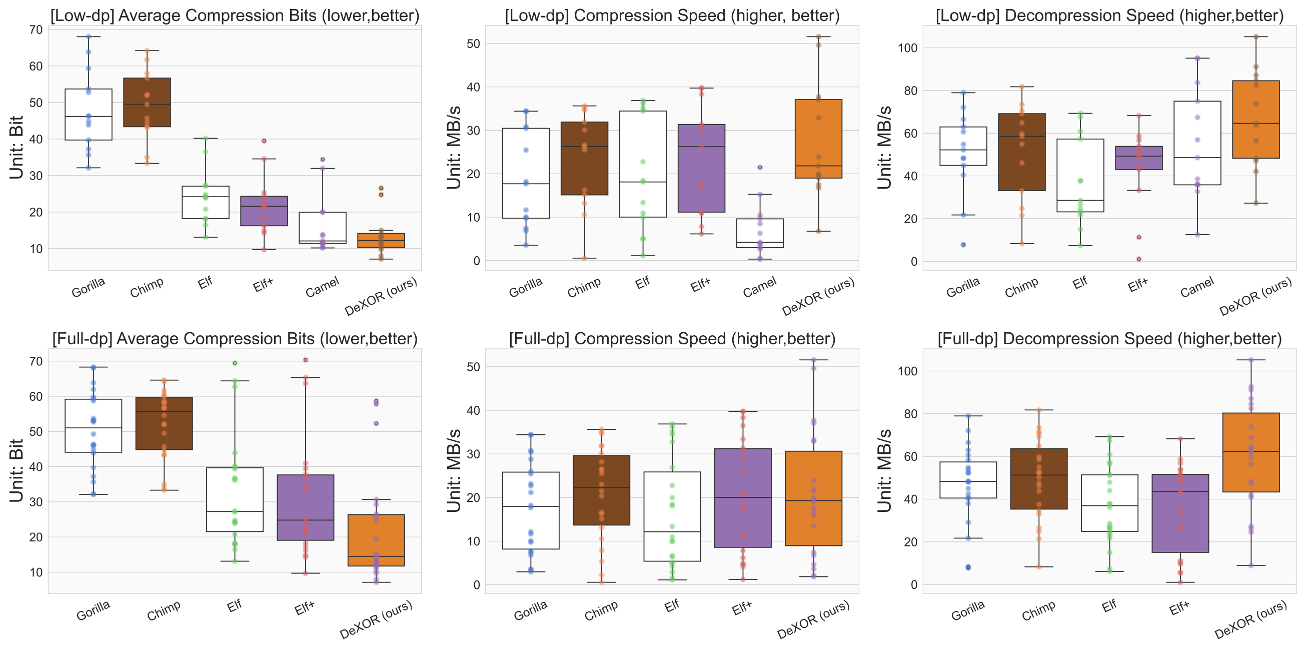

Figure 13: Box-plot visualization of average compression bits, compression speed, and decompression speed under two precision settings: low-*dp* (top row) and full-*dp* (bottom row). Each box shows the interquartile range with the median, whiskers denote 1.5×IQR, and points represent individual dataset outcomes. Camel appears only in the low-*dp* setting since its lossless mode is limited to reduced-precision corpora.

Table 13: Overall comparison of average compression bits (↓), compression throughput (MB/s, ↑) and query latency (ms/1k, ↓). All results are produced within Apache IoTDB Tsfile, evaluated on three real-world data categories: CT (time-series), FP (non-time-series), and PA (high-precision non-time-series).

Secondary Compression	Algorithm	Average Compression Bits				Compression Throughput				Query Latency			
		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**

dimensions) and Wine-Quality [16] (4,898 instances, 11 features for red/white wines — WR and WW). A query corresponds to retrieving a complete vector record, and latency is measured over 1,000 such queries.

As shown in Table 12, DeXOR consistently surpasses Gorilla: reducing average compression size by up to 34%, boosting throughput by up to 73%, and lowering query latency by over an order of magnitude on certain datasets. These results demonstrate that DeXOR generalizes effectively to vector data while maintaining strong performance guarantees.