Frequency Domain Data Encoding in Apache IoTDB

Haoyu Wang
BNRist, Tsinghua University
wanghy20@mails.tsinghua.edu.cn

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

ABSTRACT

Frequency domain analysis is widely conducted on time series. While online transforming from time domain to frequency domain is costly, e.g., by Fast Fourier Transform (FFT), it is highly demanded to store the frequency domain data for reuse. However, frequency domain data encoding for efficient storage is surprisingly untouched. We notice that (1) the precision of data value is unnecessarily high after transforming to frequency domain and (2) the data values are with skewed distribution leading to a very large bit width for encoding. To avoid such space waste in both precision and skewness, we devise a *descending bit-packing* encoding for frequency domain data. Specifically, we quantize the data values in proper precision referring to the signal-noise-ratio (SNR) in frequency domain analysis. Moreover, we sort the data values in descending order so that the bit width could be dynamically reduced in encoding. The method has been deployed in Apache IoTDB, an open-source time-series database, not only for directly encoding frequency domain data, but also as a lossy compression of the time domain data. The extensive experiments on the system demonstrate the superiority of our encoding for both frequency domain and time domain data.

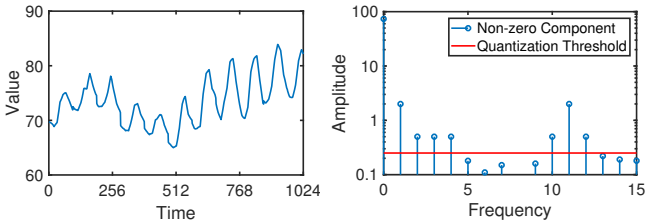
PVLDB Reference Format:
Haoyu Wang and Shaoxu Song. Frequency Domain Data Encoding in Apache IoTDB. PVLDB, 16(2): 282 - 290, 2022.
doi:10.14778/3565816.3565829

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/543202718/iotdb/tree/research/descend.

1 INTRODUCTION

Frequency domain analysis [8] is widespread in analyzing time series data. After performing FFT on a time series, we get a sequence of complex values corresponding to the entire frequency spectrum [29], i.e., the range of frequencies contained by the time series. Each value represents a sine wave at the specified frequency called frequency component (also called "coefficient" interchangeably). Utilizing the amplitude and frequency of components, there are many applications of frequency domain analysis. For example, analysis of vibration data by airlines helps identify potential problems with aircraft [47]. For an electrical company, the identification and repression of ultra-harmonics (frequency components ranging from 2-150 kHz) ensure the measurement accuracy of electric energy meters [35].

Proceedings of the VLDB Endowment, Vol. 16, No. 2 ISSN 2150-8097.
doi:10.14778/3565816.3565829



(a) Time domain (b) Frequency domain

Figure 1: Example of data in time and frequency domain

1.1 Motivation

While frequency domain analysis is repeatedly conducted for various applications, online computing of time-frequency transformation is costly due to its quasilinear time complexity. Unfortunately, most time-series databases, such as Apache IoTDB [5] or InfluxDB [10], do not directly support efficient storage of the frequency domain data for reuse. Let us first identify the unique features for encoding and storing frequency domain data.

(1) *Unnecessarily high precision*: The frequency domain data transformed from time domain, e.g., by FFT, are with very high precision in theory. However, such high precision is not necessary for analysis. For example, when detecting the seasonality [42], we only focus on where the peak is, paying no attention to the exact amplitude of each component. The high precision unfortunately brings a heavy pressure on storage. Figure 1 shows the time domain data of air temperature in a period and its transformed frequency domain data. As shown in Figure 1(b), only 8 components among 1024 are with amplitude greater than 0.25, while most others are ignorable noises (lower than 0.25) with unnecessarily high precision.

(2) *Extremely skewed distribution*: The distribution of amplitudes of frequency components is extremely skewed with huge value spread. In other words, the difference between the maximal and minimal amplitude is several orders of magnitude. For example, as shown in Figure 1(b), the maximal amplitude 73.5 in frequency 0 is 8-bit wide, while the bit widths of other amplitudes are only 1-3. The skewness leads to a serious bit waste if all values are encoded with the same width.

1.2 Intuition

Existing methods cannot handle the precision and skewness well. For lossless encoding Gorilla [38], only few bits are the same for neighboring high-precision values. For differential methods [32], the data range cannot be reduced due to the large difference between nearby values. Besides, fixed-width bit-packing [33] encodes all

values with the maximum bit width, wasting a lot of bits for small values.

To overcome the issue of high precision, quantization [34] is used to reduce to a proper precision. Without knowing the data distribution, manually specifying the precision to quantize is difficult. Instead, we use the standard signal-noise-ratio (SNR) [31] in frequency domain analysis to specify the difference that can be tolerant before and after quantization. After that, frequency domain is sparse because most components are lower than half-precision (the red line) and quantized to 0, as shown in Figure 1(b).

To address data skewness, our solution is specifying unique encoding bit width for each value. Instead of indicating bit width explicitly with extra bits, we specify implicitly as the valid bit width of the previous one. This method only works if valid bit widths are not increasing. Therefore, we sort the non-zero components in descending order in advance, which is the reason why we call our method *descending bit-packing*.

1.3 Contributions

To the best of our knowledge, this is the first study on encoding frequency domain data. Our major contributions are as follows:

(1) We observe the skewed distribution of frequency domain data, and significantly reduce the number of non-zero coefficients by quantization. While the quantization level needs to be determined individually for different datasets due to their distinct value precision, we propose to use SNR to automatically derive the quantization level.

(2) We propose to order the values after quantization such that the number of bits to represent each non-zero coefficient decreases with the descent of values. The more the data distribution is skewed, the faster the bit-width decreases, leading to lower space cost.

(3) We may apply additional compression to the output stream of encoding, to further reduce the space cost with some extra cost in compression/decompression speed. Our proposal has been in use in Apache IoTDB, an open-source time-series database.

(4) We report an extensive experimental evaluation on the system in Section 4. The results illustrate the superiority of our proposal in encoding frequency domain data.

The code of our proposal [4] has been available. The notations frequently used in this paper are listed in [7].

2 FREQUENCY DOMAIN DATA ENCODING

2.1 Overview

The overview of the encoding and decoding procedure is illustrated in Figure 2, using the example data in Figure 1. It starts from the frequency domain data in Figure 2(b) transformed from the time domain data in Figure 2(a). As shown, the transformed frequency domain values are with very high precision, often unnecessary owing to noises. Quantization is thus to map the high-precision values, with heavy storage overhead, to a smaller set of discrete values. For instance, in Figure 2(c), only one digit is reserved after the decimal point.

After quantization, only a small set of non-zero values are preserved, as shown in the red rectangle in Figure 2(c). Note that the values are in a skewed distribution, i.e., some values are significantly larger than others. It is obviously inefficient to assign the same bit-width for all of them in storage. Intuitively, we may consider the values in descending order, such that the latter smaller values will never occupy more space than the former larger ones. The encoding process thus reduces the space assigned to the descending values, in Figures 2(f) and (i). To recover the order of these values in decoding, the index of each value should also be encoded and stored in advance, in Figures 2(e) and (h).

2.2 Quantization

The quantization step considers a number of M values in frequency domain in Figure 2(b), and generates the corresponding low precision values. Only those non-zero values are reserved after quantization as illustrated in Figure 2(c).

2.2.1 Manually Specifying Quantization Level. In quantization, we use a manual integer parameter quantization level β to specify the place of precision in binary as shown in Figure 2(c), which can be implemented with bit operations. In other words, each $y[i]$ is quantized to an integer $\text{round}(y[i] \cdot 2^{-\beta})$ and can be recovered by multiplying 2^{β}. After quantization, frequency domain data turns to proper precision.

2.2.2 Automatically Determining Quantization Level with SNR. Without knowing the data distribution, manually specifying the quantization level β could be difficult. Instead, by using the standard SNR [31] in frequency domain analysis, one may specify the tolerable difference before and after quantization. With such SNR tolerance, we can automatically determine the quantization level β.

Regarding the difference before and after quantization as noise [31], SNR is the ratio of the energy of original data and noise. Let T_{SNR} denote the target SNR, our intuition is to find a maximum β whose actual SNR is not lower than T_{SNR}. Formally, we have

$$10 \log_{10} \frac{\sum_{i=0}^{N-1} y[i]^2}{\sum_{i=0}^{N-1} (y[i] - \text{round}(y[i] \cdot 2^{-\beta}) \cdot 2^{\beta})^2} \geq T_{\text{SNR}} \quad (1)$$

where the left side of the inequality is the definition of SNR.

In order to find a maximum β, we start the search from a small β which always satisfies with formula 1. Since the rounding error $|y[i] - \text{round}(y[i] \cdot 2^{-\beta}) \cdot 2^{\beta}|$ has an upper bound $2^{\beta-1}$, substituting it into formula 1, β is initially set to the following value:

$$\beta = \lfloor \frac{1}{2} \log_2 \frac{10^{-T_{\text{SNR}}/10} \cdot \sum_{i=0}^{N-1} y[i]^2}{N} \rfloor + 1 \quad (2)$$

We keep increasing β until the actual SNR is lower than T_{SNR}.

With the above method, β is determined by T_{SNR}. In Section 4.2.4, we experimentally evaluate the influence of T_{SNR}. Practically, increasing T_{SNR} leads to worse space efficiency but lower accuracy loss, which needs a trade-off according to the detailed situation of frequency domain analysis.

Example 2.1. Refer to Figure 1, suppose $T_{\text{SNR}} = 35dB$, initial β is -5 according to formula 2. We keep increasing it and find that actual SNR is $32.3dB < 35dB$ when $\beta = 0$. At that time, the search of β stops. Figure 3 shows the number of non-zero components M and SNR when β is from -5 to 0. Finally, we select $\beta = -1$ as the quantization level in Figure 2(c).

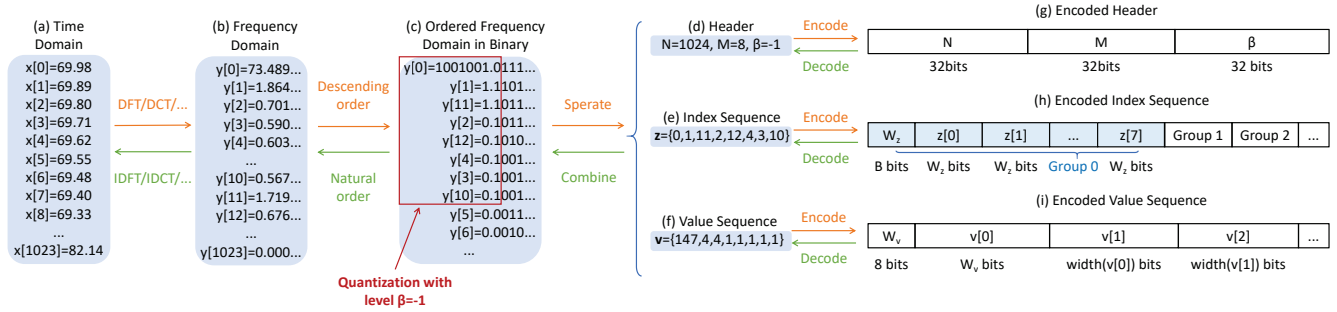


Figure 2: Example of frequency domain data encoding

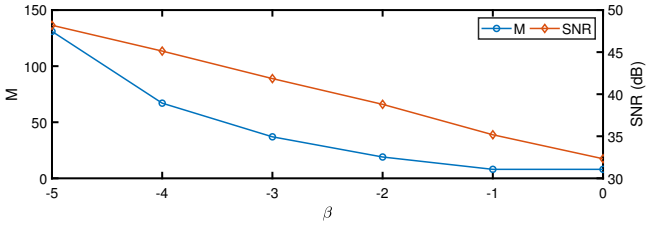


Figure 3: # non-zero components and SNR with varying β

2.3 Index Encoding

To recover the values in the original order in Figure 2(b), we also need to encode and store the index of each value. Consider a sequence $\mathbf{z}$ of M integers ranging from 0 to $N-1$ in Figure 2(e). The output of index encoding is shown in Figure 2(h).

Indexes are encoded with groups. Every 8 integers in $\mathbf{z}$ are grouped together. In a group, we get the max bit width W_z of all integers and encode each of them with W_z bits. Because the maximum of $\mathbf{z}$ is $N-1$, we have

$$W_z \leq \lceil \log_2 N \rceil$$

Thus, W_z is encoded as $B = \lceil \log_2 \log_2 N \rceil$ bits. The main procedure of index encoding is shown in Algorithm 1.

Algorithm 1: IndexEncode($\mathbf{z}, M, N$)

Data: index sequence $\mathbf{z}$, sequence length M, index range N

$B \leftarrow \lceil \log_2 \log_2 N \rceil$;
$i \leftarrow 0$;
while $i < M$ **do**
  $W_z \leftarrow \max\{\text{width}(\text{z[j]}) \mid j \in [i, \min(M-1, i+7)]\}$;
  Encode W_z with B bits;
  for $j = i \rightarrow \min(M-1, i+7)$ **do**
    Encode z[j] with W_z bits;
  $i \leftarrow i + 8$;

Example 2.2. Suppose $N = 1024$ and $M = 8$, we have an index sequence $\mathbf{z} = \{0, 1, 11, 2, 12, 4, 3, 10\}$. All 8 integers belong to the same group. To encode the group, we first calculate $B = \lceil \log_2 \log_2 1024 \rceil = 4$. Then, we find that the max bit width $W_z = \text{width}(12) = 4$. W_z is encoded as `0100` with 4 bits and 8 integers are encoded with 4 bits each as shown in Table 1. Finally, the index sequence is encoded with 36 bits as `01000000 00011011 00101100 01000011 1010`.

2.4 Value Encoding

The value encoding considers a sequence $\mathbf{v}$ of M positive descending integers, as illustrated in Figure 2(e). The key insight is that the encoded bit width of an integer is the valid bit width of the previous integer. Meanwhile, the descending order means that the bit width is never larger than the previous one, which ensures the correctness of value encoding. The value encoding output is shown in Figure 2(i) and the main procedure is shown in Algorithm 2.

Proposition 2.3. *Descending bit-packing uses* $\text{width}(\text{v}[0]) - \text{width}(\text{v}[M-1])$ *more bits in encoding than the total number of valid bits.*

The proofs of all the propositions are shown in [7].

Proposition 2.4. *Compared to fixed-width bit-packing [33] which encodes all values with the maximal width, descending bit-packing never uses more bits.*

Compared to fixed-width bit-packing, when all values have the same valid bit width, both encodings waste no bits. In fact, the superiority of our encoding shows with value skewness. Extremely, if a large value is followed by many zeros, our method only uses $2 \cdot \text{width}(\text{v}[0])$ bits but fix-width method uses $M \cdot \text{width}(\text{v}[0])$ bits, which leads to $M/2\times$ better space efficiency.

Algorithm 2: ValueEncode($\mathbf{v}, M$)

Data: value sequence $\mathbf{v}$, sequence length M

$W_v \leftarrow \text{width}(\text{v}[0])$;
Encode W_v with 8 bits;
for $i = 0 \rightarrow M-1$ **do**
  Encode v[i] with W_v bits;
  $W_v \leftarrow \text{width}(\text{v}[i])$;

Example 2.5. Suppose $M = 8$, we have a value sequence $\mathbf{v} = \{147, 4, 4, 1, 1, 1, 1, 1\}$. First, we encode $W_v = \text{width}(147) = 8$ with 8 bits as `00001000`. Then, we encode each value sequentially as

Table 1: Example of encoding and decoding

i	z[i]	z[i] in binary	v[i]	v[i] in binary
0	0	0000	147	10010011
1	1	0001	4	00000100
2	11	1011	4	100
3	2	0010	1	001
4	12	1100	1	1
5	4	0100	1	1
6	3	0011	1	1
7	10	1010	1	1

shown in Table 1 using 26 bits. The valid bits of each value are underlined. Compared to fixed-width bit-packing strategy which needs $8 \times 8 = 64$ bits, more than half bits are saved.

Connecting the above binary bits, the value sequence is finally encoded. It is encoded with 34 bits as `00001000 10010011 00000100 10000111 11`.

2.5 Index Decoding

The index decoding is the reverse of the index encoding process in Section 2.3, i.e., from Figure 2(h) to (e). With given M, we learn the group partition of the index sequence. There are $\lceil \frac{M}{8} \rceil$ groups. The last group has $[(M-1) \mod 8] + 1$ integers while other groups have 8 integers each. In each group, the first $B = \lceil \log_2 \log_2 N \rceil$ bits record the bit width W_z. All of the following integers are W_z bits wide. Concatenating the integers of each group, the index sequence $\mathbf{z}$ is recovered. The main procedure of index decoding is shown in Algorithm 3.

Algorithm 3: IndexDecode(M, N)

Data: sequence length M, index range N
Result: index sequence $\mathbf{z}$

```
B ← ⌈log2 log2 N⌉;
i ← 0;
while i < M do
    Decode W_z with B bits;
    for j = i → min(M − 1, i + 7) do
        Decode z[j] with W_z bits;
    i ← i + 8;
return z;
```

Example 2.6. Suppose $N = 1024$ and $M = 8$, the encoded index sequence is `01000000 00011011 00101100 01000011 1010`. Since $\lceil \frac{M}{8} \rceil = 1$, integers all belong to the same group.

To encode the group, we first calculate $B = \lceil \log_2 \log_2 1024 \rceil = 4$. Then, we decode the first 4 bits and get $W_z = 4$. After that, we decode for 8 times and 4 bits each as shown in Table 1. Finally, the index sequence is decoded as $\mathbf{z} = \{0, 1, 11, 2, 12, 4, 3, 10\}$.

2.6 Value Decoding

Likewise, the value decoding converts the encoded value sequence in Figure 2(i) back to the value sequence $\mathbf{v}$ in (f). First, we decode

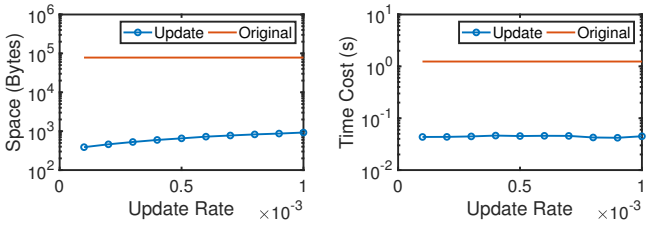


Figure 4: Performance of handling updates

W_v from the binary stream. With bit width W_v, the first integer v[0] is decoded. After that, each integer is decoded with the valid bit width of the previous one. Algorithm 4 shows the main procedure of value decoding.

Algorithm 4: ValueDecode(M)

Data: sequence length M
Result: decoded value sequence $\mathbf{v}$

```
Decode W_v with 8 bits;
for i = 0 → M − 1 do
    Decode v[i] with W_v bits;
    W_v ← width(v[i]);
return v;
```

Example 2.7. Suppose $M = 8$, the encoded value sequence is `00001000 10010011 00000100 10000111 11`. First, $W_v = 8$ is decoded from `00001000`. Then, we decode each value sequentially as shown in Table 1. Finally, the value sequence is decoded as $\mathbf{v} = \{147, 4, 4, 1, 1, 1, 1, 1\}$.

3 DEPLOYMENT IN APACHE IOTDB

3.1 Frequency Domain Data Encoding

For the time series stored in Apache IoTDB, we use User Defined Function (UDF) [6] to transform it to frequency domain. The frequency domain data is also represented as a time series although its timestamps are just a reuse for time domain. Based on the query write-back mechanism of Apache IoTDB, we can directly calculate and store the frequency domain data with SQL sentences. The frequency domain data is encoded by the approach in Section 2 named `DESCEND`. The deployment details are shown in [7].

3.2 Updates and Deletes

In real-world use cases, it is common for lots of time series data to have updates and deletes. For instance, according to our preliminary study [45] in a wind turbine manufacturer, GoldWind, the values could be occasionally misplaced in wrong series during device maintenance, e.g., the year value 2017 is misplaced in the time series of wind speed. Such errors seriously mislead the data science tasks [39], such as ice forecast of wind turbines in wind farms, thus need to be detected [37] and then either repaired or deleted.

In Apache IoTDB, where our proposal is in use, a Log-Structured Merge-Tree (LSM-Tree) [36] is employed to store time series data with immutable TsFiles. The updates and deletes are recorded as

Table 2: Dataset

Name	Data Size	Description
TEMP	171,012	Air temperatures of a wind farm
PV	44,642,859	Voltage of a PV inverter [14]
POWER	2,049,280	Household global active power [1]
GAS	4,178,504	Readings of chemical sensor [28]
HHAR	13,062,475	Smartphone accelerometer samples [44]
GPS	263,718	GPS trajectory of seabirds [12]
ECG	2,415,755	Electrocardiogram (ECG) data [13]
AUDIO	661,500	Acoustic guitar music [8]
NOISE	1,048,576	Synthetic white noise
COSINE	1,048,576	Synthetic cosine signal

mods files [3] auxiliary to TsFile. Likewise, we also use mods files to record the modifications of the frequency domain data (owing to updates and deletes in time domain). Note that the Fourier transform is conducted independently in each window. When the time domain data is deleted or updated, we recalculate the frequency domain of this window and record only the changed components.

Figure 4 presents the space and time costs of handling updates over the TEMP dataset provided by the aforesaid GoldWind. Since the frequency domain is quantized, as illustrated in Figure 1(b), most changes are below the quantization threshold and have no need to record. That is, the modifications are very small compared to the original frequency domain data in Figure 4, while the corresponding time costs are also much lower. It is not surprising that the space costs increase to record more updates.

3.3 Late Arriving Data

According to our statistics [30] in real-world IoT scenarios, only 0.0375% of data points are delayed and the average delayed time is 2.49s. However, in some cases, the data could be seriously delayed. For instance, in the aforesaid wind farm scenario of GoldWind, data arrivals could be delayed for hours during device maintenance. Even worse, when the protocol of a wind turbine is upgraded, the data center may not be able to parse the data for months and ingest them in databases till the next protocol synchronization.

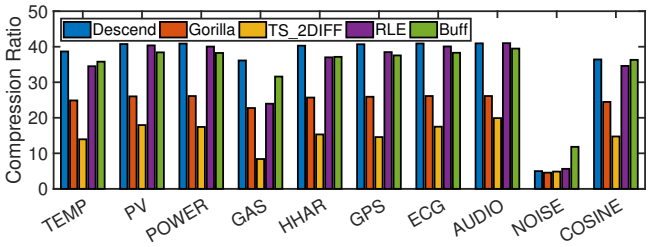
For the points with short delay, Apache IoTDB uses a MemTable to cache the data points and then reorders them by timestamps when flushing to disks. In this case, FFT is conducted over the reordered time series, no longer with late arriving data.

For the points with long delay, the time series flushed to disks are incomplete. FFT needs full knowledge of all the data in a window before it can be encoded. In this sense, we can detect [25] and impute [48] the missing values in time series before FFT. When the delayed data points finally arrive, they are treated as updates of the previously imputed values. The corresponding frequency domain data will be updated accordingly as aforesaid.

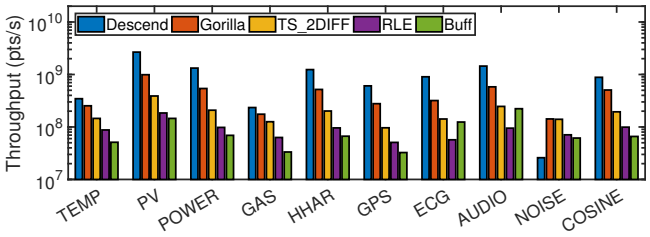
4 EXPERIMENTAL EVALUATION

4.1 Experiment Setup

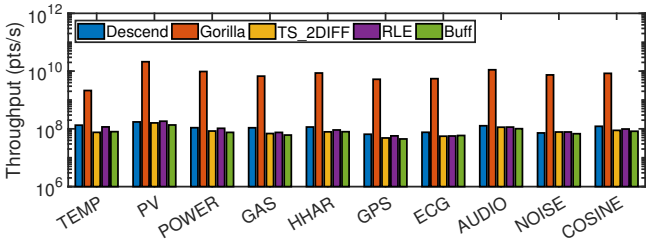
4.1.1 Algorithm. For frequency domain data encoding, we compare our proposed encoding method `Descend` in Section 2 with



(a) Compression ratio



(b) Encoding throughput



(c) Decoding throughput

Figure 5: Performance of frequency domain data encoding

`Gorilla` [38], `TS_2DIFF` [32] and `RLE` [20]. These methods are all lossless and implemented in Apache IoTDB.

4.1.2 Dataset. In the experimental evaluation, we use 8 real datasets and 2 synthetic datasets among various types. Their data sizes and descriptions are shown in Table 2.

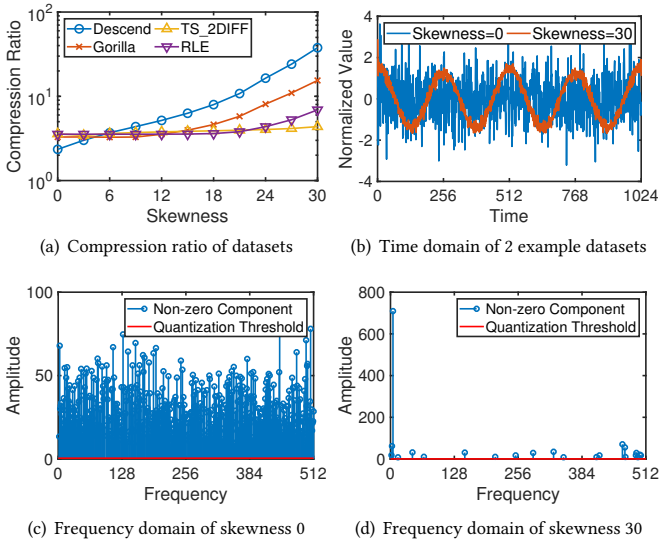
4.1.3 Environment. The evaluations are made via Java native API with tablet size 8192. The physical machine for all experiments is a PC with an Intel(R) Core(TM) i7-9700 CPU and 16 GB RAM.

4.1.4 Evaluation Metric. To evaluate the space efficiency of encodings, compression ratio, the ratio of space occupied before and after encoding, is used as a metric. For comparison, each value is saved as a 64-bit DOUBLE before encoding.

As for time efficiency, we consider encoding throughput, i.e., the number of data points that could be encoded per second. Indeed, our proposed technique only needs to know the set of values in a window for FFT, i.e., high throughput of encoding.

4.2 Frequency Domain Data Encoding

4.2.1 Overall Performance. As shown in Section 3.1, frequency domain data in the experiments is constructed via UDF `STFT`. Time



(a) Compression ratio of datasets (b) Time domain of 2 example datasets

(c) Frequency domain of skewness 0 (d) Frequency domain of skewness 30

Figure 6: Performance with varying skewness

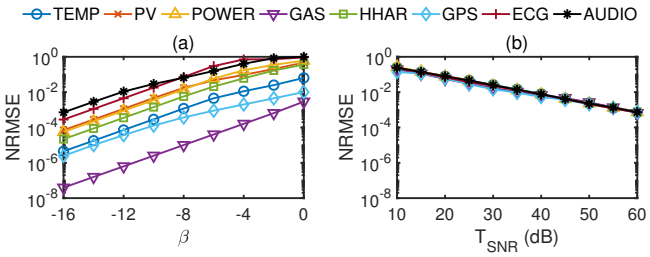


Figure 7: NRMSE in time domain under various quantization

series is stored in Apache IoTDB in advance. We set the window size as 1024 and $\beta = 0$ manually for each dataset.

Figure 5 shows the performance of frequency domain encoding with various methods over different datasets. For space efficiency, in most datasets, `Descend` achieves the highest compression ratio because it captures the sparse feature of frequency domain data and utilizes the value skewness further. It performs badly in NOISE because the energy of white noise distributes uniformly on the entire spectrum, violating the assumption of sparsity. Besides, `RLE` also benefits from sparsity. However, the basic intuition of `Gorilla` and `TS_2DIFF` is flat data change, which is not suitable in frequency domain data leading to worst performance.

As for time efficiency, `Descend` has higher encoding throughput than other algorithms in most datasets since the encoded binary is smaller. For decoding, `Gorilla` utilizes high-speed bit operations to achieve a much higher throughput. However, encoding and decoding are steps with a small proportion during write and query in the database, leading to insignificant user-perceivable influence.

4.2.2 Varying Skewness. As introduced in Section 1.1, the extremely skewed distribution is one of the motivations in this study. Intuitively, if the frequency domain data is more skewed, the values as well as the corresponding bit widths will descend more quickly, leading to better compression performance. Therefore, we introduce *skewness* [11] of the frequency components as a feature, to show better the applicable scenarios and limitations of our `Descend`. To prepare the skewed datasets, we randomly generate the amplitude of each frequency component under a certain skewness. Then, these components are superimposed to obtain time domain data. Figure 6(b) shows the time domains of two representative datasets, with low skewness 0 and high skewness 30. Their frequency domains are shown in Figure 6(c) and (d), respectively. After generating the datasets, we quantize the frequency domain with $T_{\text{SNR}} = 40dB$ and apply various encoding methods. Figure 6(a) shows the compression ratio of different encodings over the datasets with varying skewness.

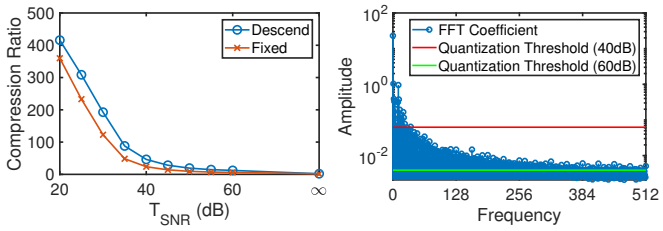
First, referring to the aforesaid intuition, it is not surprising that `Descend` works better with higher skewness. Existing encoding, such as `RLE`, works better with repeated values, which unfortunately are barely observed, e.g., in Figures 6(c) or (d). Nevertheless, with higher skewness, more components are quantized to 0 by our proposal in Figure 6(d) and have no need to store. Thereby, all the alternatives including `RLE` benefit from quantization.

On the contrary, `Descend` performs worse when skewness is extremely low, e.g., 0 in Figure 6(c). In this case, the ordered values decrease slightly in bit-width, while the auxiliary index incurs extra space cost. Alternatives, such as `RLE` relying on repeated values, are not affected much and show better results than our proposal. Nevertheless, it is the case of pure noises, as illustrated in Figures 6(b) and (c), often useless in real applications.

4.2.3 Varying Quantization. Note that the precision of values is usually different in various datasets. It needs huge effort to manually specify β the precision to quantize for each dataset individually as in [34]. To illustrate such differences among different datasets, we consider the loss of quantization. For a series $\mathbf{x}$ in time domain, let $\mathbf{x}'$ be the corresponding series by transforming $\mathbf{x}$ to the frequency domain $\mathbf{y}$ with quantization and transforming $\mathbf{y}$ back to time domain, as the process illustrated in Figure 2. The smaller the NRMSE between $\mathbf{x}$ and $\mathbf{x}'$ is, the more close the series is before and after quantization, i.e., less loss. Figure 7 illustrates the NRMSE under various quantization parameters β and T_{SNR}.

As shown in Figure 7(a), the same β leads to different NRMSE in various datasets with distinct data precision, i.e., various quantization loss. In contrast, the NRMSE of different datasets are almost the same for a certain T_{SNR}, in Figure 7(b). The reason is that T_{SNR} on signal-noise-ratio only filters those components of noises and leads to consistent quantization loss. In this sense, consistently setting the threshold T_{SNR} for various datasets is much easier than manually specifying β individually for each dataset as in [34].

Rather than using all the FFT coefficients for storage, our proposal also considers only very few coefficients. As the example illustrated in Figure 1(b), the quantization reserves only those with amplitude higher than the threshold in the red line. To compare compression performance with `Buff` [34], since it is difficult to manually specify the quantization level β for various datasets as aforesaid, we use the β derived by T_{SNR} for `Buff` as well, i.e., the



(a) Comparison on compression ratio (b) Frequency domain

Figure 8: Evaluation on alternative options

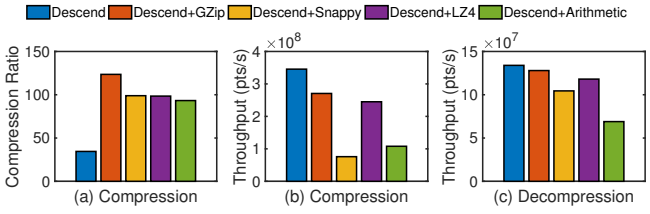


Figure 9: Complement with compression techniques

FFT coefficients are the same between Buff and our proposal. Figure 5 shows that our Descend has better compression ratio than Buff. The reason is that Buff considers only the redundancy in bytes, while our proposal reduces the bit-width for each value in a fine-grained granularity.

4.2.4 Alternative Options. Figure 8(a) reports the compression ratio of two different choices in the design of encoding, over the TEMP dataset. As shown, under various quantization threshold T_{SNR}, the descending bit-packing Descend occupies space no higher than the fixed width bit encoding Fixed. The results are consistent with the analysis in Proposition 2.4. The benefit of Descend is to dynamically reduce the bit-width with the descent of values, as illustrated in Table 1 in Example 2.5, more efficient than Fixed in bit-width.

Figure 8(a) also presents the compression ratio of different choices in the design of quantization, where the quantization threshold $T_{SNR} = \infty$ denotes no quantization applied. The smaller the threshold T_{SNR} is, corresponding to a higher red line in Figure 8(b), the more the FFT coefficients are quantized to zero and thus have no need to record, i.e., better compression ratio. Of course, as the decrease of T_{SNR} in Figure 7(b), the corresponding loss in time domain (cons) increases with the decrease of space cost (pros), a trade-off.

As the skewed data distribution illustrated in Figure 8(b), when T_{SNR} is large, corresponding to the lower green line, more values without much differences are under consideration (above the green line). The decrease of bit-width by Descend is thus limited, with compression ratio close to Fixed bit-width in Figure 8(a). On the other hand, for more efficient quantization with smaller T_{SNR}, the values, e.g., above the red line in Figure 8(b), have more significant differences. It is the case where Descend performs better, and thus shows more clear improvement compared to Fixed in Figure 8(a). In this sense, descending bit encoding (Descend) combined with the more efficient quantization (smaller T_{SNR}) indeed leads to a clearly better solution compared to the Fixed width bit encoding.

4.2.5 Complement with Compression Techniques. Similar to the JPEG approach [46] to image encoding, an additional lossless compression can also be applied to the output stream of Descend. In this sense, our proposal of quantization and encoding is complementary to the compression techniques, such as GZIP [9], LZ4 [23], SNAPPY [2], Arithmetic [40], etc.

Figure 9 presents the results of Descend with/without additional compression, over the TEMP dataset. As shown, various compression techniques further improve more or less the compression ratio in Figure 9(a), with some extra cost in compression/decompression speed in Figure 9(b), again a trade-off. In Apache IoTDB, users can choose whether or not to apply additional compression after encoding, declared by SQL.

4.3 Application in Data Science

Frequency domain is widely used in time series data science tasks. The proposed compression method does not only reduce the storage but also improves the efficiency of data analytics tasks without sacrificing effectiveness.

4.3.1 Similarity Search. To search similar time series, [15] proposes to compute the Euclidean distance on the amplitude of Fourier coefficients. In the similarity search evaluation, we extract 100 time series from each dataset as the data source, and sample some as the queries to search the nearest neighbors from the source. For online-computing, we conduct FFT online and search the nearest neighbors as the ground truths (with accuracy 1). For compressed-store, the Fourier coefficients are encoded and stored in advance, and thus the query only needs to decode them instead of conducting FFT. Since there is a quantization in encoding, the decoded Fourier coefficients are different from the origin and may lead to different query results. We evaluate the accuracy by comparing them with those returned by online-computing.

Figure 10 shows the accuracy and time cost of similarity search by varying T_{SNR} of quantization. The accuracy increases with the increase of T_{SNR}, i.e., more Fourier coefficients are reserved in quantization. The corresponding time cost is of course higher. Nevertheless, with $T_{SNR} \geq 40dB$, the accuracy is close to 1. Meanwhile, the time cost is only about half of the online-computing. It illustrates that frequency domain data encoding with proper quantization can significantly reduce the time cost without losing much accuracy of time series similarity search.

4.3.2 Clustering. Frequency domain features are also used in clustering [27]. Similar to Section 4.3.1, we consider time series from each dataset as a class. The K-Means++ clustering [17] is conducted, using the Euclidean distance on the amplitude of Fourier coefficients. Again, the online-computing conducts FFT online for clustering, while the compressed-store only needs to decode the stored Fourier coefficients with quantization.

Figure 11 shows the purity performance of clustering with different T_{SNR} of quantization. As shown, the results are generally similar to similarity search in Figure 10, i.e., significantly improving the efficiency but not losing much effectiveness with proper quantization. Note that with $T_{SNR} \geq 20dB$, the purity of compression-store

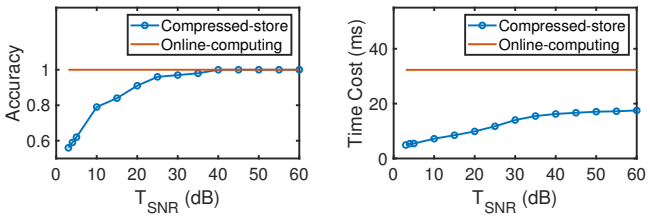


Figure 10: Application in similarity search

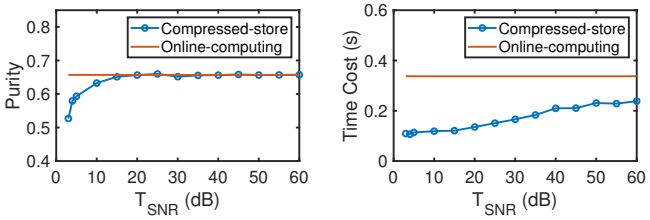


Figure 11: Application in clustering

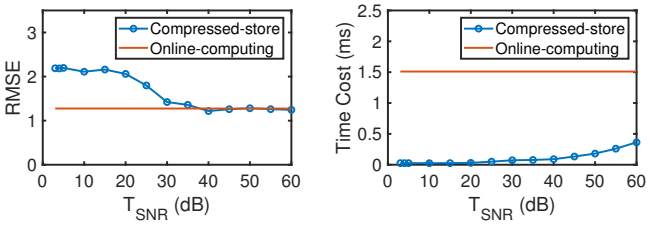


Figure 12: Application in forecasting

is already close to online-computing. The reason is that finding relatively similar series is sufficient for clustering, while the similarity search needs to return the most similar one with less tolerance to noises, requiring a larger T_{SNR}.

4.3.3 Forecasting. Frequency domain also helps in forecasting applications [18]. It builds ARIMA models on Fourier coefficients to forecast those in the next period and thus the time domain. The evaluation is conducted on the TEMP dataset with a period of 24 hours. We forecast the temperatures the next day with the data of the past 14 days. RMSE between the forecast and the real observation is used as a metric. Similar to Section 4.3.1, we also compare the performance between online-computing of all coefficients and compressed-store with quantized ones.

Figure 12 shows RMSE and time cost of forecasting. The results are similar to those in similarity search and clustering. That is, with proper quantization, e.g., $T_{\text{SNR}} = 40dB$, decoding the frequency domain data in compressed-store has almost the same forecasting RMSE as online-computing, but with a much lower time cost.

5 RELATED WORK

5.1 Bit-Packing

Bit-packing is a widely used technique in encoding integers. For an integer, the bits of its binary representation except leading zeros are called valid bits. The core idea of bit-packing is to drop the leading zeros of an integer and save only the valid bits. A selector is used to indicate the encoding bit width of the following values. Some works encode a fixed-size block in a time and specify the encoding bit width as the maximal of valid bit widths of the block [33, 49]. To avoid the influence of some extremely large values, some works store them independently and bit-pack the other values [41, 43]. However, bit-packing meets challenges in the trade-off of block size. Small block size attenuates the effect of large values [19], but leads to more selectors, while large block size is the opposite. In this field, Simple8b [16] is a brilliant implementation which applies different block sizes for different encoding bit widths. Different from the above works, our descending bit-packing assigns unique bit width for each value implicitly without selectors.

5.2 Sketch-based Representation

The concept of quantization is similar to the idea of sketch, as a compressed lossy representation for a specific problem at hand. For instance, HyperLogLog [26] uses random binning via hash function to estimate the number of distinct values. As illustrated in Figure 1(b), most values become zero after quantization and are thus efficient in storage, but do not help in counting distinct values, and vice versa. Moreover, T-Digest [24] uses an uneven binning for percentiles estimation due to a prior that the accuracy at extremes is more important. However, this prior is not suitable for our encoding because all non-zero values are equivalently important for frequency domain data and need to be encoded.

5.3 Time Series Compression

Chiarot and Silvestri [22] present a survey of time series compression techniques. It covers a broad range of approaches in several categories, such as dictionary-based methods, functional approximations, auto-encoders and sequential algorithms. The frequency domain encoding, considered in our study, however, is not discussed. LFZip [21] uses a fixed 16-bit quantization. In contrast, we use a quantization level that can be easily adjusted referring to SNR. Moreover, the bit-width of each value is not fixed either, but descends with the decrease of values, for more efficient space.

6 CONCLUSION

In this paper, we first identify the unique features in encoding frequency domain data, i.e., high precision and skewed distribution. To address the identified precision and skewness issues, we propose to determine the quantization level by signal-noise-ratio (SNR) and order the values such that the bit-width descends in encoding. We deploy the proposed encoding in Apache IoTDB and apply to several data science tasks. The superiority of our proposal is shown by extensive experiments.

ACKNOWLEDGMENTS

This work is supported in part by the National Natural Science Foundation of China (62072265, 62232005, 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] http://archive.ics.uci.edu/ml/datasets/Individual+household+electric+power+consumption.
[2] http://google.github.io/snappy/.
[3] https://cwiki.apache.org/confluence/display/IOTDB/Data+Manipulation.
[4] https://github.com/543202718/iotdb/tree/research/descend.
[5] https://iotdb.apache.org/.
[6] https://iotdb.apache.org/UserGuide/Master/Process-Data/UDF-User-Defined-Function.html.
[7] https://sxsong.github.io/doc/frequency.pdf.
[8] https://ww2.mathworks.cn/help/signal/ug/practical-introduction-to-frequency-domain-analysis.html.
[9] https://www.gzip.org/.
[10] https://www.influxdata.com/.
[11] https://www.itl.nist.gov/div898/handbook/eda/section3/eda35b.htm.
[12] https://www.kaggle.com/datasets/saurabhshahane/predicting-animal-behavior-using-gps.
[13] https://www.kaggle.com/datasets/shayanfazeli/heartbeat.
[14] https://zenodo.org/record/4467774#.Yj1yWtBByUk.
[15] R. Agrawal, C. Faloutsos, and A. N. Swami. Efficient similarity search in sequence databases. In D. B. Lomet, editor, *Foundations of Data Organization and Algorithms, 4th International Conference, FODO'93, Chicago, Illinois, USA, October 13-15, 1993, Proceedings*, volume 730 of *Lecture Notes in Computer Science*, pages 69–84. Springer, 1993.
[16] V. N. Anh and A. Moffat. Index compression using 64-bit words. *Softw. Pract. Exp.*, 40(2):131–147, 2010.
[17] D. Arthur and S. Vassilvitskii. k-means++: the advantages of careful seeding. In N. Bansal, K. Pruhs, and C. Stein, editors, *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, pages 1027–1035. SIAM, 2007.
[18] M. Beiraghi and A. Ranjbar. Discrete fourier transform based approach to forecast monthly peak load. In *2011 Asia-Pacific Power and Energy Engineering Conference*, pages 1–5. IEEE, 2011.
[19] D. W. Blalock, S. Madden, and J. 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.
[20] J. Capon. A probabilistic model for run-length coding of pictures. *IRE Trans. Inf. Theory*, 5(4):157–163, 1959.
[21] S. Chandak, K. Tatwawadi, C. Wen, L. Wang, J. A. Ojea, and T. Weissman. Lfzip: Lossy compression of multivariate floating-point time series data via improved prediction. In A. Bilgin, M. W. Marcellin, J. Serra-Sagristà, and J. A. Storer, editors, *Data Compression Conference, DCC 2020, Snowbird, UT, USA, March 24-27, 2020*, pages 342–351. IEEE, 2020.
[22] G. Chiarot and C. Silvestri. Time series compression: a survey. *CoRR*, abs/2101.08784, 2021.
[23] Y. Collet et al. Lz4: Extremely fast compression algorithm. *code. google. com*, 2013.
[24] T. Dunning and O. Ertl. Computing extremely accurate quantiles using t-digests. *CoRR*, abs/1902.04023, 2019.
[25] C. Fang, S. Song, and Y. Mei. On repairing timestamps for regular interval time series. *Proc. VLDB Endow.*, 15(9):1848–1860, 2022.
[26] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*, pages 137–156. Discrete Mathematics and Theoretical Computer Science, 2007.
[27] K. Fokianos and V. J. Promponas. Biological applications of time series frequency domain clustering. *Journal of Time Series Analysis*, 33(5):744–756, 2012.
[28] J. Fonollosa, S. Sheik, R. Huerta, and S. Marco. Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring. *Sensors and Actuators B: Chemical*, 215:618–629, 2015.
[29] D. Galar and U. Kumar. Chapter 3 - preprocessing and features. In D. Galar and U. Kumar, editors, *eMaintenance*, pages 129–177. Academic Press, 2017.
[30] Y. Kang, X. Huang, S. Song, L. Zhang, J. Qiao, C. Wang, J. Wang, and J. Feinauer. Separation or not: On handing out-of-order time-series data in leveled lsm-tree. In *38th IEEE International Conference on Data Engineering, ICDE 2022, (Virtual) Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 2022.
[31] S. Khalid. *Introduction to data compression*. 2009.
[32] S. T. Klein and M. Meir. Delta encoding in a compressed domain. In J. Holub and J. Zdárek, editors, *Proceedings of the Prague Stringology Conference 2009, Prague, Czech Republic, August 31 - September 2, 2009*, pages 55–64. Prague Stringology Club, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, 2009.
[33] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Softw. Pract. Exp.*, 45(1):1–29, 2015.
[34] C. Liu, H. Jiang, J. Paparrizos, and A. J. Elmore. Decomposed bounded floats for fast compression and queries. *Proc. VLDB Endow.*, 14(11):2586–2598, 2021.
[35] K. Murakawa, N. Hirasawa, H. Ito, and Y. Ogura. Electromagnetic interference examples of telecommunications system in the frequency range from 2khz to 150khz. In *2014 International Symposium on Electromagnetic Compatibility, Tokyo*, pages 581–584. IEEE, 2014.
[36] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.
[37] D. P and S. S. Abraham. Fairlof: Fairness in outlier detection. *Data Sci. Eng.*, 6(4):485–499, 2021.
[38] T. Pelkonen, S. Franklin, P. Cavallaro, Q. Huang, J. Meza, J. Teller, and K. Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proc. VLDB Endow.*, 8(12):1816–1827, 2015.
[39] Z. Qi, H. Wang, and A. Wang. Impacts of dirty data on classification and clustering models: An experimental evaluation. *J. Comput. Sci. Technol.*, 36(4):806–821, 2021.
[40] J. Rissanen and G. G. Langdon. Arithmetic coding. *IBM Journal of research and development*, 23(2):149–162, 1979.
[41] B. Schlegel, R. Gemulla, and W. Lehner. Fast integer compression using SIMD instructions. In A. Ailamaki and P. A. Boncz, editors, *Proceedings of the Sixth International Workshop on Data Management on New Hardware, DaMoN 2010, Indianapolis, IN, USA, June 7, 2010*, pages 34–40. ACM, 2010.
[42] K. Schwarz, M. Sideris, and R. Forsberg. The use of fft techniques in physical geodesy. *Geophysical Journal International*, 100(3):485–514, 1990.
[43] J. Shieh and E. J. Keogh. *i*sax: disk-aware mining and indexing of massive time series datasets. *Data Min. Knowl. Discov.*, 19(1):24–57, 2009.
[44] A. Stisen, H. Blunck, S. Bhattacharya, T. S. Prentow, M. B. Kjærgaard, A. K. Dey, T. Sonne, and M. M. Jensen. Smart devices are different: Assessing and mitigatingmobile sensing heterogeneities for activity recognition. In J. Song, T. F. Abdelzaher, and C. Mascolo, editors, *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems, SenSys 2015, Seoul, South Korea, November 1-4, 2015*, pages 127–140. ACM, 2015.
[45] Y. Sun, S. Song, C. Wang, and J. Wang. Swapping repair for misplaced attribute values. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pages 721–732. IEEE, 2020.
[46] A. Tinku and T. Ping-Sing. Jpeg–still image compression standard. *JPEG2000 Standard for Image Compression*, pages 55–78, 2005.
[47] I. Trendafilova, M. P. Cartmell, and W. Ostachowicz. Vibration-based damage detection in an aircraft wing scaled model using principal component analysis and pattern recognition. *Journal of Sound and Vibration*, 313(3-5):560–566, 2008.
[48] A. Zhang, S. Song, Y. Sun, and J. Wang. Learning individual models for imputation. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 160–171. IEEE, 2019.
[49] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Super-scalar RAM-CPU cache compression. In L. Liu, A. Reuter, K. Whang, and J. 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.