Frequency Domain Data Encoding in Apache IoTDB

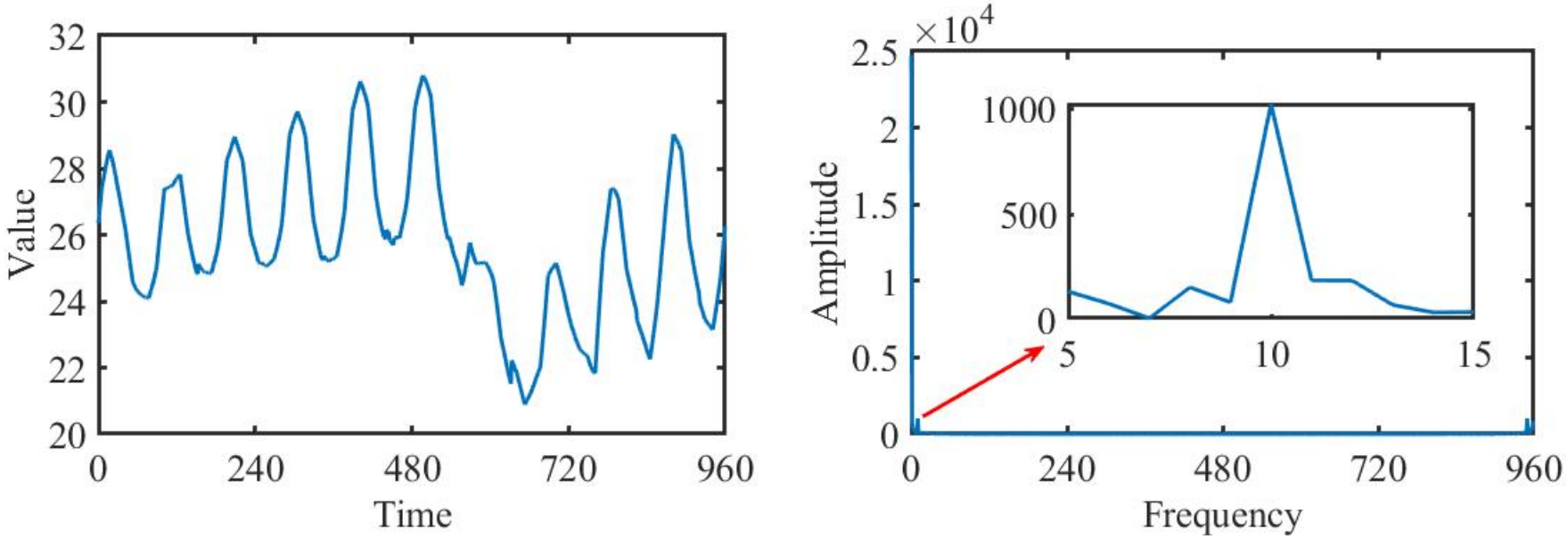
Haoyu Wang, Shaoxu Song

BNRist, Tsinghua University

Proceedings of the VLDB Endowment, PVLDB 16 (2022)

Background

- With the rapid development of IIoT, a lot of time series data is produced and stored in time-series databases.
- **Frequency domain analysis** is widespread in analyzing time series data.
  - Period discovery: finding the period with the index of peaks in the frequency domain.



Motivation and Problem

- The complexity of FFT is O(NlogN). While frequency domain analysis is repeatedly conducted for various applications, online computing of time-frequency transformation is costly due to its quasilinear time complexity.
- Intuition:
  - The **pre-computing and storage of frequency domain data** is significant.
- Problem:
  - **How to encode and store the frequency domain data in time-series databases?**

Challenge

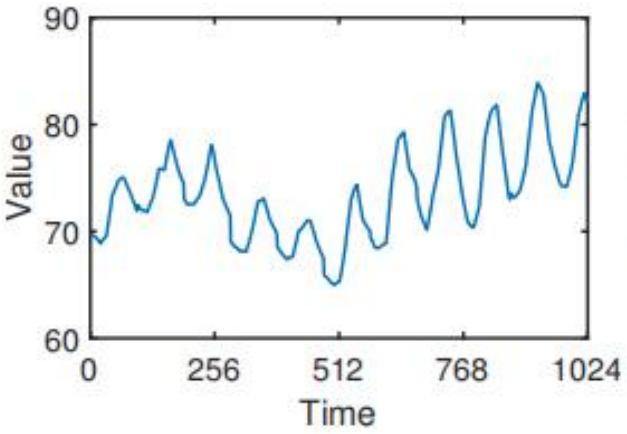
- **How to encode and store the frequency domain data in time-series databases?**
  - A plain encoding as IEEE 754 floating-point data is not realistic due to the large space cost.
  - There is **no existing work** target on frequency domain data encoding.
  - Most time-series databases, such as Apache IoTDB or InfluxDB, **do not directly support** efficient storage of the frequency domain data for reuse.

Contribution

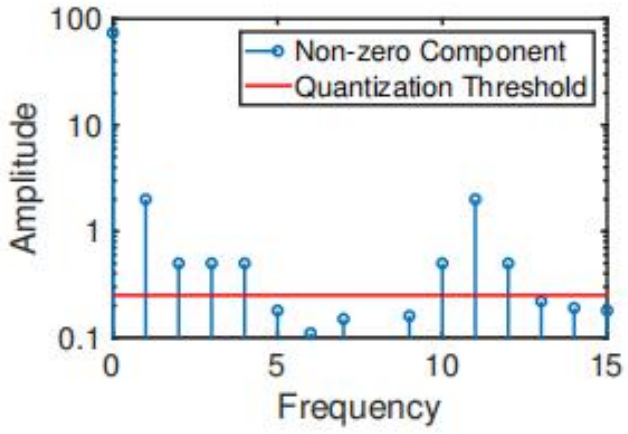
- We observe the **skewed distribution** of frequency domain data, and significantly reduce the number of non-zero coefficients by quantization. While the quantization level needs to be determined individually for different datasets due to their distinct value precision, we propose to **use SNR to automatically derive the quantization level**.
- 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.
- 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.
- We report an **extensive experimental evaluation** on the system. The results illustrate the superiority of our proposal in encoding frequency domain data.

Data Feature of Frequency Domain

- **Unnecessarily high precision**
  - Such high precision is not necessary for analysis.
- **Extremely skewed distribution**
  - The skewness leads to a serious bit waste if all values are encoded with the same width.



(a) Time domain

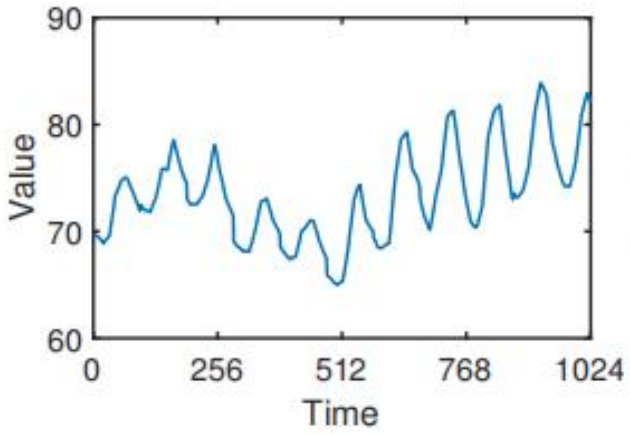


(b) Frequency domain

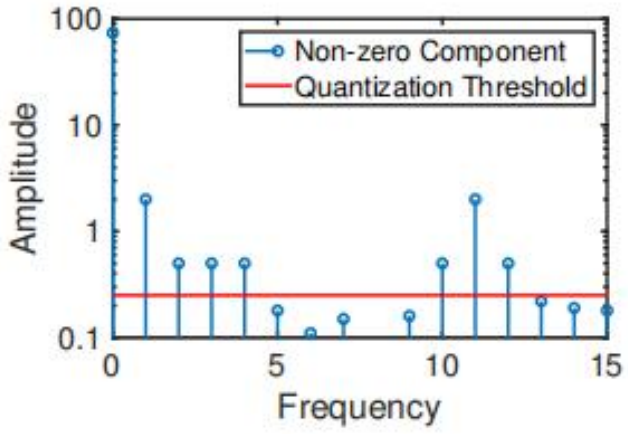
y[0]=73.4886113281250…
y[1]=1.86352736056101…
y[2]=0.700581753631760…
y[3]=0.590147317780781…
y[4]=0.602915456564395…
y[5]=0.182843351988000…
y[6]=0.117359057048072…
y[7]=0.151711502880389…
y[8]=0.0720156288113422…
y[9]=0.166983324499835…
…

Solution

- Unnecessarily high precision => **Quantization**
  - Quantization is used to reduce to a proper precision. After that, frequency domain is **sparse** because most components are quantized to 0.
- Extremely skewed distribution => **Descending bit-packing**
  - Descending Bit-packing **implicitly** specifies **unique** encoding bit width for each value.



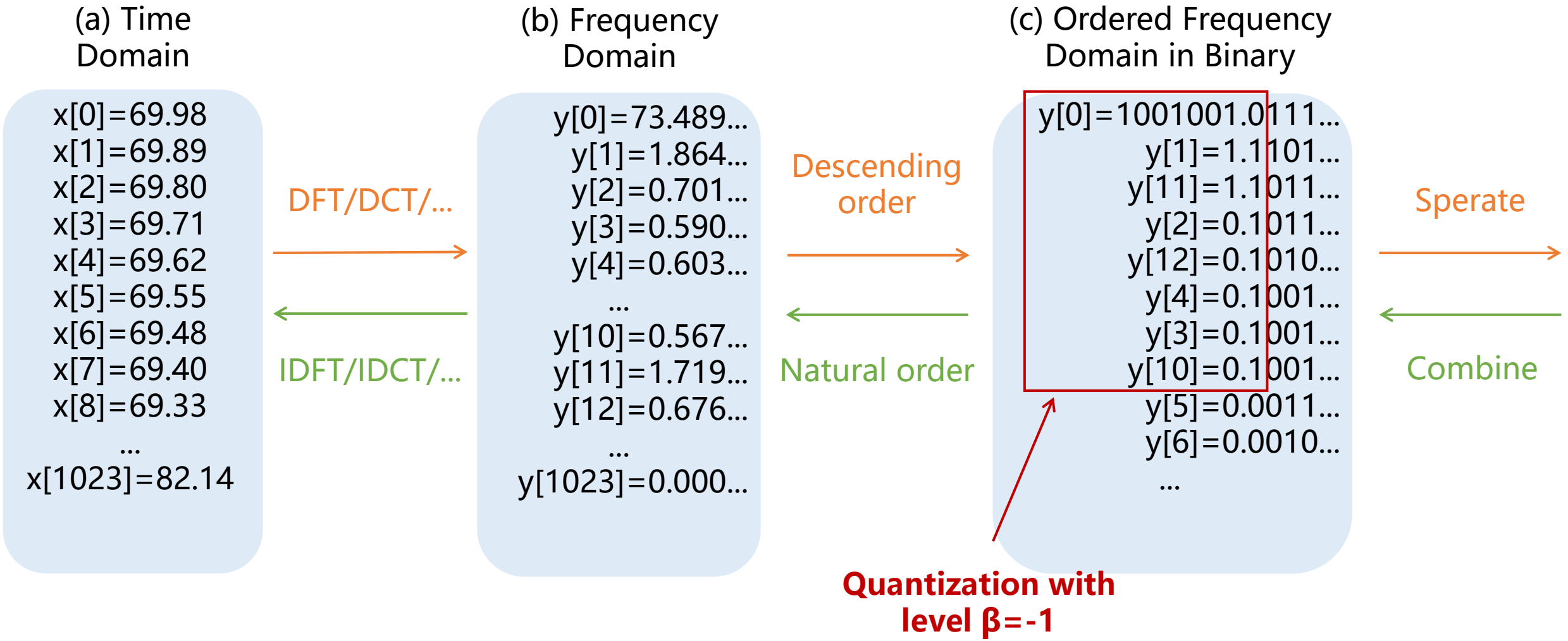
(a) Time domain



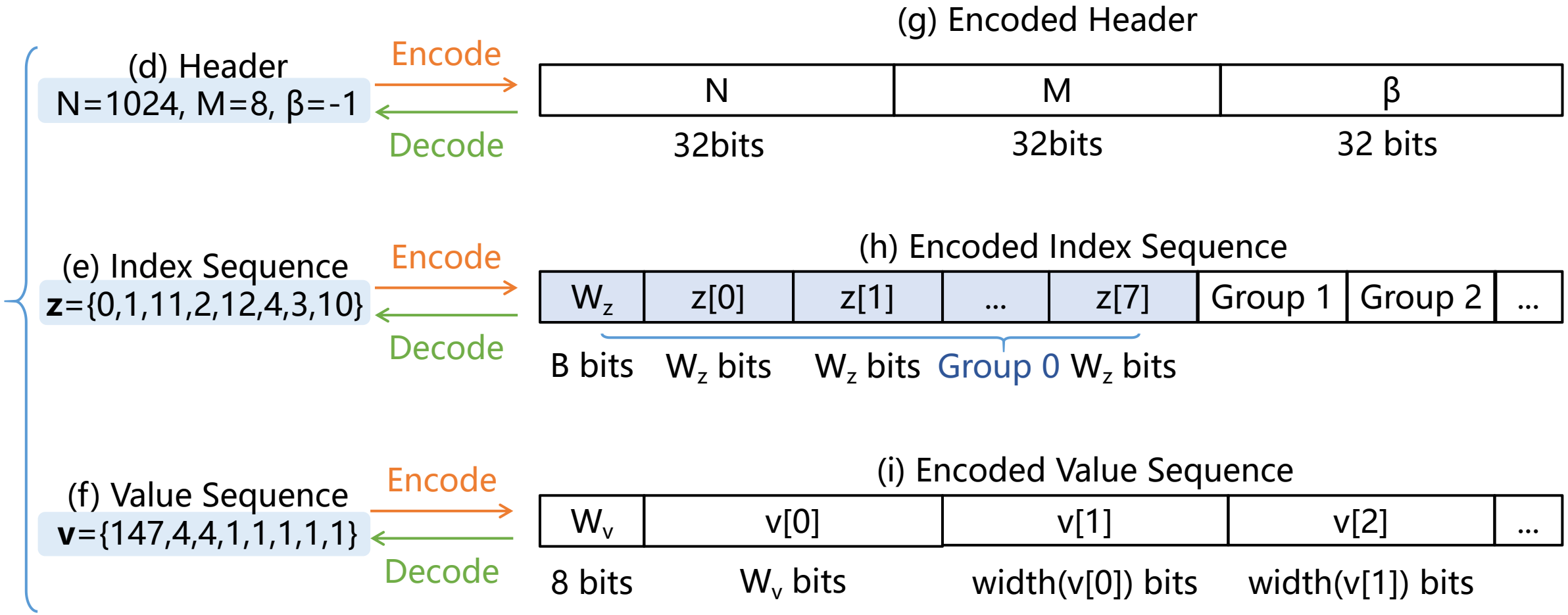
(b) Frequency domain

```
y[0]=73.4886113281250...    => 73.5
y[1]=1.86352736056101...    => 2.0
y[2]=0.700581753631760...   => 0.5
y[3]=0.590147317780781...   => 0.5
y[4]=0.602915456564395...   => 0.5
y[5]=0.182843351988000...   => 0.0
y[6]=0.117359057048072...   => 0.0
y[7]=0.151711502880389...   => 0.0
y[8]=0.0720156288113422...  => 0.0
y[9]=0.166983324499835...   => 0.0
...
```

Overview (Quantization & Reorder)



Overview (Encoding & Decoding)



Quantization

- Quantization reduces the **precision and bit width** at once.
- **Manually** specifying quantization level
  - Quantizatoin level β specifies the place of **precision in binary.**
    - Negative for the reserved bits after the radix point
    - Positive for the discarded bits before the radix point
    - Encoding: y[i] is quantized to integer round(y[i]·$2^{-\beta}$)
    - Decoding: the integer is recoved by multiplying 2^{β}
  - **Bit operation** is used to speed up.

y[0]=1001001.0111...
y[1]=1.1101...
y[11]=1.1011...
y[2]=0.1011...
y[12]=0.1010...
y[4]=0.1001...
y[3]=0.1001...
y[10]=0.1001...
y[5]=0.0011...
y[6]=0.0010...
...

Quantization with level β=1

S Exp=6 Fraction, reserving the first Exp-β=6-(-1)=7 bits

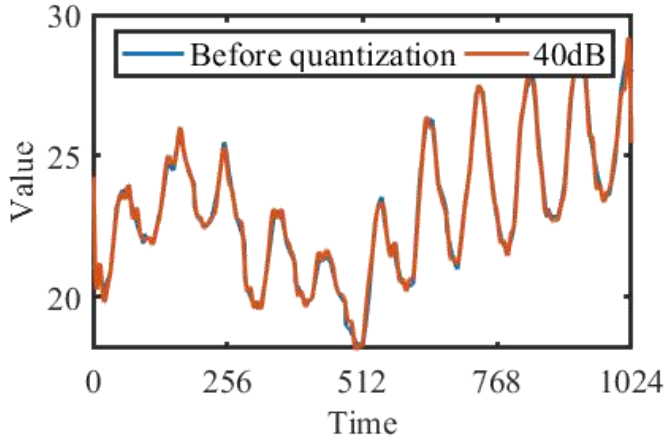
0 100 0000 0101 0010 0101 1111 0100 0101 0110 1000 0111 0010 1011 0001 0110 0010

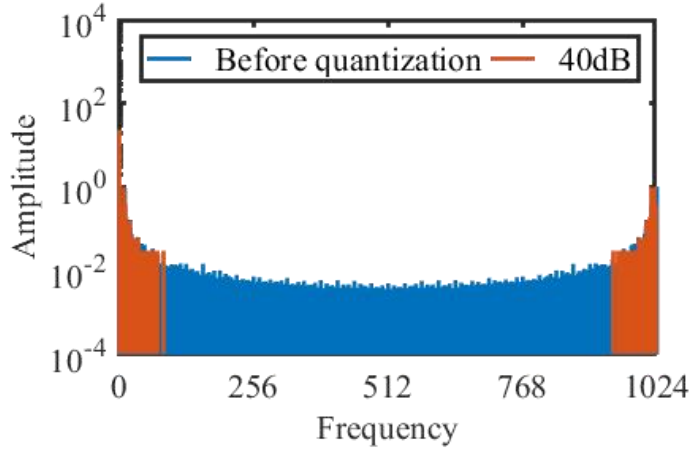
1 0010 010

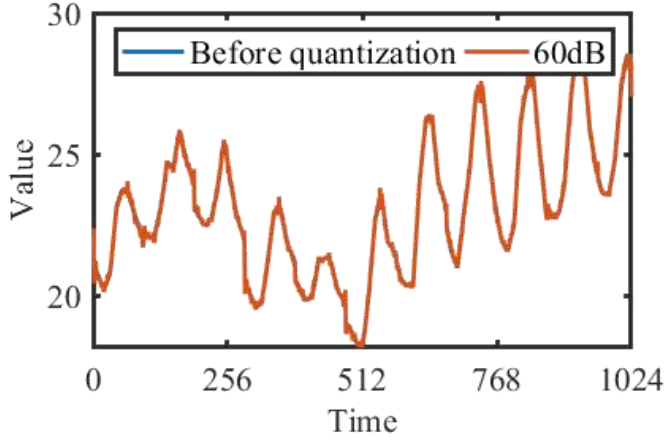
- Without knowing the data distribution, manually specifying β could be difficult.

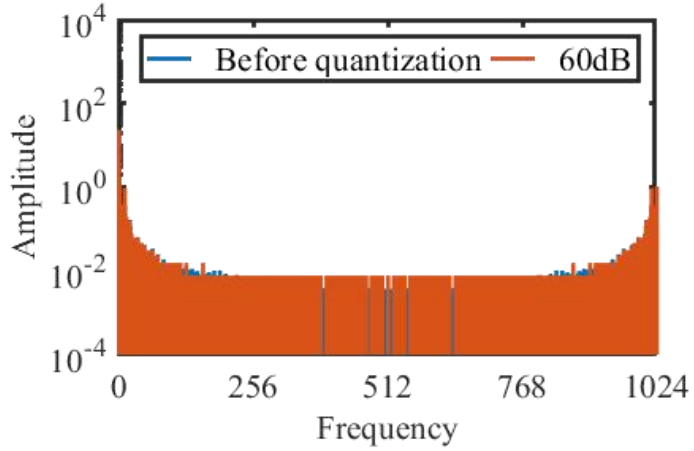
Automatically Determining with SNR

- Automatically determining quantization level with SNR
  - Regarding the difference before and after quantization as noise, SNR is **the ratio of the energy of original data and noise**
    - ➢ Energy is the **square sum** of all elements in the sequence.
    - ➢ **Logarithmic scale** is often used for SNR with the unit **dB**.







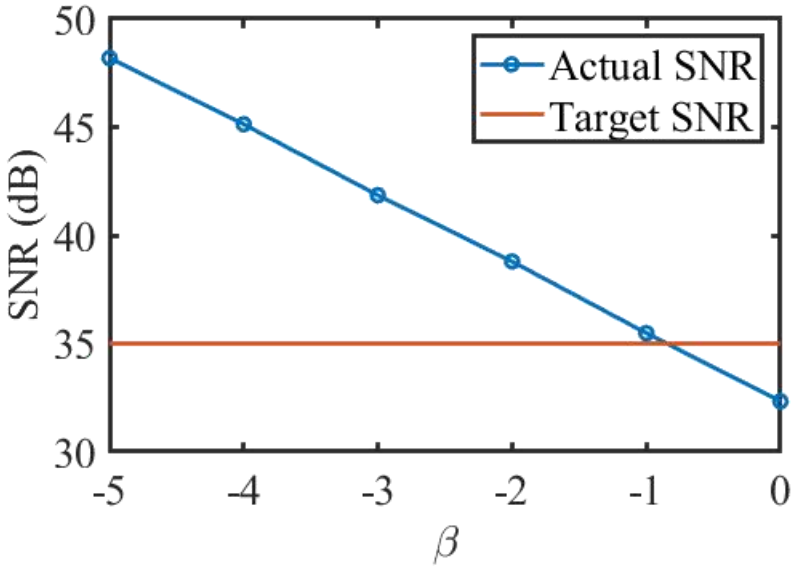


Automatically Determining with SNR

- Automatically determining quantization level with SNR
  - Intuition: find a **maximum β** whose actual SNR is not lower than target SNR T_{SNR}.

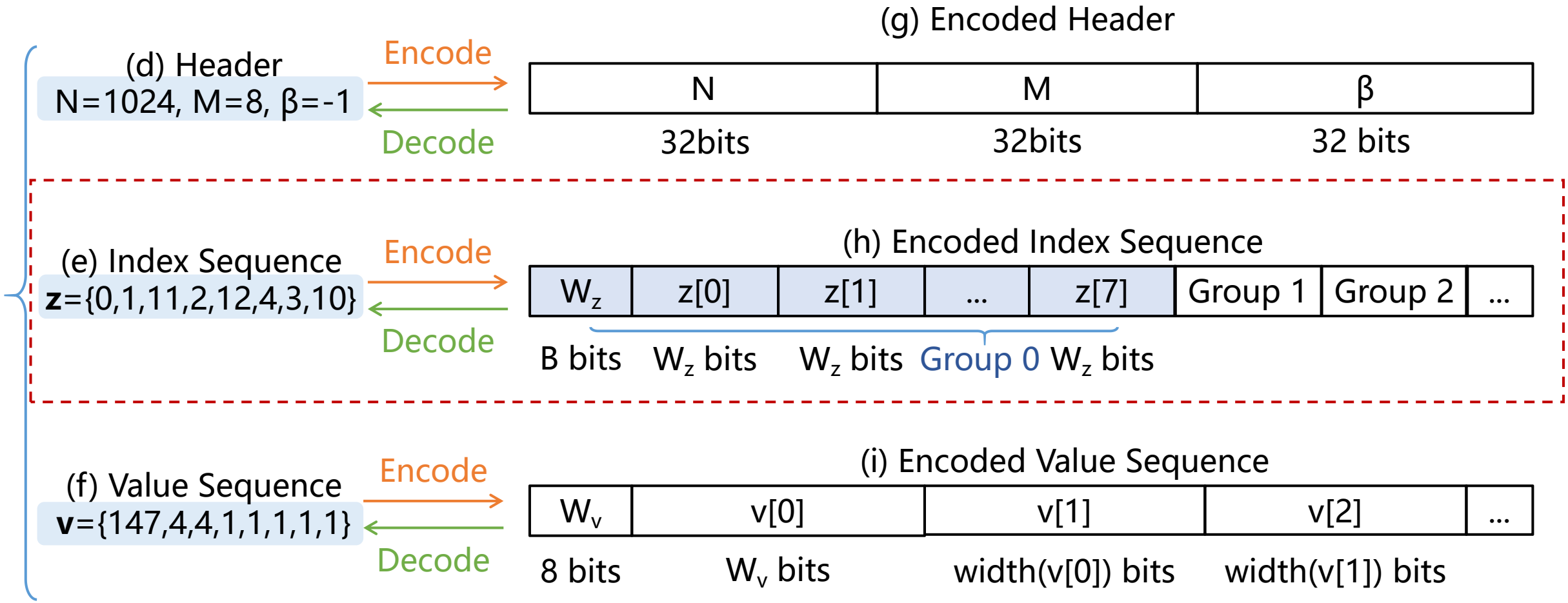
$$10\log_{10}\frac{\sum_{i=0}^{N-1} y[i]^2}{\sum_{i=0}^{N-1}(y[i]-\text{round}(y[i]\cdot 2^{-\beta})\cdot 2^{\beta})^2} \geq T_{\text{SNR}}$$

Definition of SNR



  - Operation:
    - Set β initially as $\beta = \lfloor \frac{1}{2}\log_2 \frac{10^{-T_{\text{SNR}}/10}\cdot\sum_{i=0}^{N-1} y[i]^2}{N} \rfloor + 1$
    - Keep increasing β until the actual SNR is lower than T_{SNR}

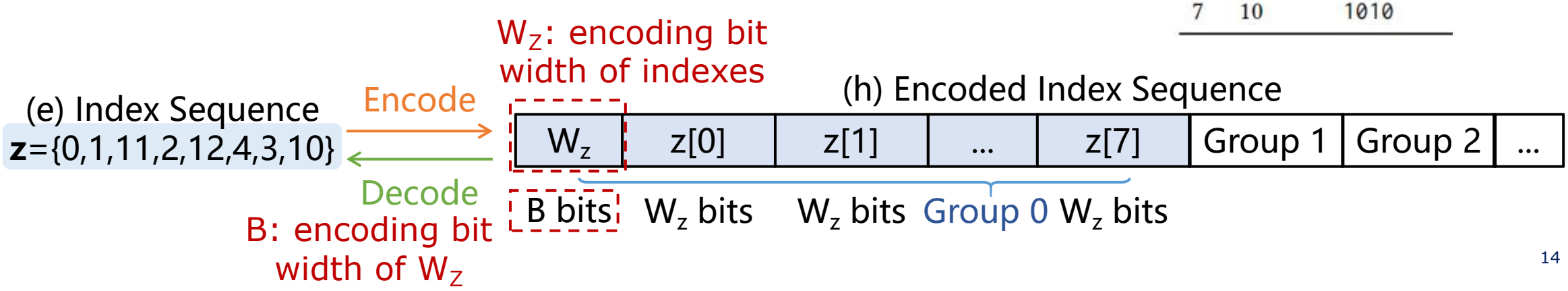
Overview (Encoding & Decoding)



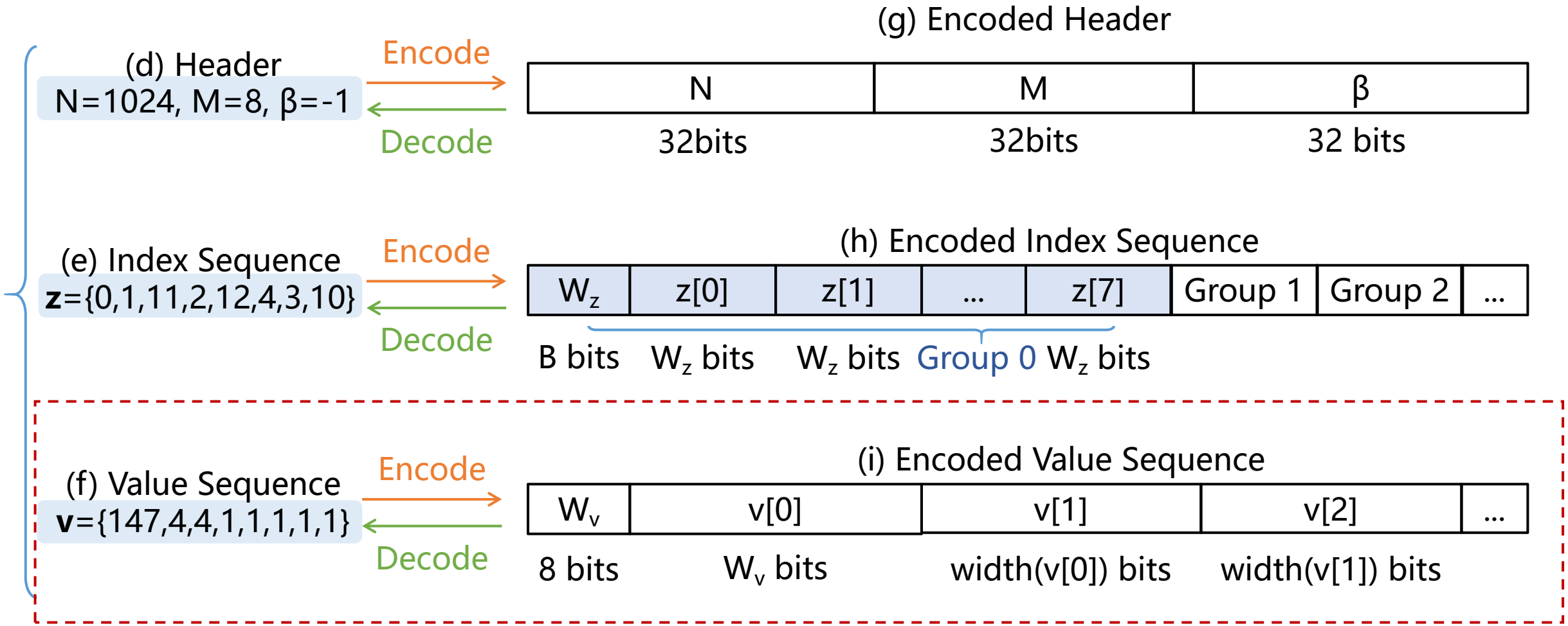
Index Encoding: Fixed-Width Bit-Packing

- Every 8 indexes are grouped together and encoded with W_Z bits. W_Z is the max bit width of all indexes in the group.
  - Because the maximum of index is N-1, W_Z is encoded with $B = \lceil \log_2 \log_2 N \rceil$ bits
- Example:
  - Since the data length is N=1024, we have B=4
  - The max of indexes is z[4]=12, leading to W_Z=4
  - Thus, W_Z=4 is encoded as 0100 with B=4 bits

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



Overview (Encoding & Decoding)

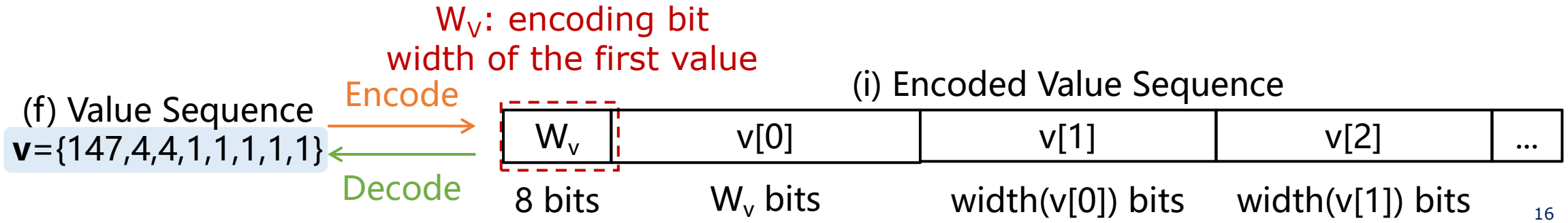


Value Encoding: Descending Bit-Packing

- The encoded bit width of a value is the **valid bit width of the previous value**.
  - **Descending order** => the bit width is never larger than the previous one

- Example:
  - The valid bit width of the first value v[0]=147 is W_V=8
  - v[0]=147 is encoded as 10010011 with W_V=8 bits
  - The valid bits of each value are underlined in the table

i	v[i]	v[i] in binary
0	147	10010011
1	4	00000100
2	4	100
3	1	001
4	1	1
5	1	1
6	1	1
7	1	1



Implementation in Apache IoTDB

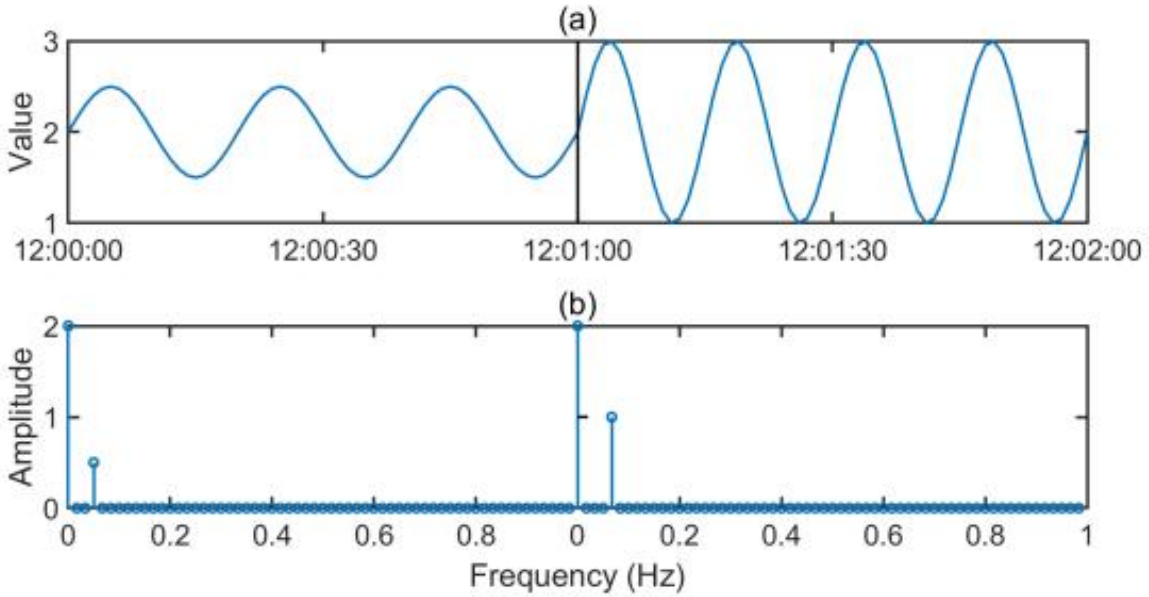
- In Apache IoTDB, frequency domain data has to be stored as **time series**
  - Timestamp column: reuse the timestamps of original time domain data
  - Value column: the amplitude of each frequency component

Timestamp	root.sg1.d1.td	root.sg1.d1.fd
2022-01-01 12:00:00	2.000	2.00
2022-01-01 12:00:01	2.155	0.00
2022-01-01 12:00:02	2.294	0.00
2022-01-01 12:00:03	2.405	0.50
2022-01-01 12:00:04	2.476	0.00
...	...	...
2022-01-01 12:01:01	2.000	2.00
2022-01-01 12:01:02	2.407	0.00
2022-01-01 12:01:03	2.743	0.00
2022-01-01 12:01:04	2.951	0.00
2022-01-01 12:01:05	2.995	1.00
...	...	...

Implementation: SQL

- Directly storing frequency domain data in Apache IoTDB with SQL
  - **create timeseries** root.sg1.d1.fd **with** datatype = double, encoding = descend
  - **select** STFT(td, 'nfft' = '60', 'beta' = '-1') **into** fd **from** root.sg1.d1

Timestamp	root.sg1.d1.td	root.sg1.d1.fd
2022-01-01 12:00:00	2.000	2.00
2022-01-01 12:00:01	2.155	0.00
2022-01-01 12:00:02	2.294	0.00
2022-01-01 12:00:03	2.405	0.50
2022-01-01 12:00:04	2.476	0.00
...	...	...
2022-01-01 12:01:01	2.000	2.00
2022-01-01 12:01:02	2.407	0.00
2022-01-01 12:01:03	2.743	0.00
2022-01-01 12:01:04	2.951	0.00
2022-01-01 12:01:05	2.995	1.00
...	...	...



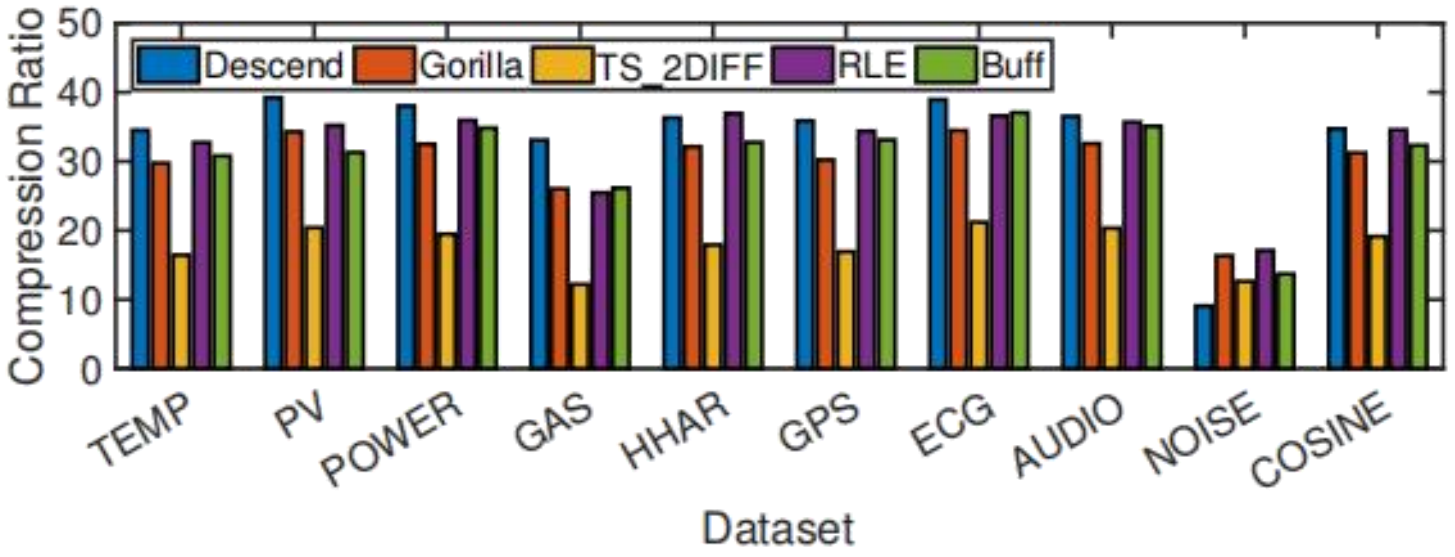
Experimental Setup

- Encoding algorithms:
  - **Descend**
  - Gorilla
  - RLE
  - TS_2DIFF
- Environment:
  - Intel(R) Core(TM) i7-9700 CPU
  - 16GB Memory
  - 64-bit Windows 10 OS

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

Overall Performance

- **Descend** achieves the **highest compression ratio** in most datasets
  - In NOISE, the energy of white noise distributes uniformly on the entire spectrum, violating the assumption of sparsity.

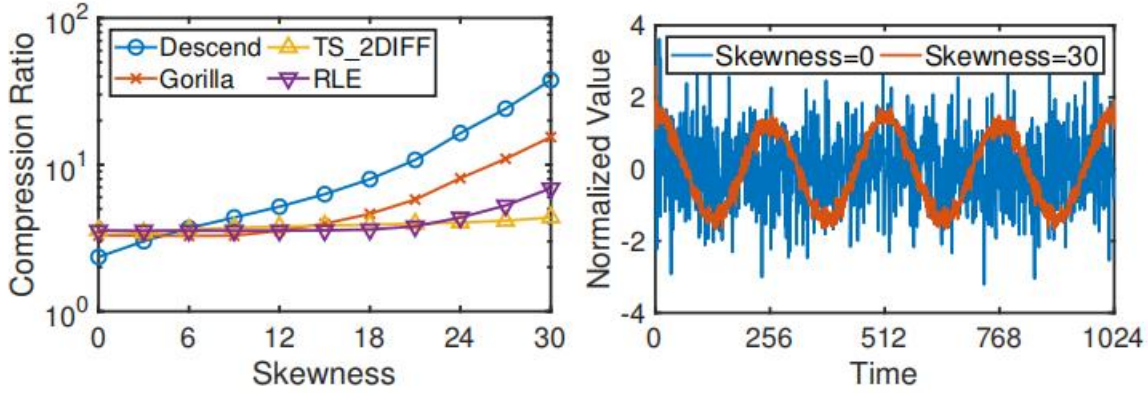


Varying Skewness

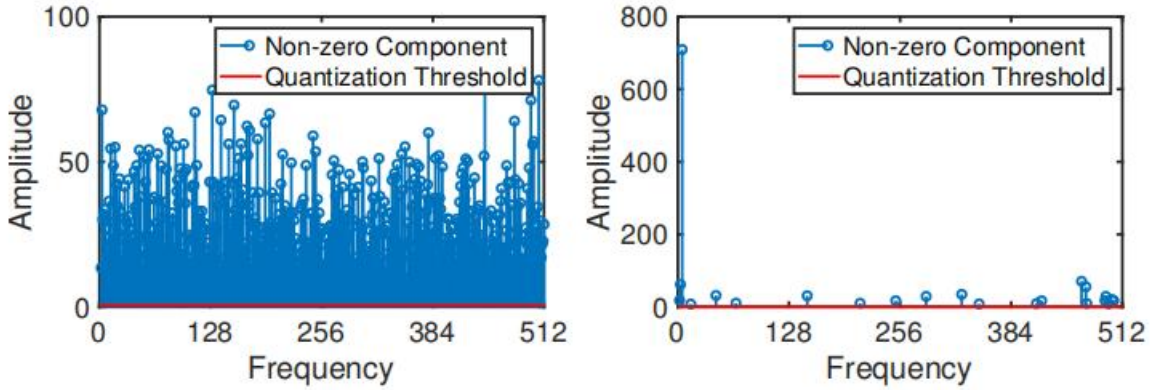
- Skewness

$$\text{Skew} = \frac{\mu_3}{\sigma^3}$$

- Skewness improves the performance of Descend



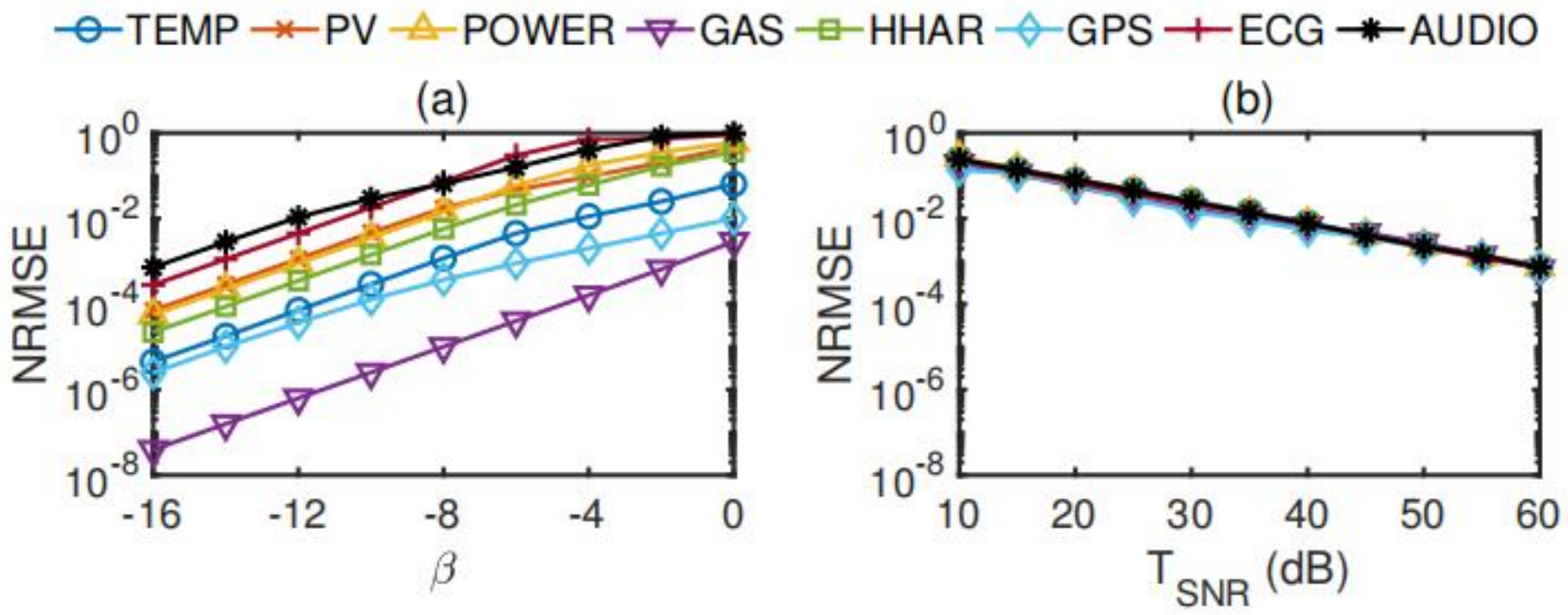
(a) Compression ratio of datasets (b) Time domain of 2 example datasets



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

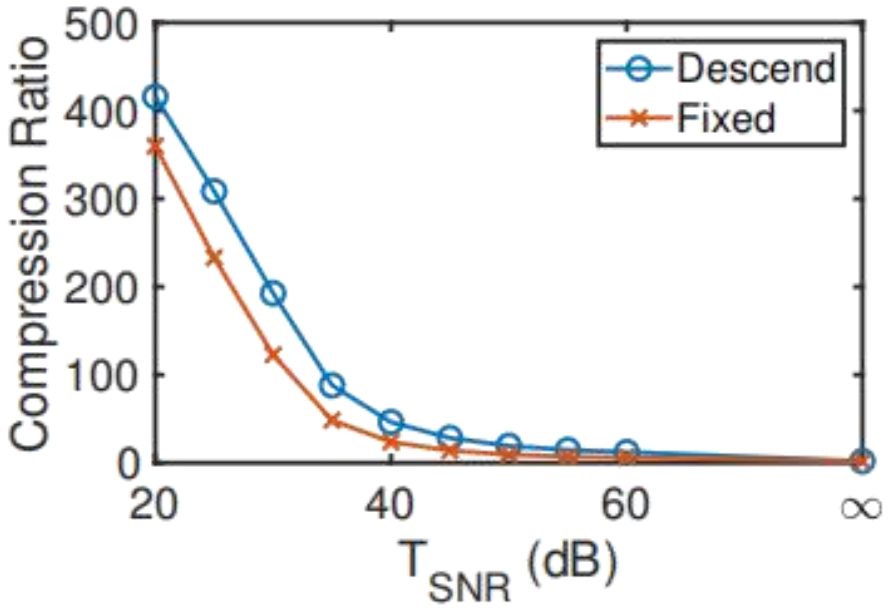
Varying Quantization

- Consistently setting the threshold T_{SNR} for various datasets is much easier than manually specifying β individually for each dataset.

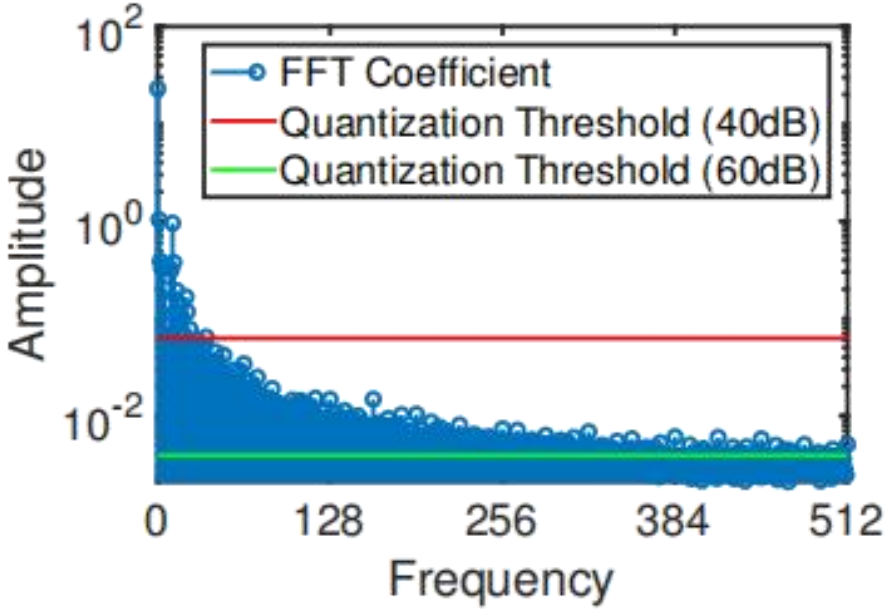


Alternative Options

- **Descending bit encoding combined with the more efficient quantization** leads to a clearly better solution compared to the fixed width bit encoding.



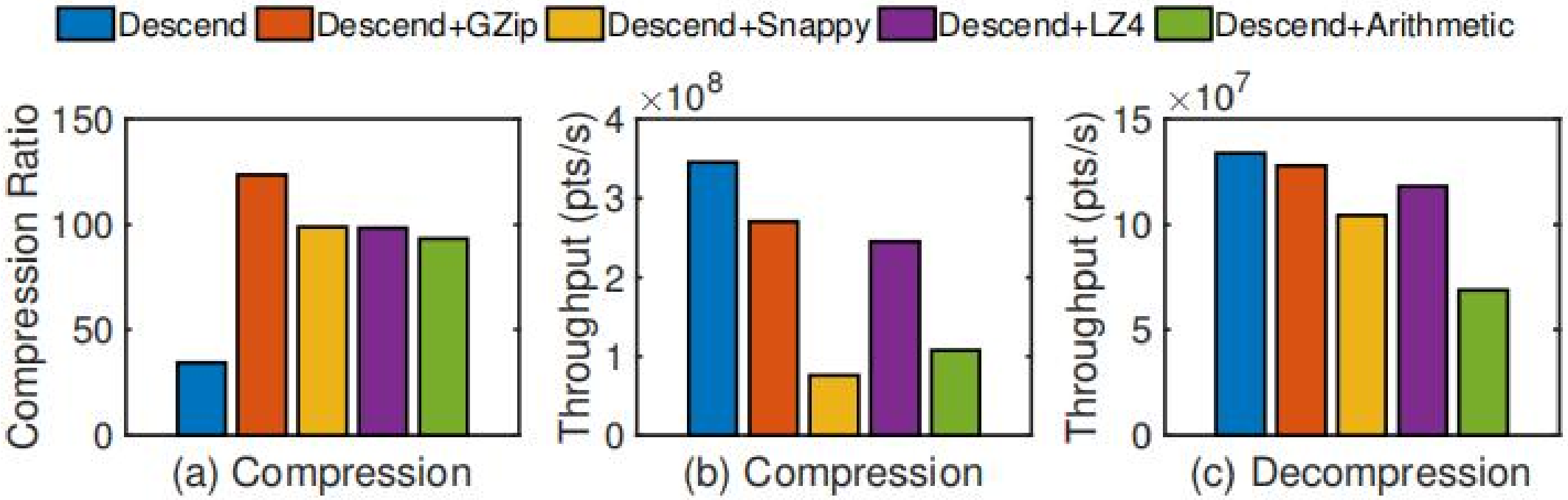
(a) Comparison on compression ratio



(b) Frequency domain

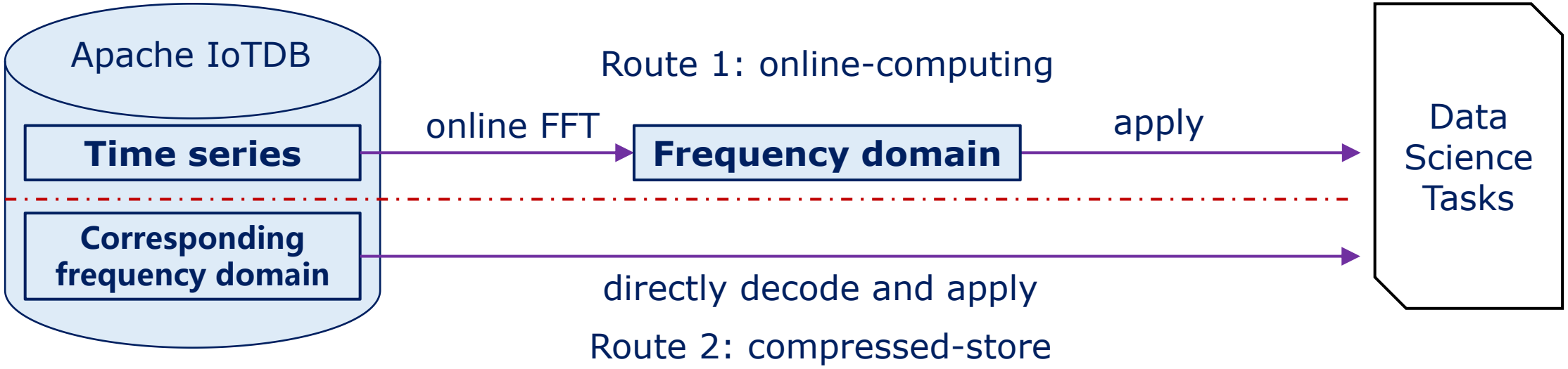
Complement with Compression

- Various compression techniques **further improve the compression ratio** with some extra cost in compression/decompression speed.



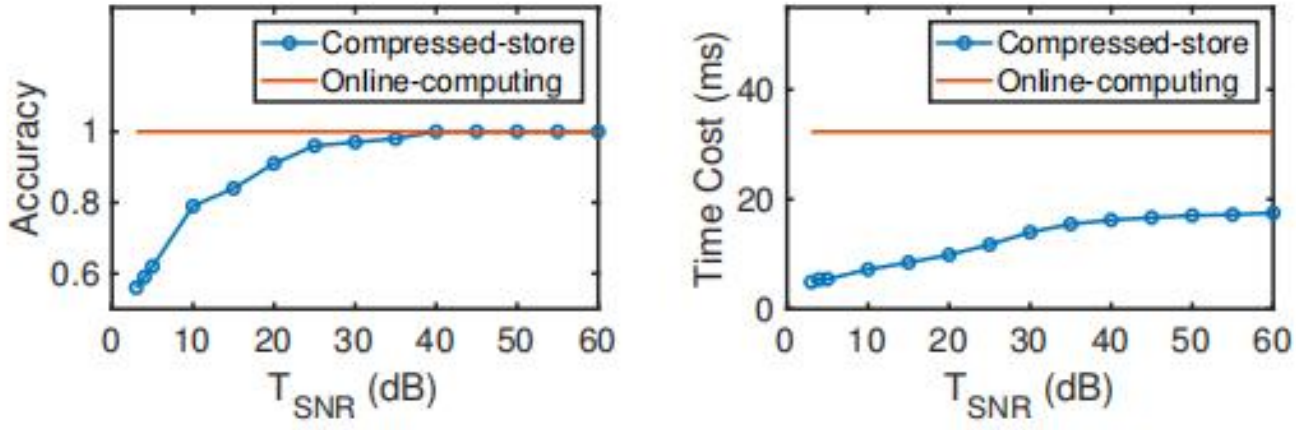
Application in Data Science

- Frequency analysis is widely used in data science tasks of time series
  - Online-computing: FFT online to calculate the frequency domain data for data science tasks
  - Compressed-store: directly decode the pre-stored frequency domain data for data science tasks



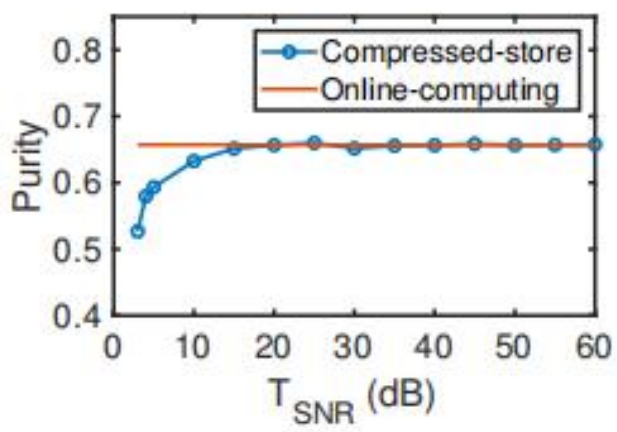
Data Science: Similarity Search

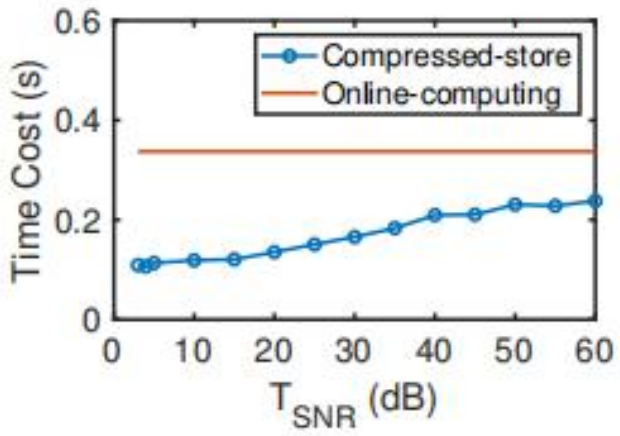
- Similarity search:
  - Compute the **Euclidean distance on the amplitude of Fourier coefficients**
  - Return the nearest neighbor of query
- Frequency domain data encoding with proper quantization can significantly reduce the time cost without losing much accuracy of time series similarity search.



Data Science: Clustering

- Clustering:
  - Compute the **Euclidean distance on the amplitude of Fourier coefficients**
  - Cluster the time series into several clusters
- Frequency domain data encoding significantly improves the efficiency but not loses much effectiveness with proper quantization.

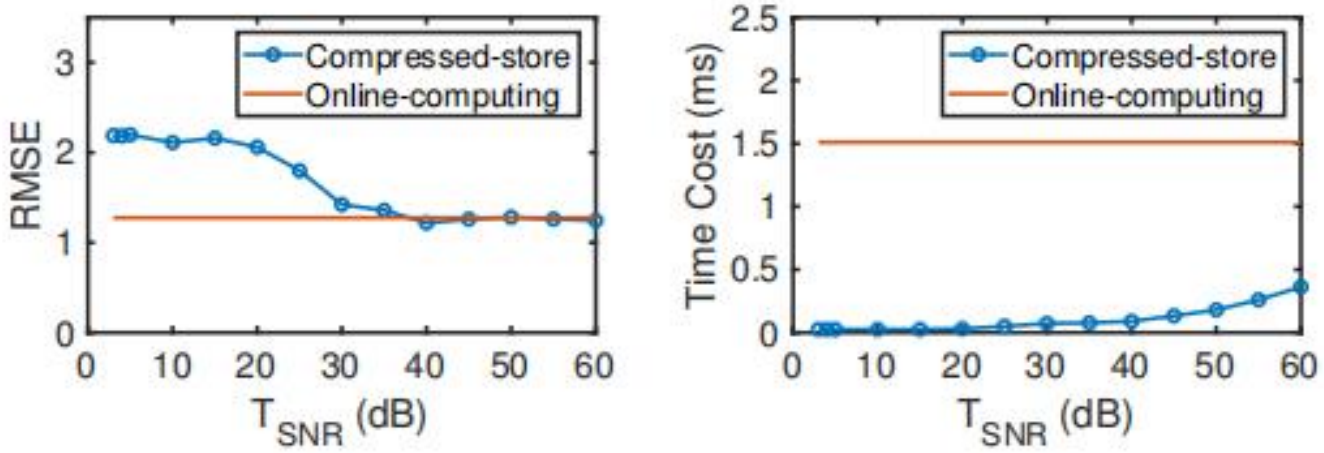




Data Science: Forecasting

- Forecasting:
  - Build **ARIMA models on Fourier coefficients** to forecast those in the next period and thus the time domain

- With proper quantization parameters, compressed-store has almost the same forecasting RMSE as online-computing, but with a much lower time cost.



Conclusion

- 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.