2025 IEEE 41st International Conference on Data Engineering (ICDE)

BOS: Bit-packing with Outlier Separation

Jinzhao Xiao
Tsinghua University
xjc22@mails.tsinghua.edu.cn

Zihan Guo
Tsinghua University
gzh23@mails.tsinghua.edu.cn

Shaoxu Song*
Tsinghua University
sxsong@tsinghua.edu.cn

***Abstract*—Bit-packing serves as the fundamental operator in various data encoding and compression methods. The idea is to use a fixed bit-width to represent all the (processed) values in a sequence. Some extremely large values, known as outliers, obviously amplify the bit-width, and thus lead to wasted bits for most other small values. We notice that not only the large values (upper outliers) but also the small ones (lower outliers) could incur wasted bit-width. In this paper, we propose to store both the upper and lower outliers separately, namely Bit-packing with Outlier Separation (BOS). While the remaining center values have a narrow spread, i.e., condensed bit-width, the separated outliers need some extra cost to denote their positions. The problem is thus how to determine better thresholds for separating the upper and lower outliers, yielding smaller storage cost. Rather than enumerating all the possible values as upper and lower outlier separators, in $O(n^2)$ time, we consider bit-width as the separators, with $O(n \log n)$ search time. Theoretical analysis illustrates all the possible cases such that the bit-width separation still returns the optimal solution as the value separation, and further leads to an approximate separation strategy with both median and bit-width, in $O(n)$ time. Remarkably, our BOS is compatible to any existing compression methods using Bit-packing, and has replaced Bit-packing in Apache IoTDB and Apache TsFile. The extensive experiments on many real-world datasets demonstrate that by replacing Bit-packing with the proposed BOS in various compression methods, the compression ratio is significantly improved from about 2.75 to 3.25.**

***Index Terms*—series, outlier, compression**

I. Introduction

There are many algorithms proposed to compress series data [37], [11], [25], [29], [18], [2]. Among them, many algorithms [37], [11], [2] employ Bit-packing [19] to improve storage by using the same bit-width for storing values in a block and removing leading zeros. Take a series of values $X = (3, 2, 4, 5, 3, 2, 0, 8)$ as an example. Its maximum value is 8. The bit-width of 8 is 4 after removing leading zero. Thus, these 8 values can be stored with 4 bits respectively in bit-packing.

A. Motivation

Note that in the above example series, only the large value 8, an outlier, needs 4 bits to store, while a bit-width 3 is sufficient for all the remaining values. That is, the outlier 8 incurs all the other values wasting 1 bit in Bit-packing.

1) Outlier Separation Strategy: A natural idea is thus to store the outliers separately, so that the remaining values could use a smaller bit-width. PFOR [37] and its variations, NewPFOR [34], OptPFOR [34], FastPFOR [17] and SimplePFOR [17], propose to use b bits to store a part of the values, while separately storing others exceeding $2^b - 1$. For example, the aforesaid series can use 3 bits to store all the values except 8, which is processed separately.

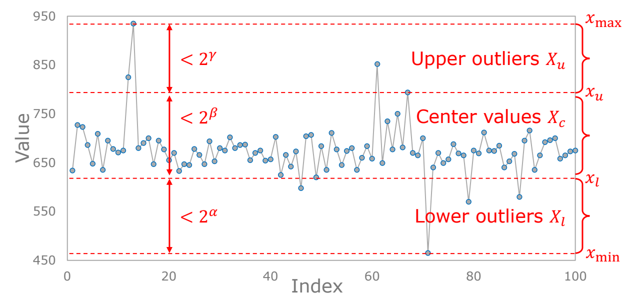


Fig. 1: A series X could be divided into 3 parts to compress separately the lower outliers X_l, center values X_c and upper outliers X_u, where α, β, γ are the bit-widths for storing the corresponding values.

We further notice that not only the large values, such as the above 8 known as upper outliers, but also the extremely small ones could amplify the bid width, e.g., value 0 in the series. By further separating the lower outlier 0, the remaining values $(3, 2, 4, 5, 3, 2)$ need only a bit-width 2 to store, by subtracting the minimum value 2 from each value yielding $(1, 0, 2, 3, 1, 0)$ in storage.

Figure 1 illustrates the outlier separation strategies over a series of values. The original Bit-packing needs $\lceil \log(x_{\max}+1) \rceil$ bits to store all the values, or $\lceil \log(x_{\max} - x_{\min} + 1) \rceil$ bits by subtracting the minimum value $x_{\min}$ from each value. PFOR separates the upper outliers greater than x_u, and uses $\lceil \log(x_u + 1) \rceil$ bits (or $\lceil \log(x_u - x_{\min} + 1) \rceil$ bits with $x_{\min}$ subtraction) to store the remaining values. We propose to further separate the lower outliers smaller than x_l. The remaining values thus occupies only $\lceil \log(x_u - x_l + 1) \rceil = \beta$ bits.

2) Outlier Separation Determination: Note that the separated outliers need some extra costs to indicate their indexes in the series, e.g., index 6 for value 0 and index 7 for value 8 in the example series. In other words, separating outliers may decrease the space of center values but introduce extra costs of indicating outlier positions. It thus needs a proper separation of outliers that would lead to lower total storage cost.

Unfortunately, existing methods use simple heuristics to determine outliers without considering the actual compression ratio performance. For example, NewPFOR [34] simply considers top 10% of values as outliers, and thus the storage

*Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

2375-026X/25/$31.00 ©2025 IEEE
DOI 10.1109/ICDE65448.2025.00107

cost of values is not necessarily small. Other algorithms, such as OptPFOR [34], try to find the outliers with bit-width distribution of values. Again, the estimation by distribution does not lead to lower storage cost either.

Even worse, we need to determine both thresholds x_u and x_l for separating the upper and lower outliers, respectively, denoted by two red dashed lines in Figure 1, which are not considered in the existing studies. Note that a bit-width β can represent 2^β distinct values. Intuitively, rather than enumerating all the values as possible x_l and x_u, we may search the separation in a small space of possible bit-widths α, β, γ for lower, center and upper values, respectively.

B. Contribution

In this paper, we propose *Bit-packing with Outlier Separation* (BOS) to improve Bit-packing, by separating both upper outliers and lower outliers. It is worth noting that BOS is complementary to the existing compression methods, such as RLE [11], SPRINTZ [2] and TS2DIFF [33]. By replacing their used Bit-packing operator with BOS, we have RLE+BOS, SPRINTZ+BOS, TS2DIFF+BOS, etc. Our major contributions in this paper are as follows.

(1) We formalize the optimization problem of outlier separation for bit-packing (BOS). Rather than simple heuristics, lower and upper outliers are formally defined with the corresponding data and position storage costs.

(2) We introduce the optimal separation based on values (BOS-V). It considers all the values from X as possible x_l and x_u for separating lower and upper outliers, with $O(n^2)$ search time. We propose proposition to ensure the correctness of the separation, i.e., with the minimum storage cost. The solutions based on values from X lead to a more efficient method on bit-width below.

(3) We devise a bit-width separation method (BOS-B). Note that we only need to consider the bit-widths β and γ for center values and upper outliers, in $O(n \log n)$ search time. We provide two propositions to illustrate all the possible cases such that the bit-width separation still returns the optimal solution as the value separation. The correctness is ensured by transforming the solutions based on values from X.

(4) We propose an approximate median separation (BOS-M). Observing the normal distribution of values, especially after delta processing, we consider median in separation. Together with the aforesaid bit-width, the approximate separation can be determined in $O(n)$ time.

(5) We conduct extensive experiments on many real world datasets. As summarized in the experiments, the compression ratio is significantly improved from about 2.75 by existing methods to 3.25 by our BOS-B on average. It is thus highly suggested to replace bit-packing by the proposed BOS in practice.

Our method BOS has been adopted in Apache IoTDB [29] and Apache TsFile [35], to replace Bit-packing. The code of the compression algorithm is included in the official GitHub repository of Apache IoTDB [15] and Apache TsFile [27] by system developers. The experiment related code and data are available in [10] for reproducibility.

II. Related Work

There are many algorithms to compress series data, including some lossy compression algorithms [8], [7], [16], and more importantly lossless algorithms [31], [32], [33].

A. Compression of Integer and Float

Integer Compression: Lossless compression algorithms of integer include run-length-based [11] and differential-based algorithms. The storage cost of run-length-based algorithm, such as RLE [11], is better than that of other algorithms when values have high repeatability. However, these algorithms perform worse on values with small consecutive repeat. Differential-based algorithms, including TS2DIFF [33] and SPRINTZ [2], perform better on the series of values with small delta. These algorithms subtract the previous data from the current data and remove redundant leading zeros with bit-packing to reduce the storage cost of values. However, when there are several outliers leading to larger bit-width of values, the storage cost of these algorithms is very high.

Float Compression: GORILLA [25], CHIMP [20], Elf [18], and BUFF [21] are compression algorithms designed for floating-point numbers. GORILLA [25] computes a XOR of the current and previous float values, and then compresses these XOR values. CHIMP [20] improves GORILLA with distribution of leading and trailing zeros, and Elf [18] eases trailing zeros with precision of floating-point before computing XOR. However, if there are several larger outliers in float datasets, these algorithms have to store larger XOR values. BUFF [21] uses sparse encoding to handle outliers of floats. Nevertheless, BUFF [21] only splits values into two parts, outliers and normal values according to frequency, and does not optimize the outlier separation.

B. Compression in Various Fields

Many research studies in signal processing/speech processing/data compression fields can be applied for the time series compression task. For example, 7-Zip [24] is a highly effective and efficient method for handling data compression. It is based on the LZMA (Lempel-Ziv-Markov chain algorithm) [23] compression algorithm, using dictionary compression and range encoding. LZ4 [5], derived from the LZ77 algorithm [36], searches for the longest matching string using a sliding window on the input stream. These data compression techniques for byte stream can be directly applied over the data encoded by bit-packing, i.e., complementary to our proposal, known as BOS+7-Zip or BOS+LZ4.

For signal and speech processing, frequency-based methods are often employed [30], e.g., DCT [3] to compress speech data and FFT [12] to compress signal data. Since time-frequency transform could be lossy, to enable lossless compression, the corresponding residuals need to be stored. Again, our proposal BOS can be applied to improve the storage of the residuals often with outliers, known as BOS+DCT or BOS+FFT, i.e., again complementary to the existing methods.

TABLE I: Notations

Notation	Description
X	a series
n	the number of values in a block of series
x_l, x_u	floor value and ceiling value in center values
X_c, X_l, X_u	center values, lower outliers and upper outliers
n_l, n_u	the number of lower outliers and upper outliers
α, β, γ	the bit-widths of lower, center and upper values
$C(x_l, x_u)$	storage cost with outlier separation
$c_i,\ c_i'$	the cumulative count

C. Compression with Outliers

Several previous compression schemes attempt to optimize the bit-packing algorithm by additionally handling outliers.

Patched Frame-of-Reference, PFOR: Zukowski et al. [37] propose the compression method to use a small bit-width *b* to bit-pack the center value and store outlier separately, but it does not compress additional outliers. PFOR stores the positions of outliers by organizing their indexes into lists. This solution may introduce a large number of compulsory outliers.

NewPFOR and OptPFOR: Two other algorithms are proposed by Yang et al. [34] to obtain better storage. They use a bit-width for 128 integers, and store low *b* bits of the outlier value, so that the compulsory outlier can be avoided. The difference between these two compression schemes lies in the strategy to determine *b*.

FastPFOR and SimplePFOR: To improve the compression effect of NewPFOR and accelerate it, Lemire and Boytsov [17] propose two algorithms FastPFOR and SimplePFOR. SimplePFOR compresses them together using Simple-8b, and FastPFOR classifies outliers according to the length of their high bits.

However, this family of PFOR algorithms still had many problems. The first is that all of these algorithms only consider upper outliers are shown in Figure 1. In this case, the *b* used to pack most of the center values will be greatly affected. Secondly, bitmap is not considered to store index of outliers. In some cases, bitmap could save the index storage. Finally, the value of each outlier point requires at least *b* bits to store the low bits. In fact, in our solution, it is very likely that less than *b* bits are needed to store the outlier value.

III. Problem Statement

In this section, we give some basic definitions about storage cost of bit-packing and outlier separation. The optimization problem of outlier separation is then formalized. Table I lists the frequently used notations.

A. Bit-packing Encoding

Bit-packing [19] specifies a fixed bit-width for all the values in a series. The corresponding storage cost is given as follows.

Definition 1 (Storage Cost)**.** *For a series $X = (x_1, \dots, x_n)$, its storage cost by Bit-packing is*

$$C(X) = n\lceil \log(x_{\max} - x_{\min} + 1)\rceil \tag{1}$$

x	634	727	723	686	648	...	465	640	770	...	675
bitmap	0	10	10	0	0	...	11	0	10	...	0

Fig. 2: Example of using bitmap to indicate the positions of outliers.

where $x_{\max} = \max X$ and $x_{\min} = \min X$ are the maximum and minimum values in the series X.

B. Outlier Separation

As shown in Figure 1, some large or small values increase storage cost in Definition 1. We propose to separate the outliers of both large and small values to store them separately, and thus reduce bit-widths of the remaining center values.

Specifically, we define lower bound of center values as x_l, and upper bound of center values as x_u. Based on x_l and x_u, all the values are split into 3 parts, including lower outliers, center values and upper outliers.

Definition 2 (Center Values)**.** *Center values X_c are a set of values which are in the range of spread (x_l, x_u),*

$$X_c = \{x_i \in X \mid x_l < x_i < x_u\}. \tag{2}$$

Center values are neither too larger nor too smaller with reduced bit-width $\lceil \log(\max X_c - \min X_c + 1)\rceil$.

Definition 3 (Lower Outliers)**.** *Lower outliers X_l are a set of values which are less than center values,*

$$X_l = \{x_i \in X \mid x_i \le x_l\}. \tag{3}$$

The bit-width of lower outliers is reduced from $\lceil \log(x_{\max} - x_{\min} + 1)\rceil$ to $\lceil \log(\max X_l - x_{\min} + 1)\rceil$. Thus, the storage cost of lower outliers is improved.

Definition 4 (Upper Outliers)**.** *Upper outliers X_u are a set of values which are larger than center values,*

$$X_u = \{x_i \in X \mid x_i \ge x_u\}. \tag{4}$$

The bit-width of upper outliers is decreased from $\lceil \log(x_{\max} - x_{\min} + 1)\rceil$ to $\lceil \log(x_{\max} - \min X_u + 1)\rceil$, Again, the storage cost of upper outliers is improved.

Let n_l and n_u be the number of the lower outliers and upper outliers in the series X, i.e., $n_l = |X_l|$ and $n_u = |X_u|$. To store lower outliers and upper outliers individually, we need to record the positions of outliers in the original series. Figure 2 gives an example of storing outlier index with bitmap. We write '0' for the index of center values, '10' for lower outliers, and '11' for upper outliers. In this case, the storage cost of index is $n + n_l + n_u$ bits.

C. Separation Problem

In the following, we formulate the outlier separation problem. Let us first introduce the storage cost with outlier separation. The storage of index for outliers incurs extra storage cost. The total cost of values contains index cost and value cost of lower outliers, upper outliers and center values.

Definition 5 (Storage Cost with Outlier Separation)**.** *The cost $C(x_l, x_u)$ of storing series X based on outlier separation by (x_l, x_u) is*

$$\begin{aligned} C(x_l, x_u) = & \, n_l(\lceil \log(\max X_l - x_{\min} + 1)\rceil + 1) \\ & + n_u(\lceil \log(x_{\max} - \min X_u + 1)\rceil + 1) \\ & + (n - n_l - n_u)\lceil \log(\max X_c - \min X_c + 1)\rceil + n, \end{aligned} \tag{5}$$

where $x_{\min}$ and $x_{\max}$ are the minimum and maximum values in the series X, having $x_{\min} < \max X_l < \min X_c < \max X_c < \min X_u < x_{\max}$.

If $\max X_l = x_{\min}$, the first term of $C(x_l, x_u)$ is $2n_l$. If $\min X_u = x_{\max}$, the second term of $C(x_l, x_u)$ is $2n_u$. If $\max X_c = \min X_c$, the third term of $C(x_l, x_u)$ is $(n - n_l - n_u)$. When $x_l < x_{\min}$ or $x_u > x_{\max}$, the number and bit-width of lower outliers or upper outliers are zero.

The outlier separation problem is to find the optimal range of center values with the minimum storage cost.

Problem 1 (Outlier Separation Problem)**.** *For a given series X, the outlier separation problem is to find the best (x_l, x_u) that minimizes the cost $C(x_l, x_u)$,*

$$\arg\min_{x_l, x_u} \; C(x_l, x_u). \tag{6}$$

Example 1. *Take the series in Figure 1 as an example. In the series, we set x_l as 620 and x_u as 794. Then, n_l and n_u are 5 and 4. Hence, the value cost is 698 and the cost of bitmap is 109. As a result, the storage cost is 807.*

IV. Exact Value Separation

Since the storage cost with outlier separation only depends on x_l and x_u, we could obtain the optimal solution by considering all the possible x_l and x_u. However, it takes too much time to consider each value from $x_{\min}$ to $x_{\max}$ for x_l and x_u. Thereby, we propose a separation algorithm by investigating only a set of values (BOS-V), still finding the optimal solution of outlier separation problem. The reason of introducing this baseline is as follows. (1) It illustrates the rationale of traversing the values in X for the optimal solution. which motivates the following algorithms. (2) It introduces some notations such as cumulative count, which are used in the following algorithms as well. (3) It is used to verify the correctness of the following advanced algorithm BOS-B, showing exactly the same compression results.

A. Optimal Separation with Values

According to Definition 5, since the cost of storing series X only depends on the values in the series, an optimal solution (x_l, x_u) must exist such that x_l and x_u are in the series X.

Proposition 1. *There must exist an optimal solution of outlier separation problem (x_u, x_l), where $x_l \in X$ and $x_u \in X$.*

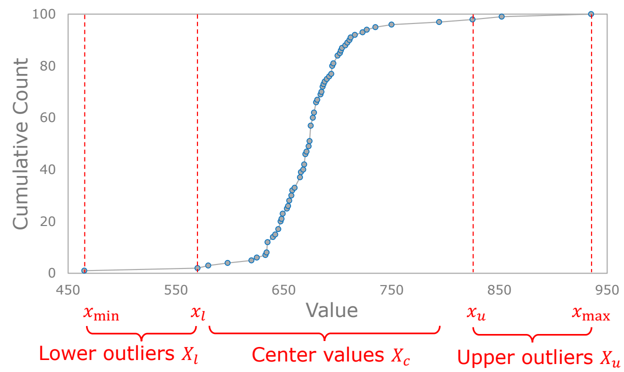


Fig. 3: Cumulative distribution function of values for outlier separation, where X_l, X_c, X_u denote the value separation for the series X in Figure 1.

Proof. For any optimal solution (x_l, x_u), we can always construct another solution $(\max X_l, \min X_u)$, which has the same cost as (x_l, x_u).

$$\begin{aligned} C(x_l, x_u) = & \, n_l(\lceil \log(\max X_l - x_{\min} + 1)\rceil + 1) \\ & + n_u(\lceil \log(x_{\max} - \min X_u + 1)\rceil + 1) \\ & + (n - n_l - n_u)\lceil \log(\max X_c - \min X_c + 1)\rceil + n \\ = & \, C(\max X_l, \min X_u). \end{aligned}$$

Note that the solution $(\max X_l, \min X_u)$ has $\max X_l \in X$ and $\min X_u \in X$. The conclusion is proved. $\square$

B. Cumulative Count

To calculate the storage cost $C(x_l, x_u)$ for each solution, traversing all the values of the series X to obtain n_l and n_u in Definition 5 is very costly. Hence, we maintain a cumulative count to reduce times of traversing. The definition of cumulative count of values is as follows.

Definition 6 (Cumulative Count)**.** *The cumulative count c_i or c'_i of a value is the number of values less than and equal to it*

$$\begin{aligned} c_i &= |\{x_j \mid x_j \le x_i, 1 \le j \le n\}|, \\ c'_i &= |\{x_j \mid x_j < x_i, 1 \le j \le n\}|. \end{aligned}$$

We present an example of cumulative count of values in Figure 3 for the series X from Figure 1. It is easy to see that lower outliers are in the left of the red line x_l, upper outliers are in the right of the red line x_u. Thus, we could get n_l and n_u with cumulative count efficiently.

Then, according to Definition 5, the value cost could be derived by cumulative count,

$$\begin{aligned} C(x_l, x_u) = & \, c_l(\lceil \log(\max X_l - x_{\min} + 1)\rceil + 1) \\ & + (n - c'_u)(\lceil \log(x_{\max} - \min X_u + 1)\rceil + 1) \\ & + (c'_u - c_l)\lceil \log(\max X_c - \min X_c + 1)\rceil + n. \end{aligned} \tag{7}$$

C. Value Separation Algorithm

Algorithm 1 presents the pseudo code of finding the optimal separation (x_l, x_u) with the minimum storage cost $C_{\min}$. First, we sort values in the series X in Line 1. Then, the cumulative count of each value in the series is calculated in Lines 2 and 3. Lines 4-10 get storage cost of each solution x_l, x_u with Formula 7, and find the one with the minimum storage cost.

Algorithm 1: Value Separation (BOS-V)

```
   Input:  Series X = (x_1, x_2, ..., x_n)
   Output: Optimal Solution (x'_l, x'_u)
1  X = Sort(X) ;
2  for x_i ← x_min to x_max do
3  |   Get c_i with Definition 6 ;
4  C_min = n * ⌈log(x_max − x_min + 1)⌉ ;
5  for x_i ← x_min to x_max do
6  |   for x_j ← x_max to x_i do
7  |   |   C_i = C(x_i, x_j) with Formula 7 ;
8  |   |   if C_i < C_min then
9  |   |   |   x'_l = x_i ;
10 |   |   |   x'_u = x_j ;
11 |   |   |   C_min = C_i ;
12 return (x'_l, x'_u) ;
```

Example 2. *Consider the series in Figure 1. First, we sort the series in ascending order and get cumulative count as shown in Figure 3. In the series, Algorithm 1 enumerates x_l from the minimum value 465 to the maximum 935, and x_u from the next value of x_l to the maximum value 935. Lastly, the algorithm finds the optimal solution (632, 696) with the minimum cost.*

D. Complexity Analysis

Algorithm 1 takes a time cost of $O(n \log n)$ to sort values, where n represents the number of values in X. After sorting values, the time cost of getting cumulative count is $O(n)$. The search of solution (x_l, x_u) with the minimum cost enumerates pairs of values in the series X in $O(n^2)$ time. In summary, the time complexity of Algorithm 1 is $O(n^2)$.

V. Exact Bit-width Separation

The quadratic time complexity of Algorithm 1 is still costly. Rather than values from X, we propose to use bit-width as the separation (BOS-B), reducing the time complexity to $O(n \log n)$. While the improved $O(n \log n)$ algorithm BOS-B is still concise, the foundation behind however is not-trivial. To find the optimal x_u, we need to prove that it is not necessary to traverse all the values in X in $O(n)$ time for each x_l. The novelty of the proposal is to give the solution determined by the bit-width β, which takes only $O(\log n)$ time. The technical depth roots in the existence of another better solution based on bit-width, for each solution (x_l, x_u) formed by values of X. The conclusion needs to be proved for two different cases. The complicated cost functions in Formulas 5 and 7, for center values, lower outliers and upper outliers, respectively, make the derivation difficult.

A. Optimal Separation with Bit-width

For any solution (x_l, x_u) with $x_l \in X$ and $x_u \in X$, let

$$\beta = \lceil \log(\max X_c - \min X_c + 1) \rceil, \tag{8}$$

$$\gamma = \lceil \log(x_{\max} - \min X_u + 1) \rceil, \tag{9}$$

denote the bit-widths of center values and upper outliers.

Proposition 2. *For any solution (x_l, x_u) with $\beta \leq \gamma$, $x_l \in X$ and $x_u \in X$, there always exists another solution (x_l, x'_u) having $C(x_l, x'_u) \leq C(x_l, x_u)$, where $x'_u = \min X_c + 2^\beta$.*

Proof. According to $\beta = \lceil \log(\max X_c - \min X_c + 1) \rceil$ in Formula 8, we have

$$\begin{aligned} \log(\max X_c - \min X_c + 1) &\leq \beta \\ \max X_c - \min X_c + 1 &\leq 2^\beta \\ \max X_c &< x'_u. \end{aligned}$$

(1) For $x_u > x'_u$, it follows $\max X_c < x'_u < x_u = \min X_u$. Since there is no value between $\max X_c$ and $\min X_u$ in X, according to Definitions 2 and 4, we have $\min X'_u = \min X_u$.

(2) For $x_u \leq x'_u$, referring to Definition 4, we have $\max X'_u \geq \max X_u$.

Combining the above two cases, we can conclude that

$$\min X'_u \geq \min X_u.$$

For $n_u = |X_u|$ and $n'_u = |X'_u|$ introduced after Definition 4, it follows $n_u \geq n'_u$. Let $n_\Delta = |X_u \setminus X'_u|$ be the size of the increment, having $n_\Delta = n_u - n'_u \geq 0$.

Given the same x_l and the corresponding identical X_l, n_l, we could get the difference C_Δ between $C(x_l, x'_u)$ and $C(x_l, x_u)$ defined in Formula 5,

$$\begin{aligned} C_\Delta = {} & C(x_l, x'_u) - C(x_l, x_u) \\ = {} & n_l(\lceil \log(\max X_l - x_{\min} + 1) \rceil + 1) \\ & + n'_u(\lceil \log(x_{\max} - \min X'_u + 1) \rceil + 1) \\ & + (n - n_l - n'_u)\lceil \log(\max X'_c - \min X'_c + 1) \rceil \\ & - n_l(\lceil \log(\max X_l - x_{\min} + 1) \rceil + 1) \\ & - n_u(\lceil \log(x_{\max} - \min X_u + 1) \rceil + 1) \\ & - (n - n_l - n_u)\lceil \log(\max X_c - \min X_c + 1) \rceil. \end{aligned} \tag{10}$$

The same x_l also infers $\min X'_c = \min X_c$. Together with $n_u = n'_u + n_\Delta$, we have

$$C_\Delta = C_1 - C_2, \tag{11}$$

where

$$\begin{aligned} C_1 = {} & (n - n_l - n_u)\lceil \log(\max X'_c - \min X_c + 1) \rceil \\ & + n_\Delta \lceil \log(\max X'_c - \min X_c + 1) \rceil \\ & + n'_u(\lceil \log(x_{\max} - \min X'_u + 1) \rceil + 1) \end{aligned}$$

and

$$\begin{aligned} C_2 = {} & (n - n_l - n_u)\lceil \log(\max X_c - \min X_c + 1) \rceil \\ & - n_\Delta(\lceil \log(x_{\max} - \min X_u + 1) \rceil + 1) \\ & - n'_u(\lceil \log(x_{\max} - \min X_u + 1) \rceil + 1) \\ = {} & (n - n_l - n_u)\beta - n_\Delta(\gamma + 1) - n'_u(\gamma + 1). \end{aligned}$$

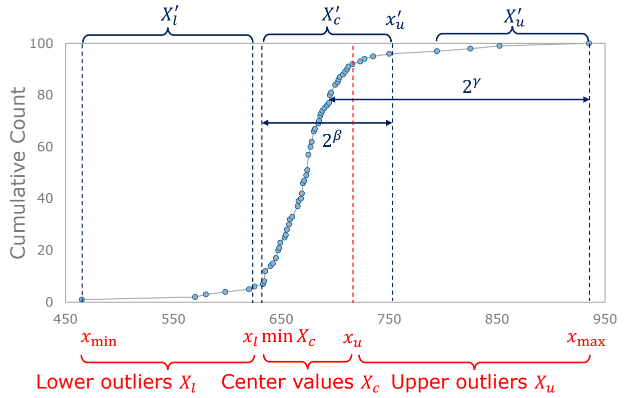


Fig. 4: Improving value separation (x_l, x_u) by bit-width separation (x_l, x'_u) using Proposition 2 with $\beta \le \gamma$.

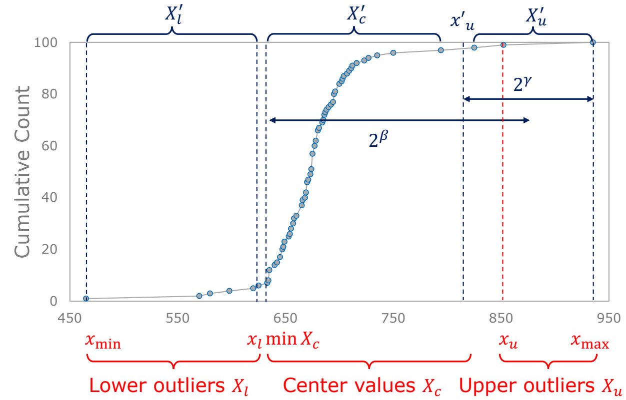


Fig. 5: Improving value separation (x_l, x_u) by bit-width separation (x_l, x'_u) using Proposition 3 with $\beta > \gamma$.

(i) Referring to Definition 2, we have $\max X'_c < x'_u = \min X_c + 2^\beta$. It follows

$$\lceil \log(\max X'_c - \min X_c + 1) \rceil \le \beta.$$

(ii) With the aforesaid proved $\min X'_u \ge \min X_u$, we infer

$$\lceil \log(x_{\max} - \min X'_u + 1) \rceil \le \lceil \log(x_{\max} - \min X_u + 1) \rceil = \gamma.$$

Applying the above two conditions, we further derive

$$\begin{aligned} C_\Delta &\le (n - n_l - n_u)\beta + n_\Delta \beta + n'_u(\gamma + 1) \\ &\quad - (n - n_l - n_u)\beta - n_\Delta(\gamma + 1) - n'_u(\gamma + 1) \\ &= n_\Delta(\beta - \gamma - 1) \le 0. \end{aligned}$$

Given $\beta \le \gamma$ and $n_\Delta \ge 0$, the conclusion is proved. □

Intuitively, as illustrated in Figure 4, all the points could be divided into 4 parts, lower outliers X_l, center values X_c, upper outliers moved from X_u to center values X'_c, and remaining upper outliers X'_u. During moving points, the cost of lower outliers X_l does not change, the bit-width of center values X_c is still β, and the bit-width of remaining upper outliers X'_u does not get larger. Moveover, the bit-width of upper outliers X_u moved to center values X'_c changes from γ to β, i.e., getting no larger given $\beta \le \gamma$. In summary, the cost of all the points becomes no greater, having $C(x_l, x'_u) \le C(x_l, x_u)$.

Proposition 3. *For any solution (x_l, x_u) with $\beta > \gamma$, $x_l \in X$ and $x_u \in X$, there always exists another solution (x_l, x'_u) having $C(x_l, x'_u) \le C(x_l, x_u)$, where $x'_u = x_{\max} - 2^\gamma + 1$.*

Proof. According to $\gamma = \lceil \log(x_{\max} - \min X_u + 1) \rceil$ in Formula 9, we have

$$x'_u \le \min X_u = x_u.$$

(1) For $x'_u = x_u = \min X_u$, it is exactly the (x_l, x_u) solution, having $\min X'_u = x'_u = \min X_u, \max X'_c = \max X_c$.

(2) For $\max X_c < x'_u < x_u = \min X_u$, since there is no value between $\max X_c$ and $\min X_u$ in X, according to Definitions 2 and 4, we have $\min X'_u = \min X_u$, $\max X'_c = \max X_c$ as well.

(3) For $x'_u \le \max X_c < x_u$, referring to Definitions 2 and 4, it follows $\max X'_c < \min X'_u \le \max X_c < \min X_u$.

Combining the above three cases, we can infer that

$$\min X'_u \le \min X_u, \max X'_c \le \max X_c.$$

For $n_u = |X_u|$ and $n_u = |X'_u|$ introduced after Definition 4, it follows $n'_u \ge n_u$. Let $n_\Delta = |X'_u \setminus X_u|$ be the size of the increment, having $n_\Delta = n'_u - n_u \ge 0$.

Similarly, we could get the difference C_Δ according to Formula 10. The same x_l also infers $\min X'_c = \min X_c$. Together with $n'_u = n_u + n_\Delta$, we have

$$C_\Delta = C_1 - C_2,$$

where

$$\begin{aligned} C_1 &= (n - n_l - n'_u)\lceil \log(\max X'_c - \min X_c + 1) \rceil \\ &\quad + n_\Delta(\lceil \log(x_{\max} - \min X'_u + 1) \rceil + 1) \\ &\quad + n_u(\lceil \log(x_{\max} - \min X'_u + 1) \rceil + 1) \end{aligned}$$

and

$$\begin{aligned} C_2 &= (n - n_l - n'_u)\lceil \log(\max X_c - \min X_c + 1) \rceil \\ &\quad - n_\Delta \lceil \log(\max X_c - \min X_c + 1) \rceil \\ &\quad - n_u(\lceil \log(x_{\max} - \min X_u + 1) \rceil + 1) \\ &= (n - n_l - n'_u)\beta - n_\Delta \beta - n_u(\gamma + 1). \end{aligned}$$

Applying two conditions similar to conditions (i) and (ii) in Proposition 2, we further derive

$$\begin{aligned} C_\Delta &\le (n - n_l - n'_u)\beta + n_\Delta(\gamma + 1) + n_u(\gamma + 1) \\ &\quad - (n - n_l - n'_u)\beta - n_\Delta \beta - n_u(\gamma + 1) \\ &= n_\Delta(\gamma + 1 - \beta) \le 0. \end{aligned}$$

Given $\beta > \gamma$ and $n_\Delta \ge 0$, the conclusion is proved. □

Intuitively, as illustrated in Figure 5, all the points could be divided into 4 parts, lower outliers X_l, upper outliers X_u, center values moved from X_c to upper outliers X'_u, and the remaining center values X'_c. Similar to Proposition 2, the cost of all parts of the points becomes no greater, having $C(x_l, x'_u) \le C(x_l, x_u)$.

TABLE II: All possible cases of pruning by separation with bit-width

Proposition	Condition	Solution
Proposition 2	$\beta \leq \gamma$	$(x_l, \min X_c + 2^\beta)$
Proposition 3	$\beta > \gamma$	$(x_l, x_{\max} - 2^\gamma + 1)$

B. Bit-width Separation Algorithm

According to Propositions 2 and 3, we could get a solution no worse than value separation, including the optimal solution, by traversing each value as x_l and the corresponding bit-width β or γ for x_u. Table II summarizes all the possible cases of $\beta \leq \gamma$ and $\beta > \gamma$, as well as their solutions to consider.

Algorithm 2 presents the pseudo code of bit-width separation (BOS-B). Firstly, same as Algorithm 1, we calculate cumulative count of values in Lines 1 - 3. Then, the algorithm enumerates the cost of each x_l and each corresponding β with $\beta \leq \gamma$ in Lines 5 - 12. The solution to consider is $(x_l, \min X_c + 2^\beta)$, according to the first case in Table II. Note that we traverse the bit-width β first. That is, for each β, the cumulative counts for x_l and $x_u = x_l + 2^\beta$ can be more efficiently fetched, given the fixed difference 2^β. For the second case in Table II, the algorithm enumerates the cost of each x_l and each γ, under the solution $(x_l, x_{\max} - 2^\gamma + 1)$ in Lines 15 - 21.

Algorithm 2: Bit-width Separation (BOS-B)

Input: Series $X = (x_1, x_2, \ldots, x_n)$
Output: Optimal Solution (x'_l, x'_u)

```
1  X = Sort(X) ;
2  for x_i ← x_min to x_max do
3  |   Get c_i with Definition 6 ;
4  C_min = n * ⌈log(x_max − x_min + 1)⌉ ;
5  for β ← 1 to ⌈log(x_max − x_{i+1} + 1)⌉ − 1 do
6  |   for x_i ← x_min to x_max do
7  |   |   x_u = x_{i+1} + 2^β ;
8  |   |   C_i = C(x_l, x_u) with Formula 7 ;
9  |   |   if C_i < C_min then
10 |   |   |   x'_l = x_l ;
11 |   |   |   x'_u = x_u ;
12 |   |   |   C_min = C_i ;
13 for x_i ← x_min to x_max do
14 |   x_l = x_i ;
15 |   for γ ← 1 to ⌈log(x_max − x_{i+1} + 1)⌉ − 1 do
16 |   |   x_u = x_max − 2^γ + 1 ;
17 |   |   C_i = C(x_l, x_u) with Formula 7 ;
18 |   |   if C_i < C_min then
19 |   |   |   x'_l = x_l ;
20 |   |   |   x'_u = x_u ;
21 |   |   |   C_min = C_i ;
22 return (x'_l, x'_u) ;
```

Example 3. *Consider the series in Figure 1. First, we sort the series in ascending order and get cumulative count, similar to Algorithm 1. In the series, Algorithm 2 enumerates x_l from the minimum value 465 to the maximum 935. For each $x_l = x_i$ and $\beta \leq \gamma$, i.e., $\beta \leq \lceil \log(x_{\max} - x_{i+1} + 1)/2 \rceil = \lceil \log(935 - x_{i+1} + 1) \rceil - 1$, we consider the cost of $x_u = x_{i+1} + 2^\beta$ as Figure 4. For each $x_l = x_i$ and $\beta > \gamma$, i.e., $\gamma \leq \log(x_{\max} - x_{i+1} + 1)/2] = \lceil \log(935 - x_{i+1} + 1) \rceil - 1$, we consider the cost of $x_u = x_{\max} - 2^\gamma$ + 1 as Figure 5. Finally, the algorithm finds the optimal solution with $x_l = 632$ and $\beta = 6$.*

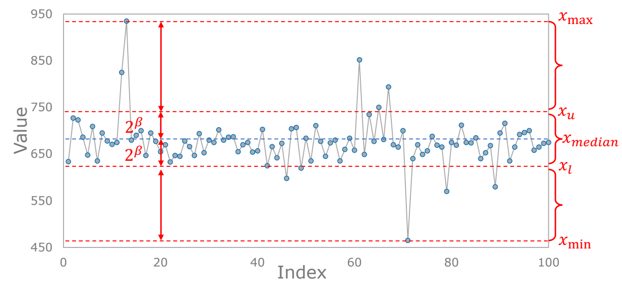


Fig. 6: Separation by x_{median} with bit-width β in both sides.

C. Complexity Analysis

In Algorithm 2, the time cost of sorting values and getting cumulative count is $O(n \log n)$, similar to Algorithm 1. Then, it takes $O(n \log n)$ time to calculate cumulative count of $x_{i+1} + 2^\beta$ with cumulative count c_i, for each x_i and β. With the fixed $x_{\max}$, the calculation for cumulative count of $x_{\max} - 2^\gamma + 1$ takes $O(n)$ time. Finally, it takes $O(n \log n)$ time to find the minimum cost by enumerating x_l and β as well as γ. The overall complexity of Algorithm 2 is $O(n \log n)$.

VI. Approximate Median Separation

Algorithm 2 of bit-width separation still needs to traverse all possible values as x_l. In this section, we further narrow down the search space of x_l to the candidates determined by the median of X and bit-width β. This is motivated by the observation that many datasets (after pre-processing) follow a normal distribution, as illustrated in Figure 8 below.

A. Approximate Separation with Median

Let x_{median} be the median of X. As illustrated in Figure 6, the center values are heuristically determined by $x_l = x_{\text{median}} - 2^\beta$ and $x_u = x_{\text{median}} + 2^\beta$, for possible bit-width β. To efficiently calculate the storage cost in Formula 5, instead of the cumulative count, we define the count of buckets divided by median x_{median} and bit-width β as follows.

Definition 7 (Bucket Count)**.** *The bucket count $h(\beta)$ is the number of values exceeding x_{median} in bit-width β,*

$$h(\beta) = |\{x_i \in X \mid x_{\text{median}} + 2^{\beta-1} \leq x_i < x_{\text{median}} + 2^\beta\}|.$$

The bucket count $h(-\beta)$ is the number of values less than x_{median} with bit-width β,

$$h(-\beta) = |\{x_i \in X \mid x_{\text{median}} - 2^\beta < x_i \leq x_{\text{median}} - 2^{\beta-1}\}|.$$

The special bucket count is $h(0) = |\{x_i \in X \mid x_i = x_{\text{median}}\}|$.

B. Median Separation Algorithm

We present the approximate median separation algorithm (BOS-M) in Algorithm 3. First, we use a fast approximate median implementation [14] of QuickSelect algorithm [13] to find median in Line 1. Then, we divide all the values for bucket count $h(\beta)$ in Lines 2-10. Finally, it computes the storage cost of solution $(x_{\text{median}} - 2^\beta, x_{\text{median}} + 2^\beta)$ for various bid-width β and finds the minimum.

Algorithm 3: Median Separation (BOS-M)

Input: Series $X = (x_1, x_2, \ldots, x_n)$
Output: Approximate Solution (x'_l, x'_u)

1 x_{median} = FindMedian(X) ;
2 **for** $i \leftarrow 1$ **to** n **do**
3   **if** $x_i < x_{\text{median}}$ **then**
4     $\beta = \lceil \log(x_{\text{median}} - x_i + 1) \rceil$;
5     $h(-\beta) = h(-\beta) + 1$;
6   **else if** $x_i > x_{\text{median}}$ **then**
7     $\beta = \lceil \log(x_i - x_{\text{median}} + 1) \rceil$;
8     $h(\beta) = h(\beta) + 1$;
9   **else**
10     $h(0) = h(0) + 1$;
11 $C_{\min} = n * \lceil \log(x_{\max} - x_{\min} + 1) \rceil$;
12 **for** $\beta \leftarrow \lceil \log(x_{\max} - x_{\min} + 1) \rceil$ **to** 1 **do**
13   $n_l = n_l + h(-\beta)$;
14   $n_u = n_u + h(\beta)$;
15   $x_i = x_{\text{median}} - 2^\beta$;
16   $x_j = x_{\text{median}} + 2^\beta$;
17   $C_\beta = C(x_i, x_j)$ with n_l, n_u and Formula 5 ;
18   **if** $C_\beta < C_{\min}$ **then**
19     $x'_l = x_i$;
20     $x'_u = x_j$;
21     $C_{\min} = C_\beta$;
22 **return** (x'_l, x'_u) ;

Example 4. *Consider the series in Figure 1. First, Algorithm 3 finds the median $x_{\text{median}} = 674$. Given the maximum $\beta = 9$, it divides values into 19 buckets. Then, the algorithm searches the bit-width β with the minimum cost by enumerating β from 9 to 1. It returns the solution $(610, 738)$ with $\beta = 6$.*

C. Approximation Performance

While it is difficult to bound the approximation ratio in general, given the various data distributions, we obtain some theoretical guarantee for the special case of normal distribution. The full proof of Proposition 4 can be found in the appendix [1].

Let C_{opt} be the storage cost of the optimal solution for outlier separation problem, and C_{approx} be the storage cost of the solution x_l and x_u returned by the heuristic BOS-M. Since many real-world datasets follow the normal distribution as illustrated in Figure 8, we study the theoretical bound of approximation ratio $\rho = \frac{C_{\text{approx}}}{C_{\text{opt}}}$ under the normal distribution.

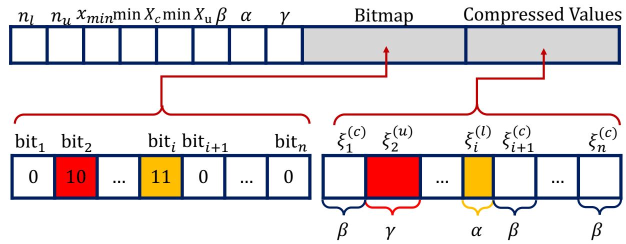


Fig. 7: Storage layout of bit-packing with outlier separation (BOS).

Proposition 4. *For normal distribution $X \sim N(\mu, \sigma^2)$, with probability 0.997, the approximation ratio ρ of BOS-M satisfies*

$$\rho \leq \begin{cases} 2 & \text{if } \sigma \leq \frac{5}{3}, \\ \lceil \log(3\sigma - 1) \rceil & \text{otherwise.} \end{cases}$$

D. Complexity Analysis

In Algorithm 3, it takes $O(n)$ amortized time complexity for the faster approximate median implementation [14] of QuickSelect algorithm [13] to find median [13]. Then, the algorithm takes $O(n)$ time to divide all values into buckets, and $O(\log n)$ time to calculate the storage cost of each solution with bit-width β. In summary, the time complexity of approximate median separation is $O(n)$.

VII. System Deployment

We implement BOS in Apache IoTDB [29] and Apache TsFile [35], and the code is available in the GitHub repository of the systems [15] and [27]. In the section, we introduce the storage layout of data compressed by BOS in the file format.

Figure 7 presents the storage structure of BOS in the file format. First, a block of values starts with some meta data of the series, including the number of outliers n_l and n_u, the minimum value $x_{\min}$, the minimum center value $\min X_c$ and the minimum upper outlier $\min X_u$. It follows the bit-width of center values $\beta = \lceil \log(\max X_c - \min X_c + 1) \rceil$, bit-width of lower outliers $\alpha = \lceil \log(\max X_l - x_{\min} + 1) \rceil$, and bit-width of upper outliers $\gamma = \lceil \log(x_{\max} - \min X_u + 1) \rceil$. Then, we store the index of outliers with bitmap as shown in Figure 2, where bit_i is the indicator of i-th value.

The lower outliers, center values and upper outliers are stored together in the original data order. Their corresponding bit-widths, α, β, γ, are marked by a bitmap. Consequently, the decompression process only needs to scan the data once. In Figure 7, center values $\xi_i^{(c)}$ stores $x_i - \min X_c$ in the blue boxes, lower outliers and upper outliers are stored as $\xi_i^{(l)} = x_i - x_{\min}$ and $\xi_i^{(u)} = x_i - \min X_u$ in the red and yellow boxes, respectively.

VIII. Experiment

In this section, we experimentally compare the compression ratio and time of our BOS with other algorithms, and we further validate the motivation, variation evaluation, and scalability of our method BOS.

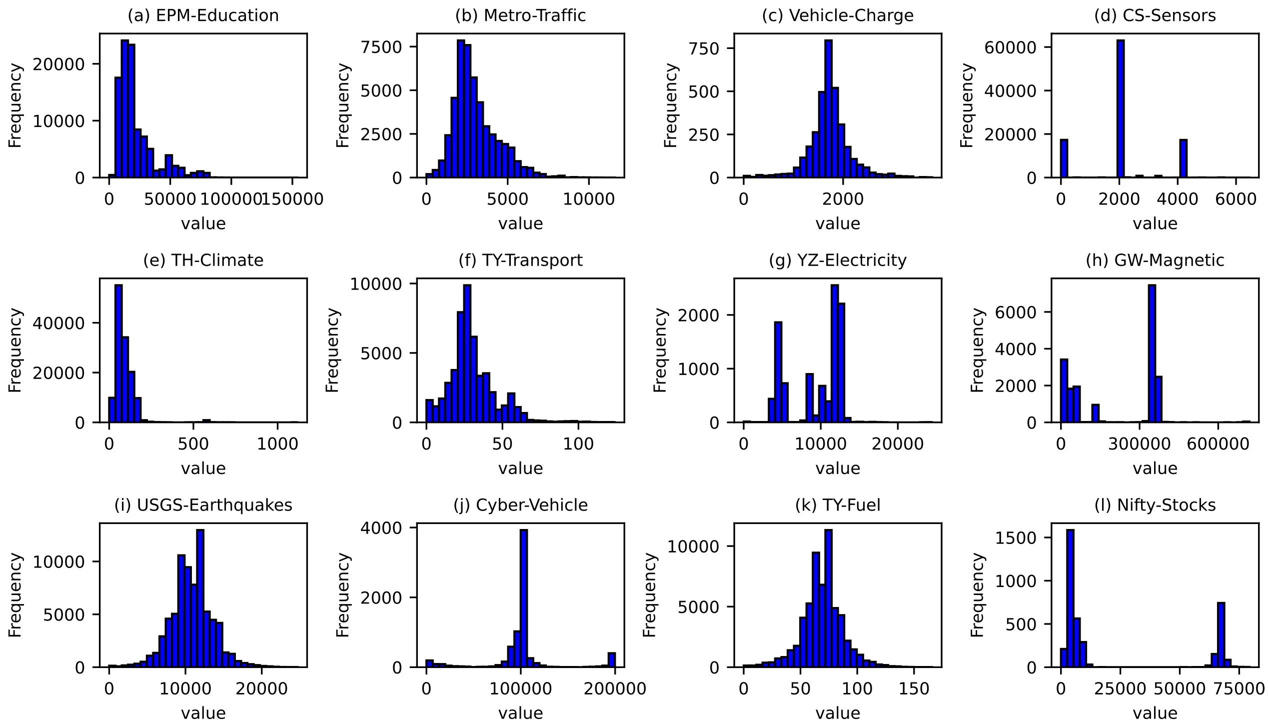


Fig. 8: Value distribution of all datasets after TS2DIFF.

TABLE III: Real-world datasets

Dataset	Abbr.	Public	# Values	Data Type
EPM-Education	EE	[9]	900,000	Integer
GW-Magnetic	GM	[22]	933,984	Float
Metro-Traffic	MT	[28]	48,204	Integer
Nifty-Stocks	NS	[26]	295,193,088	Float
USGS-Earthquakes	UE	[6]	683,290	Float
Vehicle-Charge	VC	[4]	3,396	Integer
CS-Sensors	CS		100,000	Integer
Cyber-Vehicle	CV		35,676,900	Float, Integer
TH-Climate	TC		131,747	Integer
TY-Fuel	TF		183,556,352	Float, Integer
TY-Transport	TT		16,596,252	Integer
YZ-Electricity	YE		10,108	Float

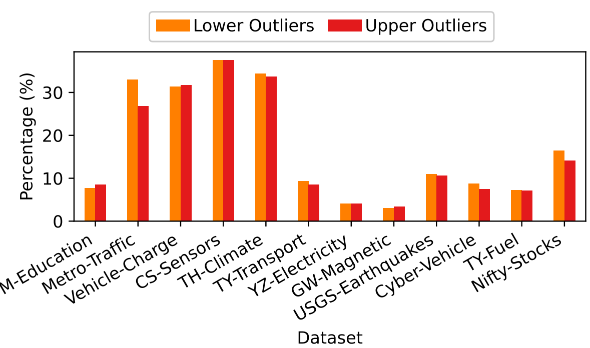


Fig. 9: Percentage of lower and upper outliers separated by BOS-V

A. Experimental Setting

The experiments were conducted on an Apple M1 Pro chip, featuring 8 CPU cores and 14 GPU cores, complemented by 16GB of unified memory.

1) Baselines: According to Section II of related work, we select several state-of-the-art algorithms in comparison, including floating-point compression algorithms (Gorilla [25], Chimp [20], Elf [18] and BUFF [21]) and integer encoding algorithms (RLE [11], SPRINTZ [2] and TS2DIFF [29]). Note that RLE, SPRINTZ and TS2DIFF use bit-packing, and thus are also denoted as RLE+BP, SPRINTZ+BP and TS2DIFF+BP.

We compare our algorithms, BOS with value separation (BOS-V), bit-width separation (BOS-B) and approximate median separation (BOS-M) with PFOR [37], NEWPFOR [34], OPTPFOR [34] and FASTPFOR [17], which also handle outliers in bit-packing. They can cooperate with other compression methods as well, by replacing BOS, e.g., RLE+BOS-V vs RLE+PFOR.

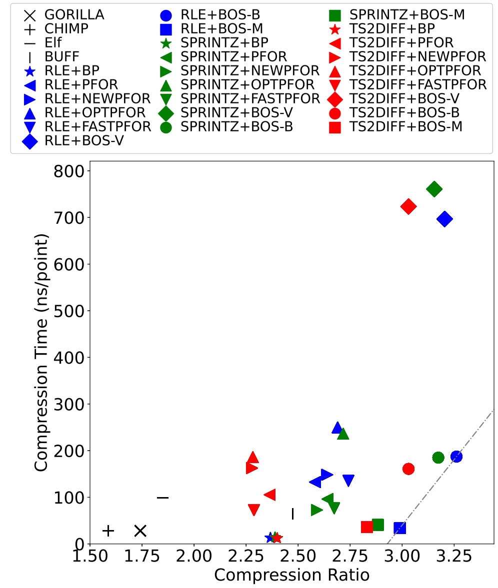
2) Datasets: Real world datasets utilized in our experimental evaluation encompass both publicly available data and the data acquired by our partners in various industries. The full dataset names in Table III indicate the corresponding domains.

Data types and the number of values in these datasets are shown in Table III. Some datasets contain only integers, where all the compression algorithms can be applied directly. There are also some datasets that contain floating-point numbers.

(a) Compression ratio on various datasets

Methods		Datasets without float						Datasets with float					
		EE	MT	VC	CS	TC	TT	YE	GM	UE	CV	TF	NS
Float	GORILLA	1.67	2.23	1.94	1.71	6.39	3.14	1.20	1.58	0.96	1.53	2.02	1.19
	CHIMP	1.36	1.72	1.47	1.73	4.58	2.17	1.22	1.87	1.07	1.62	1.84	1.39
	Elf	1.63	1.80	1.61	1.85	4.47	2.67	1.60	2.10	1.49	2.08	2.15	1.37
	BUFF	1.92	2.46	2.66	2.67	3.20	4.44	2.28	1.68	2.04	2.54	2.96	1.56
RLE+	BP	1.92	2.46	2.65	2.66	3.20	4.67	2.13	2.15	1.00	1.74	3.11	1.56
	PFOR	2.66	2.39	2.76	2.59	4.31	4.80	1.95	2.06	2.02	2.83	2.89	1.44
	NEWPFOR	2.76	2.38	2.86	2.58	4.35	4.78	2.34	2.11	2.05	2.84	2.88	1.46
	OPTPFOR	2.81	2.38	2.86	2.59	4.36	4.96	2.38	2.12	2.12	2.92	2.95	1.51
	FASTPFOR	2.79	2.40	2.91	2.63	4.40	4.82	2.51	2.22	2.23	2.92	3.04	1.70
	BOS-V / B	**3.03**	2.48	**3.16**	**5.23**	4.83	**5.67**	2.65	2.37	**2.41**	**3.61**	**3.52**	1.74
	BOS-M	2.82	2.45	3.15	3.42	4.74	5.50	2.50	2.33	2.37	3.30	3.32	1.72
SPRINTZ+	BP	2.05	2.36	2.64	2.44	3.94	4.07	2.12	1.67	1.95	2.40	2.76	1.81
	PFOR	2.37	2.45	2.60	2.98	4.64	4.17	2.88	2.43	1.97	2.59	2.70	1.91
	NEWPFOR	2.42	2.46	2.62	2.45	4.62	4.18	2.78	2.35	1.99	2.63	2.75	1.88
	OPTPFOR	2.46	2.47	2.64	2.88	5.46	4.29	2.79	2.68	2.07	2.83	2.86	1.91
	FASTPFOR	2.41	2.45	2.60	2.95	4.93	4.10	2.69	2.57	2.09	2.76	2.86	1.92
	BOS-V / B	2.67	**2.87**	2.90	4.67	7.00	4.97	2.95	**2.86**	2.26	3.34	3.34	**2.07**
	BOS-M	2.51	2.62	2.84	3.57	6.49	4.64	2.83	2.52	2.20	2.89	3.09	2.00
TS2DIFF+	BP	2.05	2.38	2.64	2.44	4.00	4.08	2.15	1.69	1.96	2.41	2.76	1.82
	PFOR	2.00	2.42	2.60	2.42	4.29	4.08	2.08	1.66	1.90	2.38	2.68	1.77
	NEWPFOR	1.93	2.36	2.47	2.35	4.03	3.85	2.05	1.60	1.84	2.30	2.59	1.72
	OPTPFOR	1.93	2.36	2.47	2.36	4.04	3.86	2.05	1.60	1.85	2.30	2.60	1.72
	FASTPFOR	1.97	2.32	2.41	2.34	3.96	3.70	2.19	1.63	1.85	2.33	2.65	1.76
	BOS-V / B	2.56	2.66	2.83	4.49	**7.09**	4.86	**2.97**	2.64	2.19	3.12	3.16	2.04
	BOS-M	2.53	2.63	2.80	3.46	6.84	4.73	2.92	2.51	2.17	3.05	3.08	2.03

(b) Average performance of different algorithms



(c) Compression time and decompression time on various datasets

Methods		Compress datasets without float						Compress datasets with float					
		EE	MT	VC	CS	TC	TT	YE	GM	UE	CV	TF	NS
Float	GORILLA	19	**15**	99	**16**	**8**	12	18	17	32	19	14	58
	CHIMP	22	23	77	22	16	16	22	22	26	20	20	52
	Elf	59	130	59	255	40	42	101	104	142	59	82	65
	BUFF	61	63	81	50	31	33	53	78	61	46	47	148
RLE+	BP	12	17	12	19	9	11	14	14	15	12	13	16
	PFOR	127	147	121	215	106	97	134	112	144	107	110	154
	NEWPFOR	124	160	116	299	100	89	134	112	143	105	115	244
	OPTPFOR	312	295	231	471	152	152	186	163	247	221	250	232
	FASTPFOR	103	158	103	283	106	82	125	132	122	93	100	197
	BOS-V	480	1131	423	82	40	63	957	1106	819	431	381	1801
	BOS-B	153	216	118	86	36	44	182	271	205	119	110	568
	BOS-M	38	26	32	70	25	28	29	30	43	29	32	26
SPRINTZ+	BP	15	24	11	24	11	12	13	14	16	12	14	12
	PFOR	79	106	62	266	56	49	74	119	92	76	66	82
	NEWPFOR	60	93	54	132	37	37	59	67	79	56	55	118
	OPTPFOR	232	280	222	373	96	117	181	205	327	200	215	258
	FASTPFOR	51	194	51	96	34	43	95	56	78	47	47	92
	BOS-V	960	989	769	128	39	137	769	1061	1505	515	554	992
	BOS-B	201	212	155	94	32	56	182	311	265	134	134	303
	BOS-M	39	45	32	74	19	33	52	31	45	37	40	33
TS2DIFF+	BP	**11**	24	**10**	23	10	**10**	**13**	**11**	**11**	**10**	**10**	**11**
	PFOR	95	232	63	213	68	55	77	97	111	78	68	80
	NEWPFOR	81	109	63	522	51	40	85	88	131	65	64	551
	OPTPFOR	125	201	139	569	93	90	144	142	180	162	115	194
	FASTPFOR	72	84	55	145	44	37	57	91	87	54	52	72
	BOS-V	929	944	738	74	37	131	722	1016	1452	500	535	927
	BOS-B	183	178	136	73	29	49	146	263	248	122	126	256
	BOS-M	28	51	27	54	50	30	23	30	31	69	39	28

Methods		Decompress datasets without float						Decompress datasets with float					
		EE	MT	VC	CS	TC	TT	YE	GM	UE	CV	TF	NS
Float	GORILLA	**1**	**1**	4	4	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**2**
	CHIMP	1	1	4	3	1	1	1	1	1	1	1	3
	Elf	38	59	37	126	21	29	101	103	148	77	56	45
	BUFF	1	2	**3**	**2**	1	1	1	2	1	1	1	3
RLE+	BP	10	11	9	12	9	9	10	9	11	8	9	12
	PFOR	33	67	33	49	34	35	33	31	34	36	37	37
	NEWPFOR	32	50	33	58	33	34	33	36	37	36	41	41
	OPTPFOR	32	37	33	65	33	34	44	34	35	44	36	35
	FASTPFOR	32	64	35	130	43	35	35	61	41	37	37	90
	BOS-V	35	28	28	51	23	23	26	32	33	30	25	37
	BOS-B	23	16	13	40	22	17	15	22	21	24	17	14
	BOS-M	19	10	14	44	13	15	16	25	27	13	16	13
SPRINTZ+	BP	14	14	12	20	10	10	12	12	13	11	11	11
	PFOR	40	50	38	71	37	39	42	53	40	41	39	40
	NEWPFOR	39	52	40	60	39	39	42	46	46	41	42	41
	OPTPFOR	40	72	40	117	42	39	48	47	47	50	41	43
	FASTPFOR	40	65	42	79	39	39	43	47	57	43	42	44
	BOS-V	41	101	41	72	33	34	54	43	44	39	40	43
	BOS-B	23	44	20	37	18	18	34	22	25	19	22	23
	BOS-M	23	25	17	38	14	22	38	20	26	25	24	18
TS2DIFF+	BP	12	14	12	19	12	11	12	12	12	12	11	12
	PFOR	37	74	37	81	37	37	47	39	58	35	35	35
	NEWPFOR	35	53	37	77	34	35	37	35	41	35	37	43
	OPTPFOR	37	58	37	49	36	37	73	40	39	40	37	36
	FASTPFOR	35	63	39	61	33	33	44	49	45	38	36	47
	BOS-V	23	45	27	38	28	24	22	27	30	30	33	25
	BOS-B	24	63	27	62	28	28	22	27	31	30	34	25
	BOS-M	13	33	12	25	14	19	7	11	17	16	19	11

Fig. 10: Compression ratio and time of applying bit-packing with outlier separation (BOS) in different compression methods.

Algorithms designed for integers, such as RLE, SPRINTZ and TS2DIFF, first convert float into integer by scaling 10^p, where p is the precision of the original floating-point data [21].

We draw the value distribution of all datasets after TS2DIFF in Figure 8. Since TS2DIFF removes trend by differencing values, most datasets after TS2DIFF follow normal distribution including datasets EE, MT, VC, TC, TT, UE, CV and TF. Moreover, owing to the existence of outliers, there are still some extreme delta values in the intermediate series by TS2DIFF, e.g., in TH.

As the data distribution illustrated in Figure 8, outliers commonly exist in real datasets. We count the corresponding number of lower and upper outliers separated by BOS-V in each dataset in Figure 9. Even for those datasets with a relatively small proportion of outliers, by separating them, the compression ratio could still be significantly improved. Therefore, the outlier issue is general and worthwhile to address in compression.

3) Metric: We compare the compression ratio with other methods, which measures the ratio of uncompressed data size to compressed data size, $compressionRatio = \frac{uncompressedSize}{compressedSize}$.

We also evaluate the compression and decompression time per value (ns/points) by different algorithms. Each experiment is conducted 500 times and report the average.

B. Comparison with Existing Methods

In the section, we compare performance of our proposals combined and compared with others. The compression ratio of algorithms is shown in Figure 10a. The corresponding compression time and decompression time are presented in Figure 10c. Figure 10b presents a summary of average compression ratio and time of each algorithm on all the datasets.

1) Compression Ratio: In Figure 10a, the red compression ratio is the best for the dataset in each column. As shown, the compression ratio of algorithms combined with BOS-V or BOS-B is always the best on all the datasets. In Figure 10b, BOS-B shows exactly the same compression ratio as BOS-V, verifying its correctness of returning the optimal solution. When combined with RLE or SPRINTZ, BOS-M has an overall performance better than the PFOR baseline and its variations. Although TS2DIFF+BOS-M might not outperform some others, its compression ratio is still better than the PFOR baselines combined with TS2DIFF. The reason is that the output of TS2DIFF follows normal distribution as described in Section VIII-A2, where our median separation performs.

For a normal distribution (after TS2DIFF), e.g., Figure 8(c) Vehicle-Charge, the approximate median separation works well, i.e., the compression ratio of (TS2DIFF+)BOS-M is similar to that of BOS-V/B in Figure 10a (datasets VC). However, for other distributions such as skew, e.g., Figure 8(e) TH-Climate, there are a large number of low outliers in a very small range. It is difficult for BOS-M to find the proper separation of lower outliers by only enumerating bit-width β. Consequently, (TS2DIFF+)BOS-M is much worse than BOS-V/B in Figure 10a (datasets TC).

2) Compression Time and Decompression Time: As shown in Figure 10c, compression with value separation is very slow, since the time cost is high to sort all the values and enumerate value pairs as possible solutions. BOS-B with bit-width separation has lower time cost than BOS-V. The result is not surprising, given the time complexity reduced from $O(n^2)$ to $O(n \log n)$. Finally, BOS-M with approximate median separation in $O(n)$ time has comparable compression time cost as other baselines, while its compression ratio is better, as illustrated in Figure 10b.

As for decompression time, there is no clear difference observed between our BOS and the PFOR baselines with outlier separation. It is due to the same $O(n)$ time cost in decompression.

3) Trade-off between Compression Ratio and Time: As illustrated in Figure 10b, the optimal solution BOS-B such as RLE+BOS-B has much better compression ratio than other algorithms, but is a bit slower in compression time. The linear time approximation BOS-M, e.g., RLE+BOS-M, achieves significantly lower compression time, and slightly weaker compression ratio (still outperforming baselines), i.e., a practical trade-off.

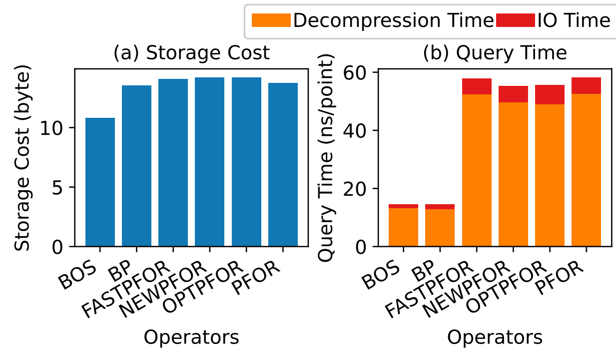


Fig. 11: Storage and query cost by various bit-packing operators in TS2DIFF.

C. Motivation Validation

1) Storage and Query Cost: To demonstrate the advantage of employing the operator, we perform an experiment to report the average storage and query processing cost over all datasets. With a better compression ratio in Figure 10a, our BOS operator yields lower storage costs as shown in Figure 11. It leads to lower IO costs and thus query processing time comparable to the simple bit-packing operator (BP).

2) Lower Outlier Separation: It is true that the number of lower outliers could be small in some datasets, such as GW-Magnetic and YZ-Electricity, illustrated in Figure 9. While the overall storage cost for them may not be significant, they could affect the storage of other center values if not separated. The reason is that as illustrated in Figure 1 and presented in Formula 5, the storage cost is determined by the minimum value of a set, i.e., lower outliers if not separated. Figure 12 reports the results of BOS by terminating the loop early without enumerating possible values for separating lower outliers, i.e., considering upper outliers only. As shown, even for those datasets with a relatively small proportion of lower outliers, such as the aforementioned GW-Magnetic and YZ-Electricity, considering both upper and lower outliers could have better compression ratio than separating upper outliers only (without considering lower outliers).

D. Variation Evaluation

1) Complement to Other Compression Methods: We conduct an experiment to compare with the compression techniques in signal processing/speech processing/data compression fields. BOS as a fundamental bit-packing operator is complementary to these existing compression methods. Therefore, we compare compression ratio and time of 7-Zip [24], LZ4 [5], DCT [3], FFT [12] with and without our BOS in Figure 13. As shown, by combining these four compression algorithms with our BOS, the compression ratios are all improved, of course with some extra overhead.

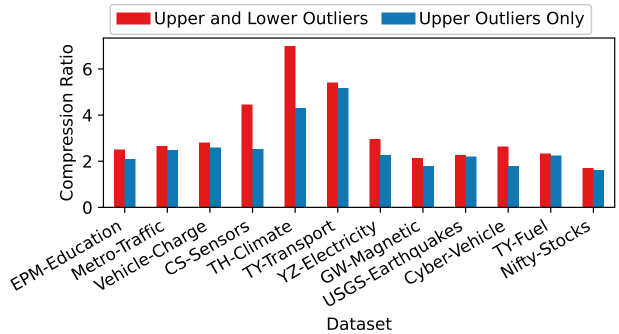


Fig. 12: Evaluating BOS terminating early without enumerating lower outliers.

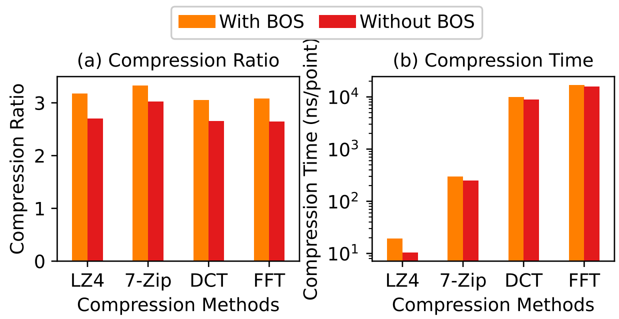


Fig. 13: Combining BOS with general data compression methods.

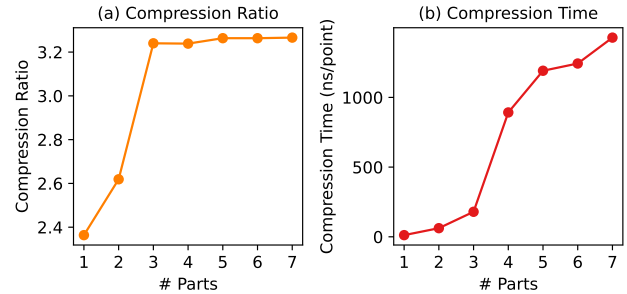


Fig. 14: Varying the number of divided value parts.

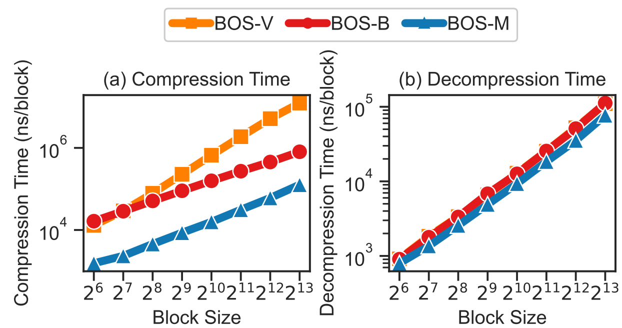


Fig. 15: Compression and decompression time by varying block size n.

2) Varying the Parts: We conduct an experiment about compression ratio and time by varying the number of divided parts in Figure 14. When the number of parts increases from 1 to 3, the compression ratio improves significantly. It verifies the intuition of our proposal in dividing the data into 3 parts, lower outliers, center values and upper outliers. However, the improvement is marginal by further dividing from 3 to 6 parts, given the close center values. Unfortunately, the corresponding compression time increases considerably. Therefore, we recommend to divide the space into 3 parts as shown in Figure 1.

E. Scalability

We conduct an experiment on the average compression time and decompression time over all datasets of BOS-V, BOS-B and BOS-M, by varying block size n, in Figure 15. All the methods increase almost linearly owing to the existence of duplicate values in the datasets. The advanced BOS-B increases much slower than BOS-V, while the approximate BOS-M is the most efficient.

It is not surprising that the decompression time increases linearly with the block size n, as illustrated in Figure 15b. BOS-M has less decompression time, since it separates fewer outliers.

IX. Conclusion

In this paper, we propose Bit-packing with Outlier Separation (BOS), which improves compression ratio of algorithms using bit-packing, by storing the outliers separately. It separates not only the upper outliers, occupying a large bit-width, but also the lower outliers, which waste the bit-width of center values as well. In order to determine a proper separation of outliers for better compression ratio, we devise an optimal separation strategy by enumerating the values in $O(n^2)$ time, known as the value separator (BOS-V). With Propositions 2 and 3, the efficiency is improved by considering bit-width as the separator (BOS-B), still returning the optimal solution but taking only $O(n \log n)$ search time. To further reduce the time cost, we propose an approximate median separation (BOS-M) in $O(n)$ time. Experiments on real world datasets demonstrate that BOS-B with bit-width separation shows significantly higher compression ratio than existing methods, and lower compression time than the value separation BOS-V. As summarized in Figure 10b, together with RLE, BOS-M with approximate median separation achieves relatively high compression ratio and low compression time. In short, our proposal BOS is highly suggested to replace bit-packing in compression algorithms, which indeed has been adopted in Apache IoTDB and Apache TsFile.

Acknowledgment

This work is supported in part by the National Key Research and Development Plan (2021YFB3300500), NSFC (92267203, 62021002, 62072265, 62232005), State Grid 5700-202435261A-1-1-ZN, and Beijing Key Laboratory of Industrial Big Data System and Application. Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

References

[1] Appendix. https://github.com/thssdb/encoding-outlier/blob/main/appendix.pdf, 2024.

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

[3] Din-Yuen Chan, Jar-Ferr Yang, and Chun-Chin Fang. Fast implementation of MPEG audio coder using recursive formula with fast discrete cosine transforms. *IEEE Trans. Speech Audio Process.*, 4(2):144–148, 1996.

[4] Charge. https://www.kaggle.com/datasets/michaelbryantds/electric-vehicle-charging-dataset, 2024.

[5] Yann Collet. Lz4: Extremely fast compression algorithm. https://lz4.github.io/lz4/, 2013. Available online.

[6] Earthquakes. https://www.kaggle.com/datasets/thedevastator/uncovering-geophysical-insights-analyzing-usgs-e, 2024.

[7] Frank Eichinger, Pavel Efros, Stamatis Karnouskos, and Klemens Böhm. A time-series compression technique and its application to the smart grid. *VLDB J.*, 24(2):193–218, 2015.

[8] Hazem Elmeleegy, Ahmed K. Elmagarmid, Emmanuel Cecchet, Walid G. Aref, and Willy Zwaenepoel. Online piece-wise linear approximation of numerical streams with precision guarantees. *Proc. VLDB Endow.*, 2(1):145–156, 2009.

[9] EPM. https://doi.org/10.24432/C5NP5K, 2024.

[10] Experiment. https://github.com/thssdb/encoding-outlier, 2024.

[11] Solomon W. Golomb. Run-length encodings (corresp.). *IEEE Trans. Inf. Theory*, 12(3):399–401, 1966.

[12] Jinmoo Heo, Yongchul Jung, Seongjoo Lee, and Yunho Jung. FPGA implementation of an efficient FFT processor for FMCW radar signal processing. *Sensors*, 21(19):6443, 2021.

[13] C. A. R. Hoare. Algorithm 65: find. *Commun. ACM*, 4(7):321–322, 1961.

[14] C. A. R. Hoare. Quicksort. *Comput. J.*, 5(1):10–15, 1962.

[15] Apache IoTDB. https://github.com/apache/iotdb/tree/research/encoding-outlier, 2024.

[16] Iosif Lazaridis and Sharad Mehrotra. Capturing sensor-generated time series with quality guarantees. In Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors, *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 429–440. IEEE Computer Society, 2003.

[17] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *Softw. Pract. Exp.*, 45(1):1–29, 2015.

[18] Ruiyuan Li, Zheng Li, Yi Wu, Chao Chen, and Yu Zheng. Elf: Erasing-based lossless floating-point compression. *Proc. VLDB Endow.*, 16(7):1763–1776, 2023.

[19] Yinan Li and Jignesh M. Patel. Bitweaving: fast scans for main memory data processing. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 289–300. ACM, 2013.

[20] Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. Chimp: Efficient lossless floating point compression for time series databases. *Proc. VLDB Endow.*, 15(11):3058–3070, 2022.

[21] Chunwei Liu, Hao Jiang, John Paparrizos, and Aaron J. Elmore. Decomposed bounded floats for fast compression and queries. *Proc. VLDB Endow.*, 14(11):2586–2598, 2021.

[22] Magnetic. https://doi.org/10.24432/C5DW43, 2024.

[23] Igor Pavlov. Lzma sdk (software development kit). https://www.7-zip.org/sdk.html, 2008. Available online.

[24] Igor Pavlov. https://www.7-zip.org/, 2024.

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

[26] Stocks. https://www.kaggle.com/datasets/tadakasuryateja/nifty-50-stocks, 2024.

[27] Apache TsFile. https://github.com/apache/tsfile/tree/research/encoding-outlier, 2024.

[28] UCI. https://archive.ics.uci.edu, 2024.

[29] Chen Wang, Jialin Qiao, Xiangdong Huang, Shaoxu Song, Haonan Hou, Tian Jiang, Lei Rui, Jianmin Wang, and Jiaguang Sun. Apache iotdb: A time series database for iot applications. *Proc. ACM Manag. Data*, 1(2):195:1–195:27, 2023.

[30] Haoyu Wang and Shaoxu Song. Frequency domain data encoding in apache iotdb. *Proc. VLDB Endow.*, 16(2):282–290, 2022.

[31] Tianrui Xia, Jinzhao Xiao, Yuxiang Huang, Changyu Hu, Shaoxu Song, Xiangdong Huang, and Jian-min Wang. Time series data encoding in apache iotdb: comparative analysis and recommendation. *VLDB J.*, 33(3):727–752, 2024.

[32] Jinzhao Xiao, Wendi He, Shaoxu Song, Xiangdong Huang, Chen Wang, and Jianmin Wang. REGER: reordering time series data for regression encoding. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*, pages 1242–1254. IEEE, 2024.

[33] Jinzhao Xiao, Yuxiang Huang, Changyu Hu, Shaoxu Song, Xiangdong Huang, and Jianmin Wang. Time series data encoding for efficient storage: A comparative analysis in apache iotdb. *Proc. VLDB Endow.*, 15(10):2148–2160, 2022.

[34] Hao Yan, Shuai Ding, and Torsten Suel. Inverted index compression and query processing with optimized document ordering. In Juan Quemada, Gonzalo León, Yoëlle S. Maarek, and Wolfgang Nejdl, editors, *Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009*, pages 401–410. ACM, 2009.

[35] Xin Zhao, Jialin Qiao, Xiangdong Huang, Chen Wang, Shaoxu Song, and Jianmin Wang. Apache tsfile: An iot-native time series file format. *Proc. VLDB Endow.*, 17(12):4064–4076, 2024.

[36] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977.

[37] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. Super-scalar RAM-CPU cache compression. In Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang, editors, *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 59. IEEE Computer Society, 2006.