Time Series Data Encoding for Efficient Storage: A Comparative Analysis in Apache IoTDB

Jinzhao Xiao
BNRist, Tsinghua University
xiaojc17@mails.tsinghua.edu.cn

Yuxiang Huang
BNRist, Tsinghua University
huang-yx21@mails.tsinghua.edu.cn

Changyu Hu
BNRist, Tsinghua University
hucy19@mails.tsinghua.edu.cn

Shaoxu Song
BNRist, Tsinghua University
sxsong@tsinghua.edu.cn

Xiangdong Huang
BNRist, Tsinghua University
huangxdong@tsinghua.edu.cn

Jianmin Wang
BNRist, Tsinghua University
jimwang@tsinghua.edu.cn

ABSTRACT

Not only the vast applications but also the distinct features of time series data stimulate the booming growth of time series database management systems, such as Apache IoTDB, InfluxDB, OpenTSDB and so on. Almost all these systems employ columnar storage, with effective encoding of time series data. Given the distinct features of various time series data, it is not surprising that different encoding strategies may perform variously. In this study, we first summarize the features of time series data that may affect encoding performance, including scale, delta, repeat and increase. Then, we introduce the storage scheme of a typical time series database, Apache IoTDB, prescribing the limits to implementing encoding algorithms in the system. A qualitative analysis of encoding effectiveness regarding to various data features is then presented for the studied algorithms. To this end, we develop a benchmark for evaluating encoding algorithms, including a data generator regarding the aforesaid data features and several real-world datasets from our industrial partners. Finally, we present an extensive experimental evaluation using the benchmark. Remarkably, a quantitative analysis of encoding effectiveness regarding to various data features is conducted in Apache IoTDB.

PVLDB Reference Format:
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 . PVLDB, 15(10): 2148 - 2160, 2022.
doi:10.14778/3547305.3547319

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/xjz17/iotdb/tree/TSEncoding.

1 INTRODUCTION

The time ordered values, write intensive workloads and other special features make the management of time series data distinct from relational databases [23, 28], and thus lead to the development of time series database management systems, open source or commercial, such as Apache IoTDB [1], InfluxDB [2], OpenTSDB [3], Prometheus [4] and so on. It is not surprising that almost all these systems employ columnar storage, given time series naturally organized by two columns, time and value. In particular, the column-oriented scheme enables effective encoding and compression of time series data. Obviously, distinct features of various data as illustrated in Figure 1 lead to different encoding performances.

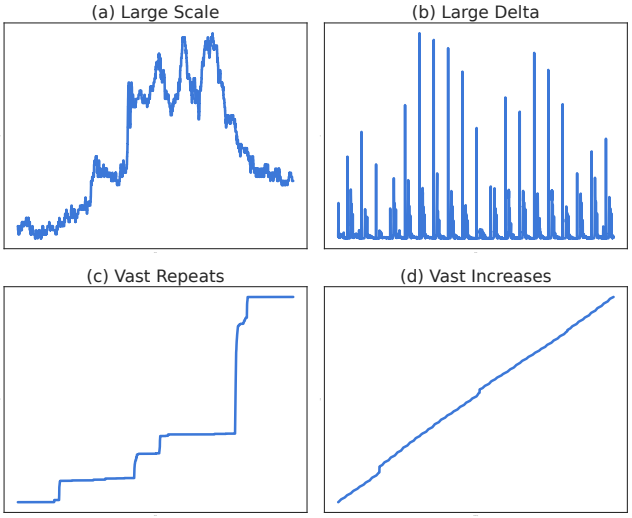


Figure 1: Example of real data with distinct features on (a) large scale, (b) large delta, (c) vast repeats and (d) vast increases, affecting the encoding performance

While general purpose data compression methods can be directly applied, e.g., SNAPPY [38] and LZ4 [19], the encoding techniques are often specialized for time series, under some intuitions like values usually not changing significantly over time, i.e., small delta. Though lossy approaches like expressing time series in piecewise polynomial [25] are highly efficient in reducing space and useful in edge or end devices, as a database, industrial customers expect a complete archive of all the digital asset, i.e., lossless. Moreover, the extremely intensive write workloads, often machine generated in IoT scenarios, prevent the time consuming approaches such as machine learning based reinforcement learning [47]. In this sense, the scope of this study is within lossless encoding with efficient system implementation.

Proceedings of the VLDB Endowment, Vol. 15, No. 10 ISSN 2150-8097.
doi:10.14778/3547305.3547319

In this paper, we present a comparative analysis of time series data encoding techniques in Apache IoTDB, an open-source time series database developed in our preliminary studies [43]. Our major contributions are summarized as follows.

(1) We summarize several time series data features that may affect the performance of encoding in Section 2. Intuitively, as illustrated in Figure 1, the scale of values is obviously an important factor of storage. Likewise, when storing the delta between two consecutive values, it becomes a key issue. The number of value repeats and increases are also essential to some encoding ideas.

(2) We present a qualitative analysis of encoding effectiveness regarding to various data features in Section 4.

While there is no winner in all the data features, TS_2DIFF performs well in a number of cases. For the cases where TS_2DIFF shows worse results, such as repeat rate 1 in Figure 17, it may be less frequent and not that significant in practice.

(3) We devise a benchmark for time series data encoding. It consists of (a) data generators for simulating various data features, (b) several real-world datasets, public or collected by our industrial partners, (c) metrics such as compression ratio (space cost after encoding and compressing divided by original space cost). In particular, multiple features could vary at the same time in the generator, such as large values but small deltas, so that the distinct cases favored by different algorithms could be illustrated. Moreover, different data types are supported, including INT32, INT64, FLOAT and DOUBLE of numerical values, as well as text values.

(4) We conduct an extensive experimental evaluation in Section 7. The quantitative analysis generally verifies the aforesaid qualitative analysis of encoding performance regarding to various data features.

Finally, we also discuss some related work in Section 8, and outline some future directions in Section 9 referring to the analysis. The source code of encoding algorithms has been deployed in the GitHub repository of Apache IoTDB [5]. The experiment related code and data are available in [6].

2 NUMERICAL DATA FEATURES

We select several features that may affect the performance of time series encoding. As illustrated in Section 4, we have three types of lossless encoding algorithms, RLE-based, Diff-based and hybrid. For all the algorithms, scale is an important feature, since techniques like bit-packing are widely used. Diff-based algorithms favor small changes in consecutive values, and thus the delta feature is considered. RLE-based algorithms handle repeating contents in consecutive values, i.e., the repeat feature. Finally, we notice that the signal bit of negative delta also affects the performance of bit compression, leading to the increase feature of consecutive values.

In addition to the aforesaid four features, there do exist others for consideration. For instance, signal-to-noise ratio (SNR) could be considered in frequency-domain-based compression [46], which however is lossy and out the scope of this study. Moreover, besides the numeral values, we further introduce two other features on value and character for the text data type in Section 3.

Table 1 outlines the major features. For simplicity, we use $TS = [v_1, v_2, \ldots, v_n]$ to denote the value list of time series. Most of these features can be directly calculated in Apache IoTDB by the data profiling tools developed in our previous study [7].

Table 1: Numerical data features

Category	Notation	Feature
Scale	Mean(TS)	Mean of values
Scale	Var(TS)	Variance of values
Scale	Spread(TS)	Maximum minus minimum of values
Delta	Mean(DS)	Mean of deltas
Delta	Var(DS)	Variance of deltas
Delta	Spread(DS)	Maximum minus minimum of deltas
Repeat	Count(RS)	Count of consecutive repeats
Increase	Count(IS)	Count of increases

2.1 Scale

The scale of data is one of the most important factors in storage. In general, the larger the values are, the more bits we need to encode them. As illustrated in Section 4 below, the run-length based algorithms [27] need to store the header, where more bits are needed for larger values. Bit-packing algorithms [35] are similarly affected. Besides, when most values are negative, bit-packing based algorithms performs bad since sign bits are 1. To this end, we employ the mean, variance and spread (maximum minus minimum) of the values in time series TS, denoted by Mean(TS), Var(TS), and Spread(TS) to represent the scale features.

2.2 Delta

The delta features show the amplitude of data fluctuations, particularly important to time series. Let $DS = [v_2 - v_1, v_3 - v_2, \ldots, v_n - v_{n-1}]$ denote the delta series of the time series DS, measuring the deltas of time-adjacent values. The differential-based algorithms [40] as introduced in Section 4.1 store these deltas. In this sense, we use Mean(DS), Var(DS), and Spread(DS), mean, variance, and spread (maximum minus minimum) of deltas, to evaluate how large the deltas could be. It is worth noting that to some extent, Var(TS) also reflects the delta features, and likewise, Var(DS) understands delta of deltas, important to some encoding such as TS_2DIFF discussed in Section 4.1.

2.3 Repeat

Repetitive values are widely observed in time series, such as unchanged temperature reading in several minutes. Such consecutive repeats can be compressed by run-length based algorithms [27] in Section 4.2. They also output zero values in XOR operators introduced in Section 4.1, shrinking the space efficiently. To this end, we introduce a method to describe the repeats of time series. The main idea is to count the number of consecutively repeated values which are in the interval of consecutive repetitive values. We define RS the repeat count series of time series $TS = [r_1, r_2, \ldots, r_n]$, having

$$r_i = \begin{cases} r_{i-1} + 1, & \text{if } v_i = v_{i-1}, \\ 1, & \text{otherwise } v_i \neq v_{i-1} \end{cases} \quad (1)$$

for $1 < i \leq n$ and $r_1 = 1$. Algorithms like SPRINTZ [20] in Section 4.3 have a block size of 8 for bit-packing numbers into integer bytes.

Table 2: Text data features

Category	Notation	Feature
Value	Exponent(TS)	Exponent of Zipfian distribution
Value	Domain(TS)	Domain size of text values
Character	Length(TD)	Length of text value
Character	Repeat(TD)	# consecutive character repeats

Therefore, we are interested in the values that repeat more than 8 times. The repeat count measure Count(RS) is thus

$$\text{Count}(RS) = |\{r_i \mid r_i \geq 8, 8 \leq i \leq n\}|.$$

2.4 Increase

While repetitive values have difference 0, the sign of difference for non-repetitive values is also concerned. The reason is that the non-zero sign bits may interfere encoding in algorithms like RLBE [41] introduced in Section 4.3. If all the difference signs are positive, in other words, the time series values are always increasing, the encoding performs better. In contrast, when the differential value is negative, i.e., decreasing, the encoding performance would be bad. In this sense, we define Count(IS) the number of increasing values with adjacent timestamps,

$$\text{Count}(IS) = |\{v_i \mid v_i > v_{i-1}, 1 < i \leq n\}|. \quad (2)$$

In addition to the features on scale, delta, repeat and increase, data type is also an important factor that affects the encoding performance. For INT32 and INT64, similar values have smaller deltas than those of FLOAT and DOUBLE. Moreover, the longer INT64 and DOUBLE may have more 0 bits, where bit compacting strategies may perform. Therefore, in the qualitative analysis in Table 4 and the quantitative evaluation such as Figure 9, the data types are also considered as important data features.

3 TEXT DATA FEATURES

Similarly, text time series data has several data features which may be related to encoding performance, including the distribution of values, the domain of values, the average length of text value and consecutive repeats of characters. Table 2 outlines the major text features.

3.1 Value

The text values often follow a Zipfian distribution [18, 45] in practice. The exponent of Zipfian distribution represents the frequency of values. The larger the exponent is, the larger the skewness of value frequency is. Such skewness affects the performance of HUFFMAN encoding [29, 36], which relates to value frequency. Moreover, the domain size of text values is also important, e.g., to DICTIONARY encoding [44] that stores the value domain as dictionary.

3.2 Character

The character features could affect the encoding algorithms that encode data at character level. The length of values is of course a key factor. Longer values generally lead to larger character encoding results. Again, repeats of characters also affect the encoding performance, such as RLE [27] and HUFFMAN [29, 36].

Table 3: Properties of numerical data encoding algorithms

Encoding	First value	# Repeat	RLE-based	Diff-based
TS_2DIFF	✓			✓
GORILLA	✓			✓
RAKE			✓	
RLE		✓	✓	
RLBE	✓	✓	✓	✓
SPRINTZ	✓		✓	✓

Figure 2: Examples of TS_2DIFF encoding algorithm

4 NUMERICAL DATA ENCODING

Referring to the aforesaid discussions on lossless requirement and system architecture, we introduce six encoding algorithms that are proper to implement in Apache IoTDB, including TS_2DIFF [8], GORILLA [37], SPRINTZ [20], RLE [27], RLBE [41], and RAKE [21]. The source code of implementation is available in the GitHub repository of Apache IoTDB [5]. Table 3 lists the common ideas that may share among different encoding algorithms. Specifically, a qualitative analysis of encoding effectiveness regarding to various data features is presented in Table 4.

4.1 Differential-based Encoding

Differential encoding proposes to reduce the absolute value when the data in time series is continuous, especially when the original data is large. The number of significant bits reduces since the absolute value is decreased, which reduces storage costs. Thereby, compression ratio has an important relationship with delta features in differential encoding algorithms. While traditional differential encoding can only perform well in monotonous integer values, GORILLA and TS_2DIFF two recent advances.

4.1.1 TS_2DIFF. The TS_2DIFF encoding is a variant of delta-of-delta [37]. It consists of three steps: delta encoding, second delta encoding, bit-packing. The first step calculates the delta of every value by subtracting the current value from the previous one. Note that the first value does not have the previous value and should be stored directly. Then, the algorithm finds the minimum delta, $\min_{diff}$, and gets the final data to store by subtracting the delta from $\min_{diff}$. Finally, the leading zeros of fixed length of binary data is removed to get the final encoded byte stream.

Table 4: A qualitative analysis of encoding effectiveness regarding to various numerical data features

Encoding	INT32	INT64	FLOAT	DOUBLE	Large value mean	Large value variance	Large delta mean	Large delta variance	Vast repeats	Vast increases
TS_2DIFF	✓	✓	✓	✓	◯	×	✓	×	◯	◯
GORILLA	✓	✓	◯	◯	◯	×	◯	◯	◯	◯
RAKE	×	✓	×	✓	×	◯	◯	◯	◯	◯
RLE	✓	✓	◯	◯	×	◯	◯	◯	✓	◯
RLBE	✓	✓	◯	◯	◯	◯	×	◯	✓	✓
SPRINTZ	✓	✓	◯	◯	◯	×	×	×	✓	◯

✓good performance, ◯ no preference, × bad performance

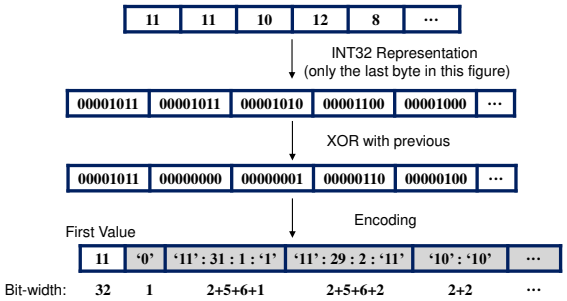


Figure 3: Examples of GORILLA encoding algorithm

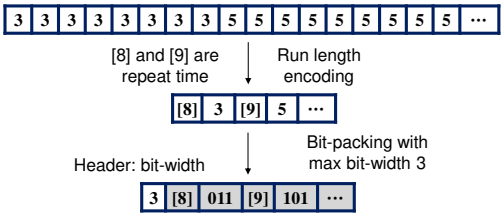


Figure 4: Examples of RLE encoding algorithm

Therefore, the smaller variance and delta variance the sequence has, the smaller bit-width the difference of the sequence has, and the smaller the final compression ratio is. As summarized in Table 4, TS_2DIFF is also suitable for large delta mean values, since the values can get a small value to store by subtracting large minimum in the second delta encoding process.

Figure 2 shows a case of small delta variance, i.e., all the deltas in the second delta are small and thus have lower space cost.

4.1.2 GORILLA. The GORILLA encoding is originally designed for Facebook's time series database (TSDB) [37]. First, it processes the timestamps with second order differential, which is effective when the values come in an almost fixed interval. The values are divided into four areas by significant bit width. Then it writes the packed timestamps and values. As for the value it uses the XOR coding method. Typically, this procedure results in many leading and tailing zeros for float number. If the XOR result is zero, it only writes a bit '0' to represent it. Otherwise, it writes the different bits and numbers of leading/tailing zeros of the result.

As shown in Table 4, GORILLA is suitable for small variance data, as it increases the number of leading and tailing zeros in XOR results. On the other hand, it may fail on the time series with drastic change, as more non-zero bits are used to encode the values.

In Figure 3, GORILLA shows a good performance by compressing 160 bits of 5 INT32 values into 66 bits.

The time series data has a small variance and lots of leading and tailing zeros.

4.2 Run-length-based Encoding

Run-length encoding (RLE) [27] targets on reducing the space cost of adjacent repeating values. The compression effect of traditional RLE algorithm is limited when times of repeating values are small. RLE with bit-packing and RAKE [21] are more effective advances.

4.2.1 RLE with bit-packing. RLE encoding [27] stores the continuous repeating time of one element instead of repeating the same elements over and over. For example, a series 444556666 can be stored as 435264 with run-length, where the number 3 after 4 denoting that 4 repeats 3 times. RLE introduces extra space cost to store repeat times when values are consecutive, thus IoTDB implementation combines RLE with bit-packing. Run-length is only applied to values whose repeat time is larger than 8. Simple bit-packing is implemented to others.

It is not surprising that RLE with bit-packing performs good when time series has vast repeats, as more values can be encoded into one value and its repeat time. Bit-packing reduces storage cost caused by data with few repeats. The algorithms perform better when repeat rate is high and value mean is low but positive.

For example, in Figure 4, the series has many consecutive repeat values which are also smaller positive numbers. Thereby, RLE with bit-packing performs.

4.2.2 RAKE. The RAKE encoding [21] is based only on bits counting operations. It considers a T-teeth rake to process T bits every time. If all the T bits are zeros, a setting bit of 0 is stored. Otherwise, it first stores a setting bit of 1. Then, a codeword of $L = \lceil \log_2 T \rceil$ bits is generated according to the numbers in the rake. The codeword records the position of the first 1, p_{first}, in binary notation. And, the rake shifts $p_{first} + 1$ bits to the right.

Therefore, we expect the '1's of binary numbers to be more sparsely, so that T zeros can be compressed into one '0'. For INT64

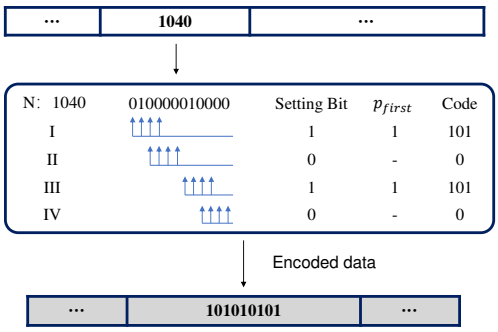


Figure 5: Examples of RAKE encoding algorithm

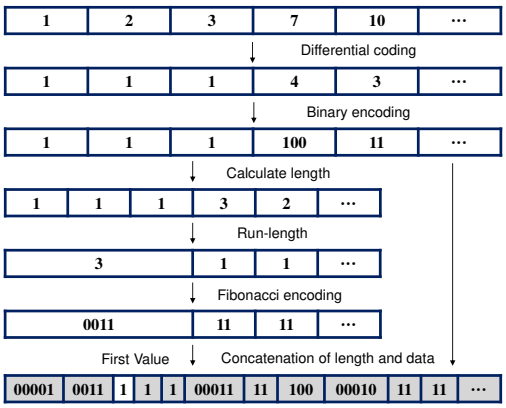


Figure 6: Examples of RLBE encoding algorithm

data, data has more leading zeros, and will be compressed more efficiently than INT32 data, as summarized in Table 4.

Figure 5 shows a simple example of how values are encoded by RAKE algorithm. Since the first 20 bits will be obviously encoded to five zeros, the process of compressing the first 20 bits is not shown. For the number of Figure 5, a sparse number, $N = [010000010000]$ is compressed by the RAKE algorithm (with $T = 4$) to produce a compressed sequence of 8 bits, [10101010].

4.3 Hybrid Encoding

While the differential-based and run-length-based encoding algorithms can perform well in different scenarios, there are certain cases with both small delta features and vast repeats. To this end, hybrid encoding with both ideas can achieve a better result, such as RLBE [41] and SPRINTZ [20].

4.3.1 RLBE. The RLBE encoding [41] proposes to combine delta, run-length and Fibonacci based encoding ideas. It has five steps: differential coding, binary encoding, run-length, Fibonacci coding [42] and concatenation. Specifically, delta encoding is first applied to original data (integers of 32bits), and lengths of each differential value (in binary notation) are calculated. Run-length is then applied to the length codes. In the concatenation phase, the first 5 bits represent the length of binary words (the length is encoded in binary word), followed by the Fibonacci code words of repeat time

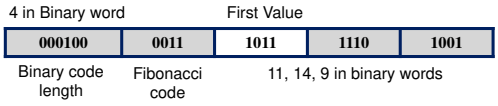


Figure 7: Examples of INT32 and INT64 extensions for RLBE

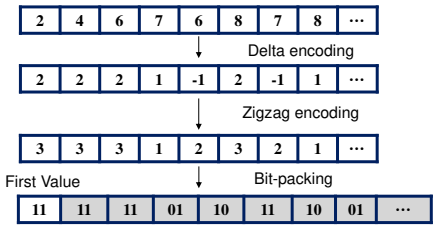


Figure 8: Examples of SPRINTZ encoding algorithm

of length code, and sequentially followed by binary code words of differential values with the same length.

As summarized in Table 4, when the differential value is positive and small, RLBE performs good. RLBE performs bad when the differential value is negative, as the sign bit is '1' and no leading '0's can be abolished. When adjacent differential values are of different order of magnitude, i.e., variance is large, RLBE also performs bad as run-length on length code cannot be applied.

The examples are shown in Figure 6. While values are all increasing in the time series data and all the deltas are positive, RLBE has a good performance such as the example in Figure 6.

To enlarge the encoding range, we extend the first 5 bits representing binary code length to 6 bits, as shown in Figure 7. The reason is that when differential value is negative, it has 32 meaningful bits, exceeding the representation range of 5 bits. Likewise, when supporting integers of 64 bits, we expand length representing binary code to 7 bits for the same reason.

4.3.2 SPRINTZ. The SPRINTZ encoding [20] combines encodings in four steps: predicting, bit-packing, run-length encoding and entropy encoding. In the first step, it uses some predictive functions (delta encoding or Fast Integer REgression encoding) to estimate the next coming value. Then it encodes the difference of the actual value and the predicted one. Typically, this step shrinks the absolute value to be encoded. Next, it bit-packs a block of residuals obtained in the first step. The largest number of significant bits in the block is written in the header, and the leading zeros are trimmed. Following that, run-length encoding and entropy encoding (e.g., Huffman coding) are applied to reduce redundancy. Run-length coding compresses the consecutive zero blocks by recording the number of zeros and entropy coding compresses the headers and payloads by encoding bytes in the form of Huffman coding.

As summarized in Table 4, SPRINTZ algorithm is suitable for predictable time series. For delta function, the vast repeats or linearly increasing time series is the best target. For the FIRE (Fast Integer REgression) predictor, a constant slope is the best fit.

Since the example in Figure 8 has small value variance and delta mean, the SPRINTZ encoding has a good performance.

Table 5: A qualitative analysis of encoding effectiveness regarding to various text data features

Encoding	Large exponent	Large domain	Large length	Vast repeats
HUFFMAN	✓	×	×	✓
DICTIONARY	◯	×	✓	◯
RLE	◯	◯	✓	✓

✓good performance, ◯ no preference, × bad performance

Table 6: Parameters of numerical data generator for various data features

Notation	Data Features	Range
μ_v	Mean of values	$[-5 \times 10^4, 5 \times 10^4]$
μ_d	Mean of deltas	$[-2000, 2000]$
σ_d	Variance of deltas	$[0, 1000]$
γ	Repeat rate	$[0, 1]$
η	Increase rate	$[0, 1]$

5 TEXT DATA ENCODING

In this section, we introduce 3 text encoding algorithms, including DICTIONARY [44], HUFFMAN [29, 36] and RLE [27]. Table 5 presents a qualitative analysis of encoding effectiveness regarding to various text data features.

5.1 DICTIONARY Encoding

The DICTIONARY algorithm [44] finds value in the dictionary. If the value is found successfully, the value is replaced by a key in the dictionary; otherwise, the algorithm adds a new pair of key and value in the dictionary. For example, if the map in the dictionary is {1:True, 2:False}, the time series TS = {True, False, True, True} could be encoded as 1211. Obviously, a large domain leads to higher cost in DICTIONARY encoding. In contrast, DICTIONARY favors large length values, by encoding it to a short key.

5.2 Run Length Encoding

Run-Length Encoding (RLE) [27] performs especially for data with strings of repeated characters (the length of the string is called a run). The main idea of the algorithm is to encode repeated characters as a pair of the length of the repeated characters and the character. For example, the value 'abbaaaaabaabbbaa' of length 16 bytes is represented as '1a2b5a1b2a3b2a'. However, if there are no repeat characters in the value, the size of output data can be twice as large as the size of input data.

5.3 HUFFMAN Encoding

The HUFFMAN encoding algorithm [29, 36] decreases the overall length of the data by assigning shorter codewords to the more frequently occurring characters, employing a strategy of replacing fixed length codes (such as ASCII) by variable length codes. It creates a uniquely decipherable prefix-code precluding the need for creation of a separator to determine codeword boundaries. For data with many high frequency values in skewed data distribution and many repeated characters, HUFFMAN's ability to shorten high frequency character encoding performs.

6 ENCODING BENCHMARK

To evaluate the encoding algorithms, we extend IoTDB-Benchmark [9], developed in our previous study, by introducing advanced data generator for various time series data features. Some real-world data collected by our industrial partners are also employed, together with necessary metrics for evaluation.

6.1 Synthetic Numerical Data

To evaluate the effect of encoding algorithms working on different data features, we design a data generator for varying data features, controlled by 5 parameters. Table 6 lists the parameters, generally analogous to the data features in Section 2. The parameter μ_v controls the mean of values $\mathsf{Mean}(TS)$ in Table 1. For each data point, we employ a normal distribution with μ_d and σ_d to determine its delta to the previous value, i.e., analogous to mean of deltas $\mathsf{Mean}(DS)$ and variance of deltas $\mathsf{Var}(DS)$ in Table 1. Since the point has been already determined by this delta and its previous value, we are not able to further control the variance or spread of values. Nevertheless, they are related to the deltas as discussed to certain extent. Repeat rate γ is the probability of generating a series of consecutive points with repeated values, analogous to repeat count $\mathsf{Count}(RS)$ defined in Section 2.3. Increase rate η denotes the probability of generating a point with value greater than the previous, for the increase count feature $\mathsf{Count}(IS)$ in Section 2.4.

Algorithm 1 presents the pseudo-code of the data generator. Let DS denote the delta series, as introduced in Section 2.2, to generate. Lines 3-6 generate repeats with probability γ specified in the parameters. Likewise, Lines 8-12 generate an increase point with probability η. The delta is given by a normal distribution with parameters μ_d and σ_d. Finally, the delta series DS is transformed to TS by a prefix summation, and zoom all the values to the target value mean μ_v.

6.2 Real-world Numerical Data

Real-world datasets, public or collected by our industrial partners, are also included in the benchmark, for learning the data features, and evaluating the encoding algorithms. Table 7 reports the major statistics of the prepared datasets. Figure 12(b) below illustrates their data features.

MSRC-12 [10] is a dataset with float values from Microsoft Kinect gestures and has a low repeat rate and small delta variance due to small fluctuation. UCI-Gas [11] also consists of float values for measuring gas concentration during chemical experiments and has a low delta mean. WC-Vehicle contains sensor readings for monitoring vehicles and has a low repeat rate. TH-Climate is a dataset of weather information collected in Tsinghua campus. It has low delta mean and high repeat rate. CW-AIOps is a dataset of application performance monitoring (APM) in cloud services, where the mean, variance and spread of both value and delta are very large

Algorithm 1: Numerical data generator

Data: $\mu_v, \mu_d, \sigma_d, \gamma, \eta$, length n
Result: TS

```
1  DS := empty_list();
2  while |DS| < n do
3      isRepeat := random_index(γ);
       /* Probability of isRepeat == 1 is γ,        */
       /* probability of isRepeat == 0 is 1 − γ     */
4      if isRepeat then
5          repeat_len := random(8, T);
           /* Get a random number in (8, T]         */
6          DS.append(0, repeat_len);
           /* Append 0 for repeat_len times         */
7      else
8          isPositive := random_index(η);
9          delta := 0;
10         if isPositive then
11             while delta ≤ 0 do
12                 delta := random_gauss(μ_d, σ_d);
13             end
14         else
15             while delta ≥ 0 do
16                 delta := random_gauss(μ_d, σ_d);
17             end
18         end
19         DS.append(delta);
20     end
21 end
22 TS := prefix_sum(DS);
23 TS.zoom(μ_v);
   /* Zoom TS to adjust means to μ_v              */
24 return TS;
```

due to the complex application scenarios. CS-Ship monitors the status of ship engines. Value mean and delta mean are small while increase is high. TY-Carriage contains readings of carriage monitoring sensors, and it has low delta mean. WH-Chemistry is a dataset from chemical plant. It has high value mean, value variance, value spread, delta mean, delta variance and delta spread. CR-Train is a dataset from metro system, with low delta mean and high repeat rate. CB-Engine consists of senor readings in concrete mixer. It has low delta mean, delta variance and repeat rate.

6.3 Synthetic Text Data

The text data generator considers 4 parameters in Table 8, corresponding to 4 features listed in Table 2 in Section 3. The exponent θ_v determines the skewness of value distribution. N_v is the domain size of text values. These two parameters decide the distribution of values. Moreover, ℓ_c represents the average length of all the text values, and repeat rate γ_c is the probability of generating consecutive character repeats.

Algorithm 2 shows the pseudo-code of text data generator. Let TS be the time series and TD be the value domain to generate TS.

Table 7: Real-world numerical datasets

Dataset	Public	# data points	# time series
MSRC-12	[10]	17,059	10
UCI-Gas	[11]	189,981	19
WC-Vehicle		79,992	8
TH-Climate		1,317,330	140
CW-AIOps		2,215,599	224
CS-Ship		89,991	9
TY-Carriage		9,680,088	450
WH-Chemistry		44,622	54
CR-Train		859,914	86
CB-Engine		533,901	88

Table 8: Parameters of text data generator for various text data features

Notation	Text Data Features	Range
θ_v	Value of exponent	[0, 10]
N_v	Domain size of text values	[1, 1500]
ℓ_c	Average length of text value	[100, 1100]
γ_c	Repeat rate	[0.9, 1]

Lines 2-16 generate TD with domain size N_v, given the value length ℓ_c and character repeat rate γ_c. Then, lines 17-23 generate the distribution of values, under a Zipfian distribution with exponent θ_v.

6.4 Real-world Text Data

Table 9 presents several real-world text time series datasets. CW-AIOps is a log dataset of application performance monitoring (APM) in cloud services, collected by our industrial partners. Web Server Access Logs, Incident Event Log Dataset and Web Log Dataset are public datasets in Kaggle [15]. Among them, Web Server Access Logs contain information on any event that was registered / logged. Incident Event Log Dataset is a event log dataset of a website. Web Log Dataset is the server log dataset of RUET OJ.

6.5 Evaluation Metrics

To measure the performance of encode algorithms for time series, two aspects are considered, compression ratio in space cost, encoding and decoding time cost.

6.5.1 Compression Ratio. It measures the ratio of compressed (encoded) data size to uncompressed (non-encoded) data size

$$compressionRatio = \frac{compressedSize}{uncompressedSize}.$$

6.5.2 Time Cost. We report two aspects of time cost. The *insert time* measures the total cost of inserting a time series, including adding to memTable, flushing from memory to disk with sorting, encoding, and compressing. The *select time* measures the total cost of querying a time series, with decompressing and decoding.

Algorithm 2: Text data generator

Data: $\theta_v, N_v, \ell_c, \gamma_c$, length n
Result: TS

TS := empty_list()
TD := new string[N]
for $i \in [0, N_v]$ **do**
  for $j \in [0, \ell_c]$ **do**
    if $j = 0$ **then**
      TD_i.append(rand_Char())
      continue
    end
    $isRepeat$:= random_index(γ_c)
    /* Probability of isRepeat == 1 is γ_c, probability of isRepeat == 0 is $1-\gamma_c$ */
    if $isRepeat$ **then**
      TD_i.append(TD_i.back())
    else
      TD_i.append(rand_Char_except(TD_i.back()))
    end
  end
end
$num := \{num_0, ..., num_{N_v-1}\}$, where $num_i = \frac{(\frac{1}{i+1})^{\theta_v}}{\sum_{j=0}^{N_v-1}(\frac{1}{j+1})^{\theta_v}} n$
for $i \in [0, N_v]$ **do**
  for $j \in [0, num_i]$ **do**
    TS.append(TD_i)
  end
end
TS = random_permutation(TS) return TS;

Table 9: Real-world text datasets

Dataset	Public	# points	# series
CW-AIOps		2,215,599	224
Web Server Access Logs	[12]	10,365,152	1
Incident Event Log	[13]	141,712	35
Web Log	[14]	258,441	4

7 EXPERIMENTAL EVALUATION

In this section, we conduct the experimental evaluation of encoding algorithms analyzed in Section 4. Note that encoding and compressing are complementary. Therefore, we study PLAIN (no encoding), TS_2DIFF [8], GORILLA [37], SPRINTZ [20], RLE [27], RLBE [41], and RAKE [21] encoding, combined with NONE (no compression), SNAPPY [38], GZIP [16] or LZ4 [19] compressor. We present their performances over various data types including INT32, INT64, FLOAT and DOUBLE as listed in Table 4. With the help of benchmark in Section 6, the results on both the real-world and synthetic data with various features are reported.

The experiments run on a machine with 2.1 GHz Intel(R) Xeon(R) CPU and 128GB RAM. While the source code of encoding algorithms has been deployed in the GitHub repository of Apache IoTDB [5], the experiment related code and data are available in [6].

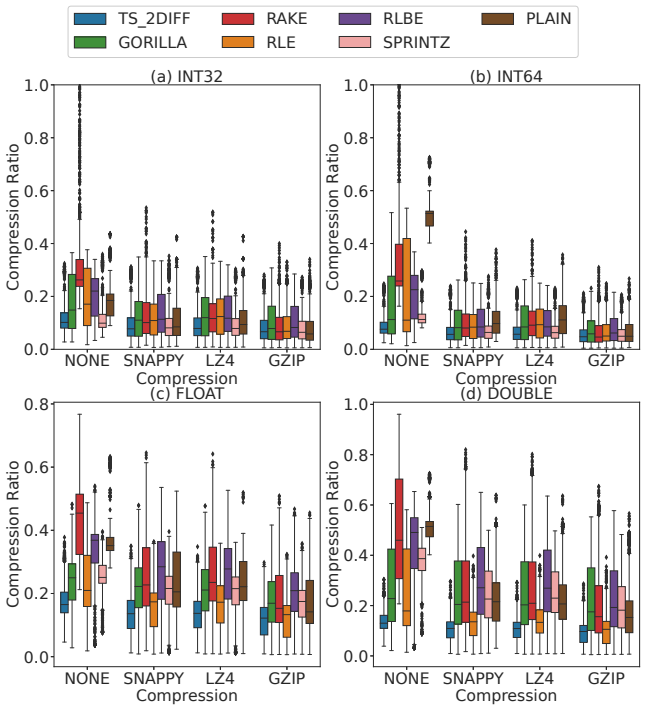


Figure 9: Compression ratio over all numerical datasets

7.1 Real-world Numerical Data Evaluation

Figures 9, 10 and 11 report the compression ratio, insert time and select time defined in Section 6.5, respectively. The experiments are conducted over all 1088 time series in Table 7. In boxplot chart, every element represents one time series. We study 28 combinations of 7 encoding schemes and 4 compression schemes on 4 different data types. In Figure 9, the lower the metrics are, the better the performance is. Since the compression ratios are the same in different runs, we do not repeat the experiments of compression ratio. The experiments on time cost are repeated 50 times.

7.1.1 Comparison. As illustrated in Figure 9, TS_2DIFF encoding achieves good (low) compression ratio, with or without compression. RAKE encoding performs even worse than PLAIN (no encoding) when handling INT32 and FLOAT.

As introduced in Section 4.2.2, when there are many consecutive 0 bits, RAKE performs. The more the 1 bits are, the worse the RAKE algorithm performs. For INT32 and FLOAT, since there are less 0 bits than INT64 and DOUBLE for the same values, the performance of RAKE is worse. In particular, for negative numbers with a small absolute value, owing to the leading sign bit '1' and more leading '1's, RAKE may perform even worse given the extra cost of setting bits and so on. Similar results are also observed in Figure 14(a), where RAKE again performs worse than PLAIN on negative numbers (with value mean less than 0).

GORILLA performs better on INT32 and INT64 than FLOAT and DOUBLE since positions of leading and trailing 0 are more similar. The results verify the qualitative analysis on data types in Table 4.

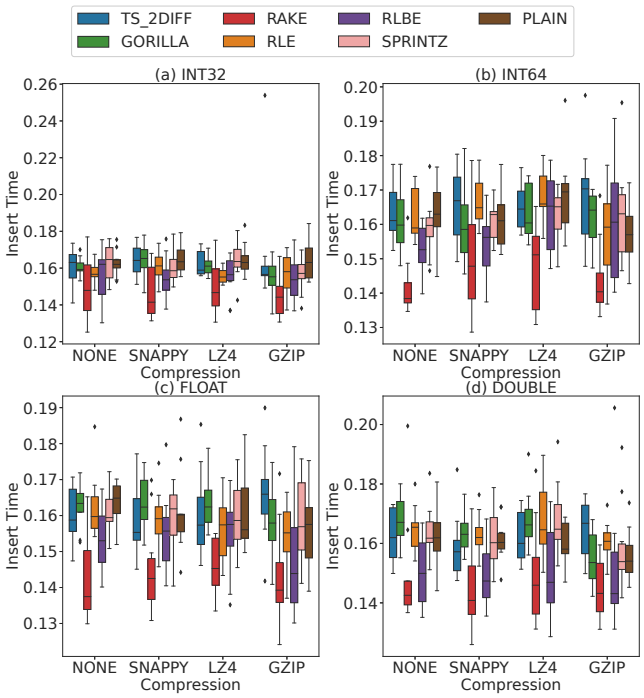


Figure 10: Insert time over all numerical datasets

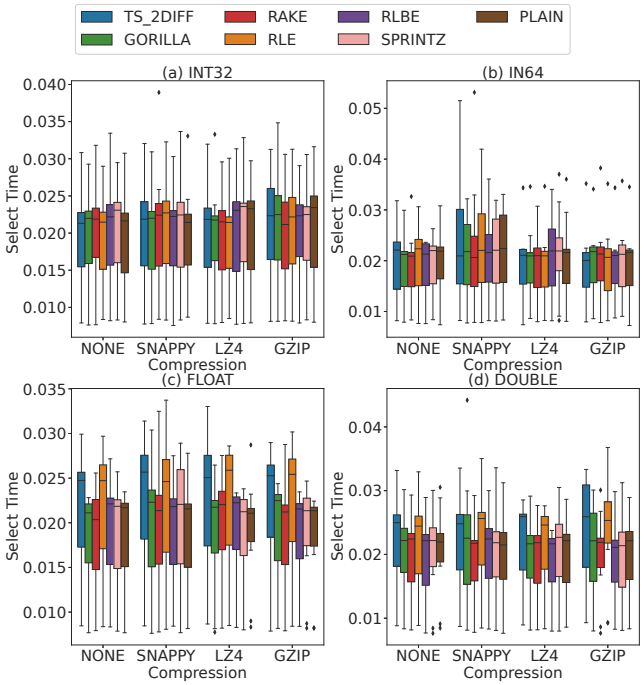
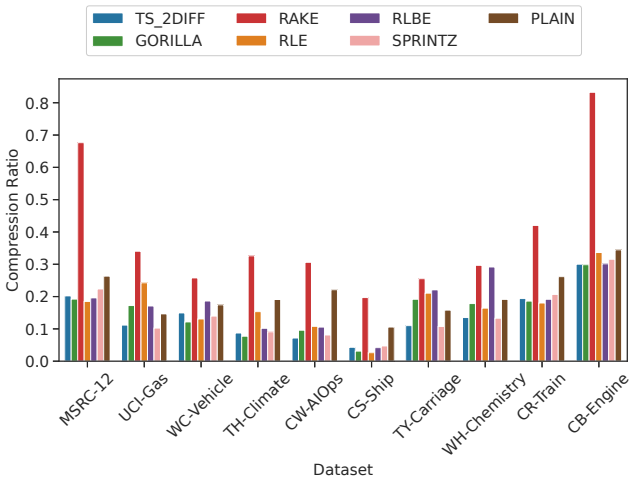
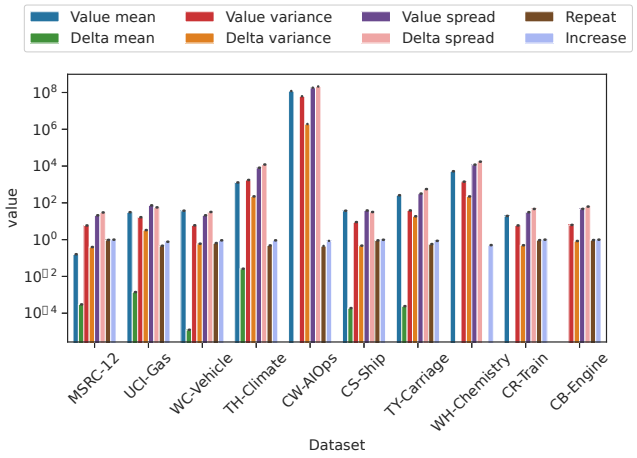


Figure 11: Select time over all numerical datasets

Nevertheless, with compression, the space cost is further reduced. The improved by compression after TS_2DIFF encoding, however, is limited, with extra compressing and decompressing time.



(a) Compression ratio of each dataset



(b) Feature value of each dataset

Figure 12: Compression ratio and features on each dataset

7.1.2 Individual Datasets. In addition to the qualitative analysis on data types, we further validate the effects of data features analyzed in Table 4. Figure 12(a) reports the compression ratio of 7 encoding schemes without compression applied (NONE). Figure 12(b) shows the corresponding 8 data features listed in Table 1.

In general, TS_2DIFF encoding still achieves good performance, while RAKE could be worse than PLAIN without encoding, similar as the observations in Figure 11. For the datasets with large delta mean, such as UCI-Gas, TH-Climate, MSRC-12, CS-Ship and TY-Carriage, TS_2DIFF still performs well. For the datasets with small value variance and delta mean, such as WC-Vehicle and CB-Engine, GORILLA performs better. RLBE performs better in CS-Ship than other datasets, since CS-Ship has relatively smaller delta mean and delta variance. The results verify again the analysis in Table 4.

Note that both time and value series are encoded and compressed, and the statistics stored in the PageHeader consider both time and value series compression. Since time is encoded and compressed

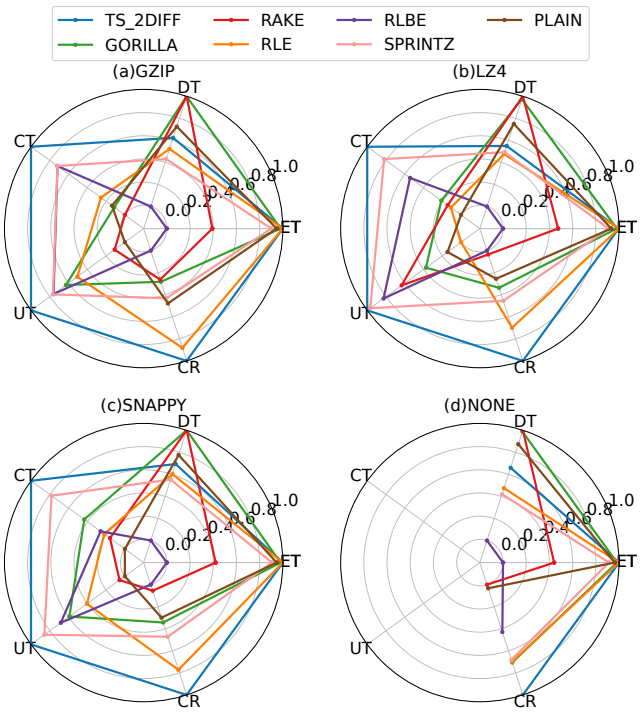


Figure 13: Trade-off between time and compression ratio

by default, the compression ratio of PLAIN encoding and NONE compression on value series together with time is less than 1.

7.1.3 Trade-off. Figure 13 breaks down the time costs into encoding time (ET), decoding time (DT), compression time (CT), uncompression time (UT) and the corresponding compression ratio (CR) of different encoding algorithms together with four compression strategies. The experiments run on all the real-world datasets in Table 7 and report the average. For each dimension, we normalize the results into a range between 0 and 1, the larger the better. For ET, DT, CT and UT, a larger value represents lower time compared with other encoding algorithms, i.e., more efficient. For compression ratio (CR), a larger metric represents lower compression ratio, again better compression performance.

As shown, most encoding algorithms are efficient in encoding (ET). TS_2DIFF has better compression ratio (CR) as well as compression time (CT) and uncompression time (UT), but the corresponding decoding time (DT) is worse. GORILLA with both better encoding and decoding time (ET and DT) has worse compression ratio (CR). Similar trade-off results are also observed in Figure 13(d) without compression (NONE).

7.2 Varying Numerical Data Features

While the experiments on real data in Section 7.1 cover only part of the data features analyzed in Table 4, in order to conduct a more extensive quantitative analysis, we further evaluate the encoding algorithm over the synthetic data with various features controlled by the data generator presented in Algorithm 1. Again, we compare 7 encoding schemes without compression (NONE). Figures 14-18 report the results on INT32. Due to the limited space, the similar results on other setting combinations are omitted.

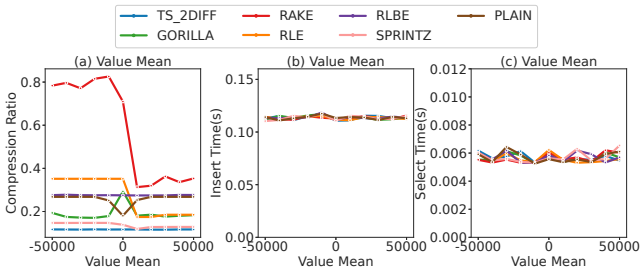


Figure 14: Varying scale feature of value mean μ_v

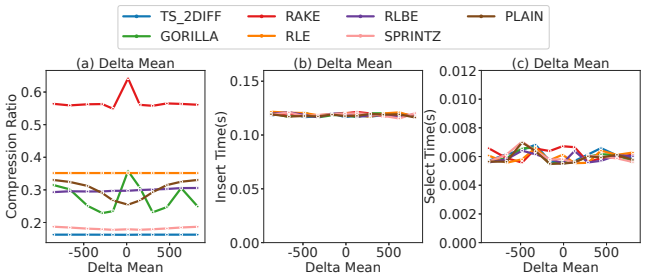


Figure 15: Varying delta feature of delta mean μ_d

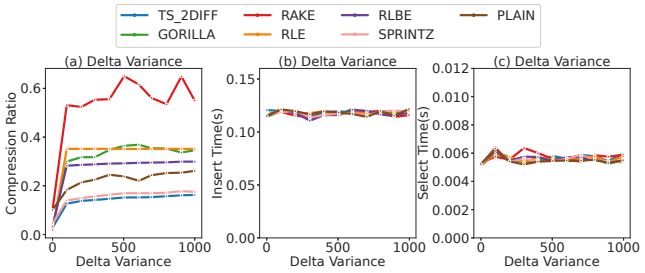


Figure 16: Varying delta feature of delta variance σ_d

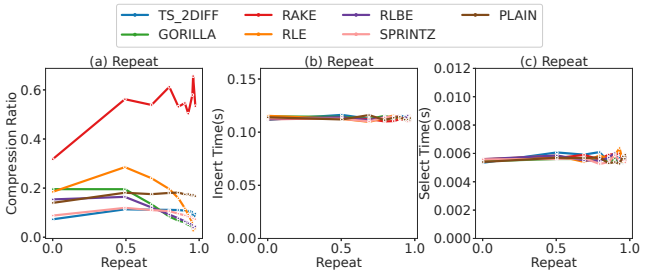


Figure 17: Varying repeat feature of repeat rate γ

7.2.1 Compression Ratio. Figure 14(a) varies the scale feature of value mean μ_v introduced in Table 6. RAKE and RLE perform better when value mean is positive. For negative value mean, sign bits are 1, i.e., compression cannot be applied on the first 4 bits of

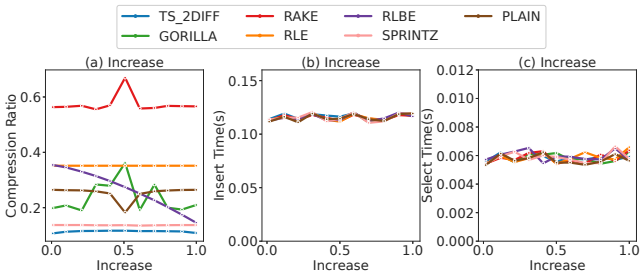


Figure 18: Varying increase feature of increase rate η

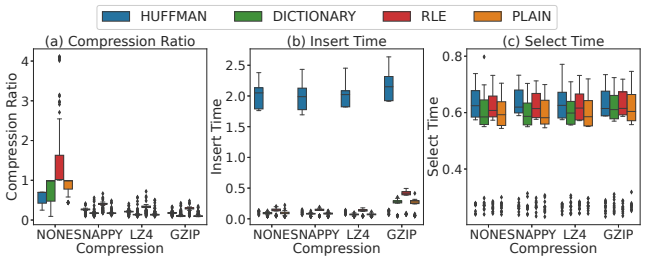


Figure 19: Performance of text encoding on real datasets

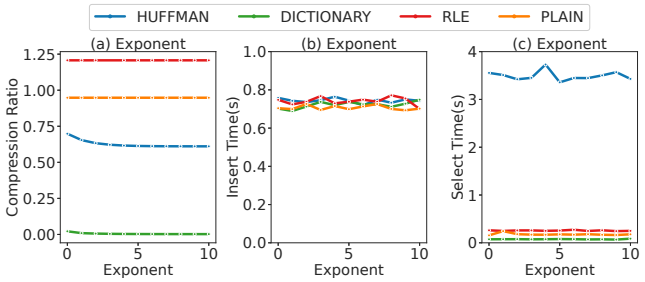


Figure 20: Varying text feature of exponent θ_v

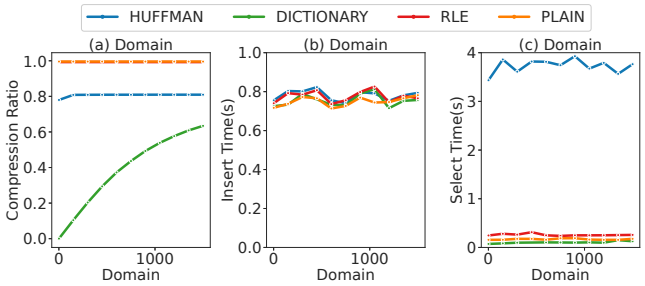


Figure 21: Varying text feature of domain size N_v

leading zeros. TS_2DIFF and RLBE are less affected by value mean, since run-length and differential encoding store the length of repeat times and differential value instead of storing lots of large values. GORILLA is unstable, as its performance is affected by XOR, unrelated to value mean.

For delta features, Figure 15(a) varies delta mean μ_d and Figure 16(a) considers delta variance σ_d. TS_2DIFF performance decreases with the increase of delta variance in Figure 16(a), which is not surprising referring to the analysis in Table 4.

When repeat rate γ increases in Figure 17(a), RLE, RLBE and SPRINTZ perform better. They are run-length based algorithms, favoring high repeats. GORILLA also performs better with repeat rates increasing, since XOR has more zeros and only a bit of '0' needs to be stored. RAKE's performance is not affected due to its run-length is on bits, instead of value repeats.

Figure 18(a) varies the increase rate η. When increase rate becomes larger, more positive values appear, beneficial to bit-pack. Thereby, RLBE is positively correlated with the increase rate.

7.2.2 Time Cost. Analogous to the compression ratio results in Section 7.2.1, Figures 14(b), 15(b), 16(b), 17(b) and 18(b), report the insert time under various value mean μ_v, delta mean μ_d, delta variance σ_d, repeat rate γ and increase rate η, respectively. Likewise, Figures 14(c)-18(c) are the corresponding select time. Due to the limited space, similar results on other setting combinations are omitted. Each test is conducted 50 times and reports the average.

As shown, while the compression ratio is affected largely by various data features in Figures 14(a)-18(a), the corresponding insert time and select time are stable. Different encoding methods indeed lead to very close insert and select time. Similar to Figures 10 and 11, the insert time is much higher than the select time under various data features. Due to the extremely low select time, the variances of select time in the tests are relatively larger. The results are generally consistent with those on real datasets in Section 7.1.

7.3 Real-world Text Data Evaluation

Figure 19 reports compression ratio, insert time and select time defined in Section 6.5, respectively, over all text datasets in Table 9. We perform the 16 combinations of 4 text encoding schemes and 4 general compression schemes on text data.

As shown in Figure 19, when there is no compression algorithm applied, HUFFMAN has the best performance in compression ratio, but it has the worst performance in time cost. RLE shows even worse compression ratio than PLAIN (no encoding), owing to the limited repeated characters as analyzed in Section 5.2. When compression is applied, either SNAPPY, LZ4 or GZIP, the DICTIONARY encoding has almost the best compression ratio and time cost.

7.4 Varying Text Data Features

Similar to the numeric data, we generate text data with different features mentioned in Section 3, by Algorithm 2, and evaluate the encoding performance on the synthetic text data.

For value features, Figure 20 varies the exponent θ_v introduced in Table 8. The larger the exponent is, i.e., the data distribution is more skewed, the better HUFFMAN performs. The improvement of compression ratio however is not significant. The other algorithms are not affected by the exponent, as also analyzed in Table 5.

Figure 21 varies the domain size N_v introduced in Table 8. It is not surprising that DICTIONARY performs worse with the increase of domain size. In contrast, DICTIONARY favors a larger value length as illustrated in Figure 22, with a slight improvement.

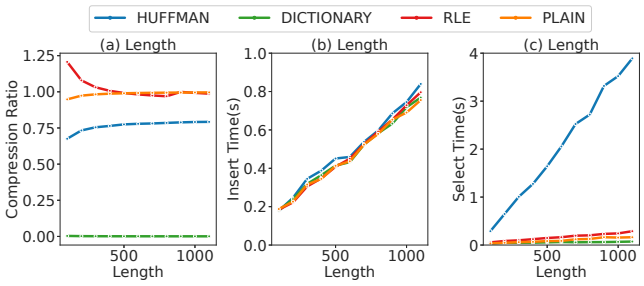


Figure 22: Varying text feature of value length ℓ_c

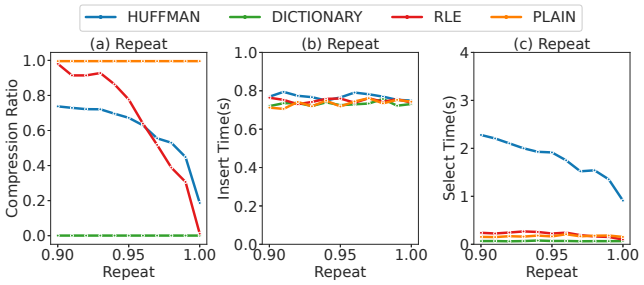


Figure 23: Varying text feature of repeat rate γ_c

In Figure 23, the compression ratio of RLE significantly improves when the character repeat rate γ_c is large, as analyzed in Table 5 and Section 5.2. However, such character repeats may not be prevalent in practice as illustrated in Figure 19(a).

Due to more characters, insert time significantly increases while length is increasing. And insert time is almost unchanged with exponent, domain and repeat varying.

Since HUFFMAN algorithm needs to recover Huffman tree in the process of selecting data, it has significantly higher select time. When repeat becomes larger, the Huffman tree becomes smaller, and select time decreases in Figure 23(c). In contrast, with the increase of length, the Huffman tree becomes larger, and select time increases in Figure 22(c).

8 RELATED WORK

While this study focuses on encoding methods that are proper to implement in time series database management systems, there do exist many other alternatives (see [22] for a survey).

8.1 Lossless Encoding

In addition to the lossless encoding algorithms studied in this paper, the dictionary-based algorithms [32, 34] are not practical to implement for numerical values, since the dictionaries could be too large to store in the PageHeader of Apache IoTDB. Similarly, machine learning based lossless encoding, such as [47] consisting of a transform stage and an encoding stage, needs to conduct reinforcement learning. Not only the models are too large to store, but also the learning is too heavy to process inside databases.

8.2 Lossy Encoding

While we focus on lossless encoding required in databases, the lossy encoding is also practical in other applications especially in end or edge devices. Plato [31] proposes to reduce noise. ODH [30] adopts different lossy algorithms to encode the linear and non-linear data, respectively. Eichinger et al. [25] propose to estimate time series in piecewise polynomial by applying a greedy method and three different online regression algorithms, including a PMR-Midrange algorithm [33], an optimal approximation algorithm [24], and a randomized algorithm [39], for approximating constant functions, straight lines, and polynomials. Fink and Gandhi [26] introduce an algorithm to encode time series by exploiting time series maxima and minima. TRISTRAN [34] and CORAD [32] use autocorrelation in one or multiple time series to improve compression ratio and accuracy.

8.3 General Compression

In Apache IoTDB, a compression step for general data is applied after the time series is encoded, i.e., complementary. The compression algorithms implemented in Apache IoTDB, GZIP [16], SNAPPY [38] and LZ4 [19], all originate from LZ77 [48], looking for the longest match string using a sliding window on the input stream. Nevertheless, the results in Figure 9 show that TS_2DIFF encoding is already efficient, while further applying general purpose compression cannot reduce the space cost further.

9 CONCLUSION

In this paper, we provide both qualitative and quantitative analysis of time series encoding algorithms regarding to various data features. The comparison is conducted in Apache IoTDB, an open-source time series database developed in our preliminary study [43]. First, we profile several features that may affect the performance of encoding. The qualitative and quantitative analysis is thus built on these data features. To evaluate the encoding algorithms, we present a benchmark with real-world data and a data generator for various features.

We notice that different encoding algorithms favor various data features. It motivates us to recommend distinct encoding algorithms referring to the features for different datasets. While some preliminary results of encoding recommender are presented in Appendix A in [17], it is expected to improve the recommender further. For instance, one may employ more advanced machine learning models to train a more accurate recommender. Incremental and transfer learning could also be applied, to address evolving data features and generalize over never-seen datasets. In addition to pursuing more concise encoding, one may also expect to balance the space cost and the time cost of efficient query processing.

ACKNOWLEDGMENTS

This work is supported in part by the National Natural Science Foundation of China (62072265, 62021002), the National Key Research and Development Plan (2021YFB3300500, 2019YFB1705301, 2019YFB1707001), Beijing National Research Center for Information Science and Technology (BNR2022RC01011), and Alibaba Group through Alibaba Innovative Research (AIR) Program. Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

REFERENCES

[1] https://iotdb.apache.org/.
[2] https://www.influxdata.com/.
[3] http://opentsdb.net/.
[4] https://prometheus.io/.
[5] https://github.com/apache/iotdb/tree/research/encoding-exp.
[6] https://github.com/xjz17/iotdb/tree/TSEncoding.
[7] https://thulab.github.io/iotdb-quality/.
[8] https://iotdb.apache.org/UserGuide/Master/Data-Concept/Encoding.html.
[9] https://github.com/thulab/iotdb-benchmark.
[10] https://www.microsoft.com/en-us/download/details.aspx.
[11] https://archive.ics.uci.edu.
[12] https://www.kaggle.com/datasets/eliasdabbas/web-server-access-logs.
[13] https://www.kaggle.com/datasets/winmedals/incident-event-log-dataset.
[14] https://www.kaggle.com/datasets/shawon10/web-log-dataset.
[15] https://www.kaggle.com/datasets/.
[16] https://www.gnu.org/software/gzip/.
[17] https://sxsong.github.io/doc/encoding.pdf.
[18] Anders Aamand, Piotr Indyk, and Ali Vakilian. (learned) frequency estimation algorithms under zipfian distribution. *CoRR*, abs/1908.05198, 2019.
[19] Matej Bartik, Sven Ubik, and Pavel Kubalík. LZ4 compression algorithm on FPGA. In *2015 IEEE International Conference on Electronics, Circuits, and Systems, ICECS 2015, Cairo, Egypt, December 6-9, 2015*, pages 179–182. IEEE, 2015.
[20] Davis W. Blalock, Samuel Madden, and John V. Guttag. Sprintz: Time series compression for the internet of things. *CoRR*, abs/1808.02515, 2018.
[21] Giuseppe Campobello, Antonino Segreto, Sarah Zanafi, and Salvatore Serrano. RAKE: A simple and efficient lossless compression algorithm for the internet of things. In *25th European Signal Processing Conference, EUSIPCO 2017, Kos, Greece, August 28 - September 2, 2017*, pages 2581–2585. IEEE, 2017.
[22] Giacomo Chiarot and Claudio Silvestri. Time series compression: a survey. *CoRR*, abs/2101.08784, 2021.
[23] E. F. Codd. Relational database: A practical foundation for productivity. *Commun. ACM*, 25(2):109–117, 1982.
[24] Marco Dalai and Riccardo Leonardi. Approximations of one-dimensional digital signals under the l_{infty} norm. *IEEE Trans. Signal Process.*, 54(8):3111–3124, 2006.
[25] 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.
[26] Eugene Fink and Harith Suman Gandhi. Compression of time series by extracting major extrema. *J. Exp. Theor. Artif. Intell.*, 23(2):255–270, 2011.
[27] Solomon W. Golomb. Run-length encodings (corresp.). *IEEE Trans. Inf. Theory*, 12(3):399–401, 1966.
[28] Muon Ha and Yulia A. Shichkina. Translating a distributed relational database to a document database. *Data Sci. Eng.*, 7(2):136–155, 2022.
[29] Paul G. Howard and Jeffrey Scott Vitter. Parallel lossless image compression using huffman and arithmetic coding. *Inf. Process. Lett.*, 59(2):65–73, 1996.
[30] Sheng Huang, Yaoliang Chen, Xiaoyan Chen, Kai Liu, Xiaomin Xu, Chen Wang, Kevin Brown, and Inge Halilovic. The next generation operational data historian for iot based on informix. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 169–176. ACM, 2014.
[31] Yannis Katsis, Yoav Freund, and Yannis Papakonstantinou. Combining databases and signal processing in plato. In *Seventh Biennial Conference on Innovative Data Systems Research, CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org, 2015.
[32] Abdelouahab Khelifati, Mourad Khayati, and Philippe Cudré-Mauroux. CORAD: correlation-aware compression of massive time series using sparse dictionary coding. In Chaitanya Baru, Jun Huan, Latifur Khan, Xiaohua Hu, Ronay Ak, Yuanyuan Tian, Roger S. Barga, Carlo Zaniolo, Kisung Lee, and Yanfang Fanny Ye, editors, *2019 IEEE International Conference on Big Data (IEEE BigData), Los Angeles, CA, USA, December 9-12, 2019*, pages 2289–2298. IEEE, 2019.
[33] 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.
[34] Alice Marascu, Pascal Pompey, Eric Bouillet, Michael Wurst, Olivier Verscheure, Martin Grund, and Philippe Cudré-Mauroux. TRISTAN: real-time analytics on massive time series using sparse dictionary compression. In Jimmy Lin, Jian Pei, Xiaohua Hu, Wo Chang, Raghunath Nambiar, Charu C. Aggarwal, Nick Cercone, Vasant G. Honavar, Jun Huan, Bamshad Mobasher, and Saumyadipta Pyne, editors, *2014 IEEE International Conference on Big Data (IEEE BigData 2014), Washington, DC, USA, October 27-30, 2014*, pages 291–300. IEEE Computer Society, 2014.
[35] V. Krishna Nandivada and Rajkishore Barik. Improved bitwidth-aware variable packing. *ACM Trans. Archit. Code Optim.*, 10(3):16:1–16:22, 2013.
[36] Ghim Hwee Ong and Shell-Ying Huang. A data compression scheme for chinese text files using huffman coding and a two-level dictionary. *Inf. Sci.*, 84(1&2):85–99, 1995.
[37] 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.
[38] Horst Samulowitz, Chandra Reddy, Ashish Sabharwal, and Meinolf Sellmann. Snappy: A simple algorithm portfolio. In Matti Järvisalo and Allen Van Gelder, editors, *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, volume 7962 of *Lecture Notes in Computer Science*, pages 422–428. Springer, 2013.
[39] Raimund Seidel. Small-dimensional linear programming and convex hulls made easy. *Discret. Comput. Geom.*, 6:423–434, 1991.
[40] Bin Song, Limin Xiao, Guangjun Qin, Li Ruan, and Shida Qiu. A deduplication algorithm based on data similarity and delta encoding. In Hanning Yuan, Jing Geng, and Fuling Bian, editors, *Geo-Spatial Knowledge and Intelligence - 4th International Conference on Geo-Informatics in Resource Management and Sustainable Ecosystem, GRMSE 2016, Hong Kong, China, November 18-20, 2016, Revised Selected Papers, Part II*, volume 699 of *Communications in Computer and Information Science*, pages 245–253. Springer, 2016.
[41] Julien Spiegel, Patrice Wira, and Gilles Hermann. A comparative experimental study of lossless compression algorithms for enhancing energy efficiency in smart meters. In *16th IEEE International Conference on Industrial Informatics, INDIN 2018, Porto, Portugal, July 18-20, 2018*, pages 447–452. IEEE, 2018.
[42] Jirí Walder, Michal Krátký, and Jan Platos. Fast fibonacci encoding algorithm. In Jaroslav Pokorný, Václav Snásel, and Karel Richta, editors, *Proceedings of the Dateso 2010 Annual International Workshop on DAtabases, TExts, Specifications and Objects, Stedronin-Plazy, Czech Republic, April 21-23, 2010*, volume 567 of *CEUR Workshop Proceedings*, pages 72–83. CEUR-WS.org, 2010.
[43] Chen Wang, Xiangdong Huang, Jialin Qiao, Tian Jiang, Lei Rui, Jinrui Zhang, Rong Kang, Julian Feinauer, Kevin Mcgrail, Peng Wang, Diaohan Luo, Jun Yuan, Jianmin Wang, and Jiaguang Sun. Apache iotdb: Time-series database for internet of things. *Proc. VLDB Endow.*, 13(12):2901–2904, 2020.
[44] Terry A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984.
[45] Raymond Chi-Wing Wong and Ada Wai-Chee Fu. Mining top-k itemsets over a sliding window based on zipfian distribution. In Hillol Kargupta, Jaideep Srivastava, Chandrika Kamath, and Arnold Goodman, editors, *Proceedings of the 2005 SIAM International Conference on Data Mining, SDM 2005, Newport Beach, CA, USA, April 21-23, 2005*, pages 516–520. SIAM, 2005.
[46] Retaj Yousri, Madyan Alsenwi, M. Saeed Darweesh, and Tawfik Ismail. A design for an efficient hybrid compression system for eeg data. In *2021 International Conference on Electronic Engineering (ICEEM)*, pages 1–6, 2021.
[47] Xinyang Yu, Yanqing Peng, Feifei Li, Sheng Wang, Xiaowei Shen, Huijun Mai, and Yue Xie. Two-level data compression using machine learning in time series database. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pages 1333–1344. IEEE, 2020.
[48] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977.