DDC: Efficient Dynamic-Dictionary-Based Compression on Floating Time Series Data

Keyue Yang[1], Dongping Wang[2], Shijie Li[1], Wenbin Zhai[3], Wenjing Wang[1], Ziyi Zheng[1], Jinan Wang[1], Liang Liu[1,*],
[1]College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China
Emails: {kexpepatm, nbvfgh, wangwenjing, zh_zyi,wangjinan}@nuaa.edu.cn
[2]Nanjing FiberHome StarrySky Communication Development, Nanjing, China
Emails: wangdongping870@fiberhome.com
[3]The Department of Computing, The Hong Kong Polytechnic University, Hong Kong
Emails: wenbin.zhai@connect.polyu.hk
*Corresponding author: Liang Liu (email: liangliu@nuaa.edu.cn)

***Abstract*—Growing demand for large-scale time-series data storage has rendered floating-point time-series compression a key technology for better storage efficiency and performance. General-purpose compression rely on exact matching to build dictionaries, making them effective for repetitive data but unsuitable for time-series data with small fluctuations and gradual changes. While time-series-specialized compression leverage correlations between adjacent data points, they ignore long-term similarities and periodic patterns. To address these limitations, we propose an efficient Dynamic-Dictionary-Based Compression (DDC) algorithm for floating-point time-series data. DDC uses three matching strategies to capture both local patterns and global trends. It dynamically updates the dictionary and adapts its compression strategy during processing. Moreover, DDC reduces precision redundancy by adjusting mantissa bit lengths and separating floating-point components. We evaluate DDC against state-of-the-art time-series-specific (Gorilla, Chimp, AFC, TSXor, FPC) and general-purpose (LZ4, LZW, Snappy) compression algorithms. Experimental results demonstrate that DDC achieves superior compression ratios and faster processing speeds, highlighting its effectiveness and practicality.**

***Index Terms*—Time series data, lossless compression, dictionary compression, floating point compression**

I. Introduction

With the rapid development of IoT, smart devices, cloud computing and big data, time-series data has been widely used in fields like temperature sensors, CPU usage, API call latency tracking and financial quantitative trading. Its generation and accumulation scale has reached unprecedented levels. Characterized by massiveness, diversity and strict real-time requirements, time-series data poses significant challenges to storage, processing and analysis, making efficient management essential for many industries.

The increasing demand for computational resources and storage capacity has led to the development of Time-Series Management Systems (TSMS) [1]. TSMS can efficiently collect, store, and analyze data, enabling real-time detection, diagnosis, and analysis [1]. As the volume of time-series data continues to grow, TSMSs rely on compression algorithms to reduce data size and improve disk utilization [2] [3]. However, data compression introduces additional computational and memory overhead. Therefore, efficient compression algorithm is crucial to balance performance and resource usage [4] [5].

Existing time-series compression techniques fall into two categories: general-purpose algorithms and time-series-specific ones. General-purpose algorithms handle diverse data types, reducing redundancy via dictionaries of frequent patterns and relying on exact matching [6] [7] [8] [9] [10]. Though effective for repetitive data, they perform poorly on floating-point time-series data—where slight numerical variations reduce exact matches—failing to capture fine-grained differences and achieving low compression efficiency.

Considering the uniqueness of time-series data [11] [12], existing specialized compression algorithms [3] [4] [5] [13] [14] optimize compression by leveraging such characteristics. Neighboring samples typically share identical high bits with minor low-bit fluctuations; to reduce storage overhead, these algorithms use XOR to extract differential bit segments, recording lengths of leading/trailing zeros and intermediate bits. However, they focus mainly on short-term similarities within limited time windows [14], underutilizing long-term similarities, and cannot dynamically adjust strategies to data characteristic changes, resulting in unstable compression performance across datasets.

To address these challenges, we propose DDC, a dynamic dictionary-based floating-point time-series compression algorithm. DDC employs a similarity-matching dynamic dictionary capturing short-term within-window and long-term global similarities, extending beyond existing algorithms' focus on neighboring point similarities to identify sequence segment similarities. It integrates three similarity-matching strategies with trailing bit similarity for dictionary matching, further compressing entries matching dictionary items and covering similar data ranges. Additionally, DDC dynamically adjusts stored trailing bits by data precision requirements to reduce redundancy. Extensive experiments on synthetic and real-world datasets show DDC outperforms state-of-the-art methods in compression ratio, balances storage cost and de/compression speed, and enhances time-series processing performance.

Our primary contributions include the following:

- We propose a precision-driven optimization for floating-point trailing bits and a similarity identification method for time-series data. This approach efficiently identifies approximate time-series segments, reducing storage cost.
- We design DDC, a novel dynamic dictionary compression framework. It integrates three strategies to dynamically build and update dictionary entries, enabling efficient approximate matching, which reduces redundant information and significantly improves the compression ratio.
- We implemented DDC in the open-source database VictoriaMetrics [15] and conducted experiments to validate its performance. The results demonstrate that DDC outperforms Chimp [13] by 3.76×, LZ4 [9] by 2.54×, and LZW [8] by 2.02× in terms of compression ratio.

The rest of this paper is organized as follows. In Section II, we provide an overview of related work. We provide the preliminaries in Section III. In Section IV, we present our proposed algorithm. Section V presents the experimental results. We conclude the paper in Section VI.

II. Related Work

Time-series compression can be broadly categorized into general-purpose and specialized methods, reflecting the trade-off between wide applicability and specialized efficiency.

A. General dictionary compression algorithm

The Lempel-Ziv (LZ) series are classic lossless compression algorithms, whose core principle involves constructing a dictionary during linear data scanning and replacing repeated substrings with dictionary indexes via greedy matching. LZ77 uses a sliding window for real-time matching, suitable for streaming data compression [6]. LZ78 optimizes dictionary structure with a prefix tree, achieving higher compression ratios in specific applications [7]. LZW initializes the dictionary with all basic characters (e.g., ASCII) and dynamically expands it, compressing by replacing sequences with the longest matching prefix [8]. Snappy combines LZ77 with entropy encoding (e.g., Huffman coding), using LZ77 to find repeated patterns before further compression via entropy encoding [16]. LZ4, based on LZ77, employs a hash table to accelerate matching, supports adjustable compression levels to balance speed and ratio, and compresses by replacing repeated patterns with smaller references [9].

B. Specialized compression algorithm

The FPC algorithm employs two context-based predictors—Fixed Context Model and Differential Finite Context Model—to sequentially predict incoming values in a streaming fashion. It selects the more accurate prediction from the two, achieving high accuracy for smooth data variations. However, its performance degrades when handling abrupt changes [4].

Gorilla uses binary tuple encoding to represent differences between consecutive values, compressing them by recording leading and trailing zeros using variable-length encoding [3]. However, its fixed zero-encoding strategy limits efficiency. Chimp improves upon Gorilla by optimizing the trailing-zero encoding scheme, redesigning control bits, and leveraging multiple historical values to find better matches, thereby achieving higher compression ratios [13].

TSXor introduces a window buffer that exploits redundancy among adjacent data points. It records matching positions, when no match is found, encodes values with more leading/trailing zeros after XOR with the current value [5]. In contrast, AFC extends XOR comparison from only the previous value to all values within the window. It also introduces four adaptive strategies to dynamically select the optimal compression method based on data patterns [14].

III. Preliminary

In this section, we provide some definitions in the paper.

Definition 1 (Unit in the Last Place, ULP): ULP is the distance between adjacent representable numbers in a floating-point system, indicating the precision of the current value [30]. In IEEE 754 [30] double-precision format:

$$ULP = 2^{\text{exp}-52} \tag{1}$$

where exp is the exponent, 52 the mantissa bits.

Definition 2 (Maximum Rounding Error, Δx_{max}): The maximum rounding error is the largest deviation when a value is rounded to the nearest representable floating-point number, an important measure of floating-point precision.

$$\Delta x_{\text{max}} = \frac{ULP}{2} = 2^{\text{exp}-53} \tag{2}$$

Definition 3 (XOR Hash): XOR hash uses bitwise XOR across fixed bits of v_i to generate a fixed-size hash value. The XOR operation mixes input bits, producing a hash with good distribution properties.

IV. DDC

The key challenge in compressing floating point data is cutting redundancy while preserving precision. However, existing algorithms often focus on exact matching or short-term similarity, failing to fully consider long-term patterns in the data. To solve these issues, we propose DDC, a dynamic dictionary-based compression algorithm for floating point time series data. By dynamically adjusting the dictionary and compression strategy based on data characteristics, DDC improves compression efficiency and reduces storage overhead.

Fig. 1 shows the overall architecture of the DDC algorithm. The splitting module (§IV.A) divides each floating-point number into integer and fractional parts. For the integer part, optimized Adaptive-delta compression (§IV.B) eliminates redundancy. For the fractional part, precision-driven dynamic digit selecting (§IV.C) retains significant mantissa bits—removing unnecessary bits while maintaining accuracy and optimizing storage. To leverage global and local similarity in fractional mantissas, a tailing-bit-based similarity detection method (§IV.D) is introduced. Additionally, three adaptive dictionary construction strategies (§IV.E) identify frequent exact/similar

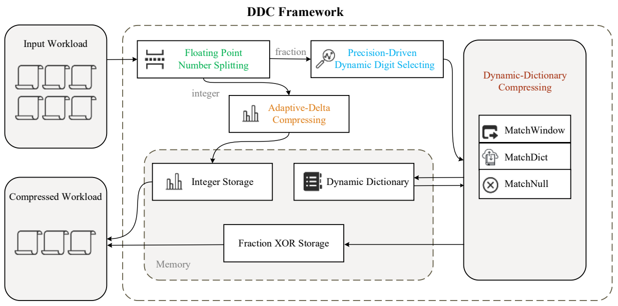

Figure 1. Architecture of DDC

values and create corresponding entries, dynamically building and matching compression patterns based on input data characteristics to improve dictionary match rates and utilization. With these modules, DDC achieves higher compression efficiency and ratios while ensuring data accuracy, and shows strong adaptability and generalizability across datasets.

A. Floating Point Number Splitting

Existing floating-point time-series compression methods usually encode the integer and fractional parts as a single unit. They overlook differences in characteristics and redundancy structures between them. The integer parts of adjacent points often have strong repetition or similarity; in contrast, fractional parts capture fine-grained, high-frequency variations and have a more dispersed distribution. So we separate the integer and fractional parts. Fig. 2 shows the process of decomposing. Using 43.43 as an example, we first convert it to binary per the IEEE 754 double-precision format. Next, we extract the exponent field with a bitmask. The exponent determines the offset to locate the integer part. Here, the exponent is 5, so we extract the first 5 mantissa bits and add the implicit leading '1'—this gives the integer part '101011'. The remaining 47 mantissa bits are the fractional part.

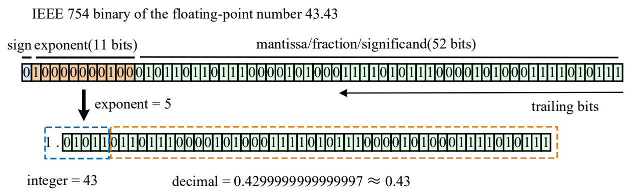

Figure 2. IEEE-754 double-precision floating-point number splitting

B. Adaptive-Delta Compress

In time-series data, the integer part often has strong regularity and slow variation. To exploit this, we propose the Adaptive-Delta, which reduces storage by encoding deltas (differences) between consecutive integers. It dynamically adapts encoding to delta patterns:

- **Zero Delta** : If delta is zero, encode using a single-bit 0, avoiding redundant storage.
- **Same Bit-Length Delta** : If delta shares the same bit length as the previous, use a two-bit 10 followed by the delta value.
- **Variable Bit-Length Delta** : If the delta's bit length changes, use a two-bit 11, followed by the new bit length and delta value.

This hierarchical strategy cuts overhead and adapts to changing delta patterns. For example, integer parts from Table I form the delta sequence: [0,0,0,0,0,0,1,0,0,-1,1,0,-1]. In Adaptive-Delta, the first six use just the 0 bit; delta=1 uses 10 plus value; when bit length changes, 11 is used with new length and value. Dynamically identifying delta patterns, it greatly reduces integer-part redundancy in time-series data.

C. Precision-Driven Dynamic Digit Selecting

Traditional storage with high-bit-width floating-point formats stores full precision, but real-world industrial datasets have effective precision of only 2-9 decimal digits. DDC addresses this by tailoring storage and compression strategies to the actual data precision, enabling the discard of unnecessary fractional bits based on the required accuracy.

Table.I illustrates the IEEE 754 double-precision representation of the example sequence. For time-series data requiring only 2 decimal places, the lower bits of the mantissa provide minimal value, leading to redundancy. DDC splits into integer part and fractional part, and truncates the mantissa as needed. This truncation and zero-filling process lets values be accurately restored within the precision range. Discarding lower mantissa bits greatly improves space efficiency notably in scenarios with lower effective precision requirements.

Discarding mantissa lower bits dynamically retains high-order bits matching target precision, cutting storage redundancy while ensuring accuracy. Theoretical justification: Given double-precision *ULP* and max rounding error $\Delta x_{\max}$, all numbers' representation error stays below $\Delta x_{\max}$ within IEEE 754 double-precision's normalized range. The float's precision depends on significant digits. To minimize rounding errors, discarded precision must be less than the maximum rounding error. Thus, we define a variable threshold for the precision limit when discarding bits.

$$\mathit{threshold} = 5 \times 10^{-(m+1)} \tag{3}$$

In this context, *thbinary* refers to the binary representation of the threshold value. Let ϵ represent the largest value smaller than *thbinary*, where $\epsilon = \mathit{thbinary} - \mathit{ULP}$. Suppose that, for a certain precision, only x bits of the mantissa need to be retained to ensure that the accuracy is not affected, while the remaining $52 - x$ bits can be discarded. The value represented by the x-th bit can be expressed as $2^{(\exp - x)}$.

$$2^{(\exp - x)} < \mathit{threshold} = 5 \times 10^{-(m+1)} \tag{4}$$

Combining the above conditions leads to the inequality:

$$\mathit{thbinary} - 2^{(\exp - x)} \geq \mathit{ULP} \tag{5}$$

TABLE I
BINARY REPRESENTATION OF TIME-SERIES DATA

Value	Value's Binary Representation	Optimization's Binary Representation (mantissa)	Value
43.81	0100000001000101111001111010111000010100011110101110000101001000	01011 11001111000000000000000000000000000000000000	43.809
43.33	0100000001000101101010100011110101110000101000111101011100001010	01011 01010100000000000000000000000000000000000000	43.328
43.55	0100000001000101110001011100001010001111010111000010100011110110	01011 10001011000000000000000000000000000000000000	43.547
43.43	0100000001000101101101110000101000111101011100001010001111010111	01011 01101110000000000000000000000000000000000000	43.429
43.53	0100000001000101110000111101011100001010001111010111000010100100	01011 10000111000000000000000000000000000000000000	43.529
43.34	0100000001000101101010111000010100011110101110000101000111101100	01011 01010111000000000000000000000000000000000000	43.339
43.47	0100000001000101101111000010100011110101110000101000111101011100	01011 01111000000000000000000000000000000000000000	43.469
44.39	0100000001000110001100011110101110000101000111101011100001010010	01100 01100011000000000000000000000000000000000000	44.387
44.55	0100000001000110010001100110011001100110011001100110011001100110	01100 10001100000000000000000000000000000000000000	44.547
44.37	0100000001000110001011110101110000101000111101011100001010001111	01100 01011110000000000000000000000000000000000000	44.367
43.44	0100000001000101101110000101000111101011100001010001111010111000	01011 01110000000000000000000000000000000000000000	43.439
44.45	0100000001000110001110011001100110011001100110011001100110011010	01100 01110011000000000000000000000000000000000000	44.449
44.65	0100000001000110010100110110101110000101000111101011100001010010	01100 10100110000000000000000000000000000000000000	44.648
43.61	0100000001000101110011100100001010001111010111000010100011110110	01011 10011100000000000000000000000000000000000000	43.609

Define *dropbits* to denote the values represented by the number of bits that can be rounded off, so

$$\max(\mathit{dropbits}) = 2^{\exp} - \mathit{ULP} \tag{6}$$

Further derivation yields:

$$\mathit{thbinary} - \max(\mathit{dropbits}) \geq 2 \cdot \mathit{ULP} \tag{7}$$

The final derivation is derived by considering the rounding error case when the *threshold* is represented by a floating point:
Summing up yields

$$\mathit{threshold} - \max(\mathit{dropbits}) \geq 3\,\mathit{ULP}/2. \tag{8}$$

If the value lost due to discarding part of the mantissa, $\max(\mathit{dropbits})$, does not affect the final accuracy of the value, then the inequality

$$r_{\text{binary}} - \max(\mathit{dropbits}) \geq r_{\text{threshold}} \tag{9}$$

holds. Since this inequality always holds, we establish the mapping between effective precision and minimum retained mantissa bits (Fig. 3), which guarantees no accuracy loss when preserving the specified bits.

Precision	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Fraction Bit Numbers	5	8	11	15	18	21	25	28	31	36	37	41	45	48	51	52

Figure 3. Accuracy-driven dynamic digital selection

D. Tailbit-Based Similarity Detection

Existing dictionary compression algorithms build dictionaries for identical, frequent data points. However, they are unsuitable for floating-point time-series data—neighboring points often have slight fluctuations, making exact matching hard.Thus, we propose a new similarity determination method: it measures similarity by comparing the highly significant trailing bits of the fractional part , then builds dictionaries for similar, frequent data points. Specifically, time-series data with consistent or highly similar higher-order bits in the fractional part's effective range is considered numerically similar.This approach effectively captures small data fluctuations, overcomes traditional methods' reliance on exact matching via a structural similarity mechanism, and better preserves underlying data similarity during compression and analysis.

DDC considers varying retained bit lengths across precision levels. Of the fractional part's total *n* bits, the lowest *d* bits are variable, and the remaining *n-d* high bits are base bits. Two data points are similar if their base bits match exactly (variable bits can fluctuate within a range). During compression, DDC adjusts *d* in real time based on similarity matching rate: a low rate means current base bits fail to capture data changes, so *d* is increased, narrowing base bit range and expanding variable bit tolerance. This boosts similarity judgment flexibility, captures more similar points, and via dynamic adjustment, DDC balances stability and flexibility, improving similarity accuracy and compression efficiency.

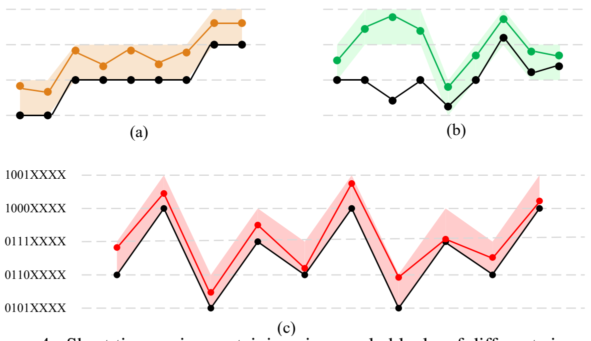

Figure 4. Short time series containing six sample blocks of different sizes and the range of pattern tolerances they represent (shaded portion of the overlay)

The similarity of a time series can be visually illustrated, as shown in Fig. 4. In the figure, the vertical axis indicates whether the base bits of each data point in the time series are consistent. The shaded areas represent the allowable fluctuation range of the variable bits, with the range size being 2^d. When multiple segments of time series data have corresponding data points falling within the same variable bit interval, these segments are considered similar. In panel (a), the two segments of time series data show similarity, while in panel (b), they do not. Panel (c) provides an example of similarity matching using data from Table.I.

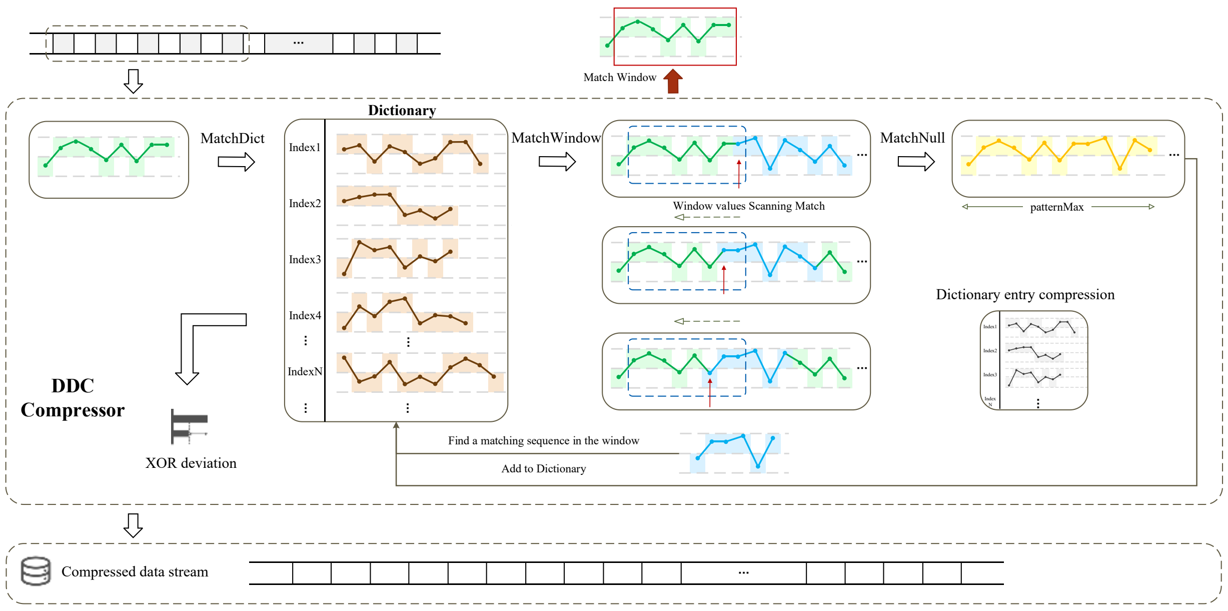

Figure 5. Dynamic dictionary compression algorithm architecture

E. Dynamic-Dictionary Compression

This section details dynamic dictionary compression. Traditional single-strategy compression methods struggle with the heterogeneity of time-series data. When data shows local similarities, efficient algorithms are needed to quickly identify redundant patterns within short time windows. However, when data includes long-term periodic trends or mixed patterns, more flexible dictionary matching mechanisms are required to capture evolving characteristics. To address these, three core strategies are used. These strategies dynamically balance compression efficiency and storage space, ensuring stable performance across different data types and characteristics.

During the dictionary initialization phase, the dictionary is empty. So we propose the ***MatchWindow*** strategy, which uses a local sliding window to discover potential patterns. It searches for the longest similar subsequence within the subsequent data window of the current value. If the subsequence length meets the minimum matching threshold, it is added to the dictionary as a new pattern. This process gradually builds the dictionary and achieves data compression.

Once the dictionary is initialized, DDC uses the ***MatchDict*** strategy to compress the data stream. It calculates XOR hash based on the similarity of trailing valid bits. The hash measures how similar the current value is to dictionary patterns. If a match is found, the pattern index and XOR difference replace the original data. If no, DDC triggers the ***MatchWindow*** strategy. When both ***MatchDict*** and ***MatchWindow*** fail to find a match, DDC activates the ***MatchNull*** strategy. This uses the current data sequence as a new pattern, updating the dictionary and enabling self-expansion of the encoding. The dynamic dictionary compression process is illustrated in Fig. 5.

1) ***MatchDict****:* First, the current time-series is compared for similarity with existing dictionary patterns. DDC generates a hash value with distribution properties by performing a fixed-length bitwise XOR operation on the first value of the sequence. And it‘s encoded as a dictionary index. This ensures that similar data sequences are mapped to adjacent hash regions, enabling fast matching. Under the MatchDict strategy, DDC uses a flag bit ‘0‘ to represent this type of compression entry. It then stores the dictionary index of the matching pattern, the offset of the pattern node, the match length, and the XOR result array of the variable bits between the current sequence and the dictionary pattern. As shown in Fig. 6, the sample segment (01101110, 10000111, 01010111) successfully matches the similar pattern (01100011, 10001101, 01011111, 11010001) in the dictionary.

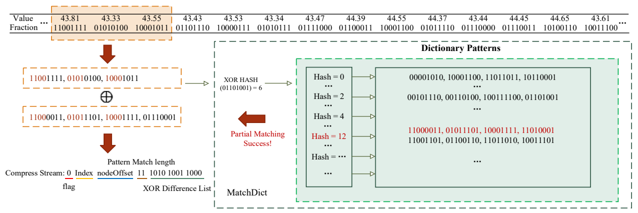

Figure 6. MatchDict strategy

2) ***MatchWindow****:* When the dictionary is empty or no similar patterns are found, DDC uses a fixed-length sliding window W_i to identify potential patterns in the time-series data. For the current value v_i, the window W_i covers the range $(v_{i+1},\ldots,v_{i+W_i})$. DDC performs a reverse scan within the window. It compares v_i with each data point v_s in the window. If $v_i \approx v_s$, v_s is considered a candidate match. Once a candidate match is found, a dual-pointer mechanism is used. We traverses backward from both v_i and v_s, comparing subsequent data pairs. It checks if these pairs meet the similarity condition. If the length of the continuous match exceeds the minimum similarity threshold $Match_{min}$, the subsequence is added to the dictionary as a valid pattern. Otherwise, the matching attempt is deemed unsuccessful, and the search for another similar sequence continues.

This reverse scanning strategy helps capture the longest continuous match early, reducing invalid comparisons and improving space utilization. MatchWindow strategy uses 2-bit '10' marks successful matches, with storage of dictionary index, pattern entry offset, and XOR difference array. This prevents repeated adjacent pattern reads, improving storage and computational efficiency. As shown in Fig. 7, when the sample point '01101110' cannot be directly matched in the dictionary, DDC implicitly triggers the MatchWindow strategy. Similarity comparisons are performed sequentially from back to front in a sliding window. Upon identifying the matching data point '01100011', the algorithm further verifies if the subsequent data points satisfy the constraints for continuous similarity between the matched subsequence and the window pattern sequence. Given that the sequence meets all similarity constraints, DDC adds it as a new pattern to the dictionary.

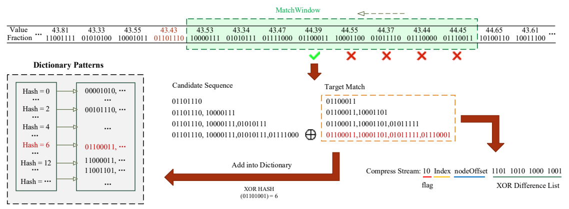

Figure 7. MatchWindow strategy

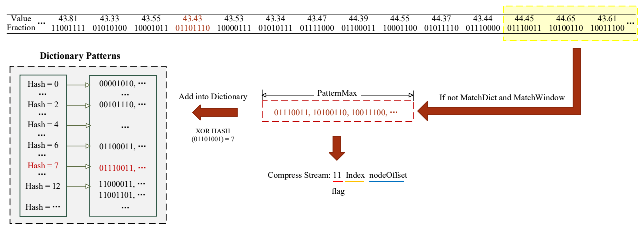
Value … 43.81 43.33 43.55 43.43 43.53 43.34 43.47 44.39 44.55 44.37 43.44 44.45 44.65 43.61 …
Fraction … 11001111 01010100 10001011 01101110 10000111 01010111 01111000 01100011 10001100 01011110 01110000 01110011 10100110 10011100 …
Dictionary Patterns
Hash = 0
Hash = 2
Hash = 4
Hash = 6
Hash = 7
Hash = 12
Hash = …
00001010, …
00101110, …
01100011, …
01110011, …
11000011, …
11001101, …
Add into Dictionary
XOR HASH (01101001) = 7
PatternMax
01110011, 10100110, 10011100, …
If not MatchDict and MatchWindow
Compress Stream: 11 Index nodeOffset
flag

Figure 8. MatchNull strategy

3) ***MatchNull****:* When no similar sequences are found in either dictionary or window, DDC avoids directly storing the raw data value. Instead, starting from the current data point v_i, DDC implicitly generates a new pattern of length $Match_{max}$ and adds it to the dictionary. This ensures the dynamic updating and maintains the validity of the dictionary. To represent this situation, DDC stores a flag '11' and records the corresponding dictionary index along with the node offset.

As shown in Fig. 8, when the data point '01110011' cannot be matched in the dictionary or find a similar match in the subsequent window, DDC implicitly generates a new pattern of length $Match_max$, such as (01110011, 10100110, 10011100, …), and adds it to the dictionary. In this manner, DDC ensures the effective expansion of the dictionary while further improving compression efficiency.

4) Optimization of Dictionary Entry Encoding: In compressing dictionary patterns, DDC uses a space-efficient dynamic storage strategy, storing bitwise XOR results of data points. This dynamic selection enhances storage utilization.

V. Experiments

We implemented DDC and performed performance tests. To comprehensively assess its compression effectiveness and efficiency, we compared it with 8 representative time-series compression methods: specialized algorithms and general-purpose ones. Experiments on real and synthetic datasets validated DDC's advantages in compression ratio and speed.

A. Experimental Setting

In independent algorithm tests, we integrated each compression algorithm into the open-source time-series spatiotemporal database VictoriaMetrics [15]. Benchmark tests ran on Ubuntu 20.04, with hardware: Intel i9-12900H @ 2.50GHz CPU and 16GB RAM. Development used Golang 1.18.0.

B. Datasets

To comprehensively evaluate DDC's performance, we built a heterogeneous test set with 20 public or industry-standard time-series datasets, covering meteorology, finance, sensor data, etc., with effective decimal precisions of 1–17 digits. Their time-series characteristics are summarized in Table.II with detailed descriptions.

TABLE II
Detail of our Datasets

DATASET	SIZE	DECIMAL	SOURCE
City-temp	2905887	1	UDayton
Stock-UK	115146721	1	INFORE
Stock-USA	374428996	2	INFORE
Stock-DE	45403710	3	INFORE
AMPds	1051200	3	Harvard Dataverse
IR-bio-temp	380817839	2	NEON
Wind-dir	199570396	2	NEON
PM10-dust	222911	3	NEON
Dew-point-temp	5413914	3	NEON
Air-pressure	137721453	5	NEON
Stack-ETFs	339496256	4	Kaggle
Stack-Stocks	339496256	4	Kaggle
Basel-wind	124079	7	meteoblue
Basel-temp	124079	9	meteoblue
Bitcoin-price	2741	4	InfluxDB
Bird-migration	17964	5	InfluxDB
Air-sensor	8664	17	InfluxDB
AMMMO_IoT	3057009	12	Alibaba
AMMMO_Server	2605575	18	Alibaba
AMMMO_UCR	967447	0	Alibaba

C. Compression Ratio

DDC outperforms both time-series-specific and general-purpose compressors. Across five large-scale datasets, it achieved a maximum compression ratio of 36.73, with advantages scaling with dataset size. In high-redundancy datasets (e.g., city-temp, basel-temp, air-pressure), DDC delivered 3× and 4.5× higher ratios than Chimp and Gorilla, respectively. For dynamic datasets (stock-DE, bitcoin), it improved ratios by 60–80% over traditional algorithms, demonstrating robustness. On the synthetic air-sensor dataset—characterized by large variations and infrequent repeating patterns—gains were smaller but still notable (10% over Chimp, 6% over FPC), highlighting strong adaptability even in non-typical time-series scenarios.

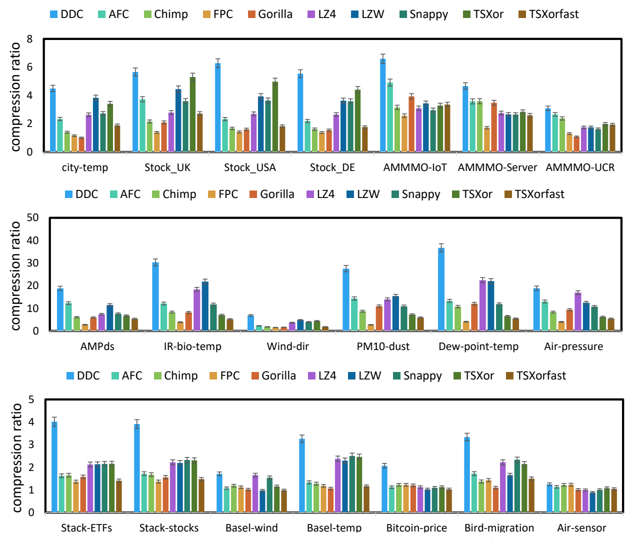

Figure 9. Real-world dataset compression ratio

D. Compression/Decompression Time

Table.III compares DDC's compression time with other algorithms. On the Stock-USA dataset, DDC compresses in 7400ms—slightly longer than AFC and Chimp but notably better than TSXor and LZW. On some datasets, its compression time is comparable to existing time-series algorithms. While AFC, Chimp, FPC, and Gorilla excel in compression speed, they sacrifice substantial compression ratios, particularly for high-redundancy or complex time-series data. Overall, DDC balances compression efficiency and time performance, suiting applications needing both.

Table.III also summarizes decompression times. Results show DDC ranks in the middle tier overall, balancing decompression speed and compression ratio. For instance, in the city-temp dataset, its decompression time (92ms) outperforms AFC, Chimp, and FPC, which typically have longer times. In high-dynamics datasets like Bitcoin-price, DDC exhibits stable performance, with faster decompression than FPC and Chimp—demonstrating efficient handling of highly dynamic data.

E. Compression Ratio and Time Trade-off

Fig. 10 shows the trade-off between compression time and ratio across various algorithms. DDC stands out with significant compression efficiency, reducing dataset storage needs while maintaining speeds comparable to or even faster than general-purpose compression algorithms. It strikes an optimal balance between the two, leading in performance, making it highly competitive for large-scale time-series data processing and applications requiring high compression efficiency.

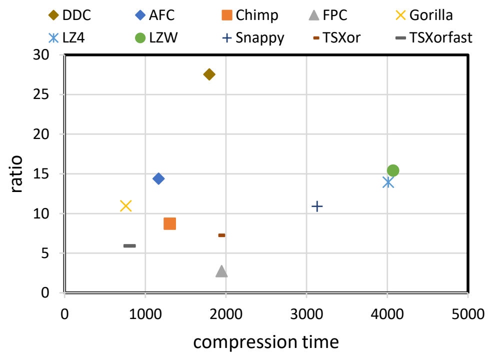

Figure 10. Trade-off between compression time (ms) and compression rate for various algorithms

F. End-to-End Performance

TSDB performance hinges on compression algorithms, with throughput and latency key for write performance. Fig. 11 shows DDC boosts insertion throughput by 30% on average over general-purpose algorithms, sitting between generic and time-series-specialized compressors, near AFC and Chimp. FPC and Gorilla have higher throughput but much higher latency, while DDC consistently achieves low-latency inserts—thanks to its high compression ratio reducing disk I/O. Thus, DDC has the lowest insertion latency and best overall end-to-end performance among evaluated algorithms in TSDBs.

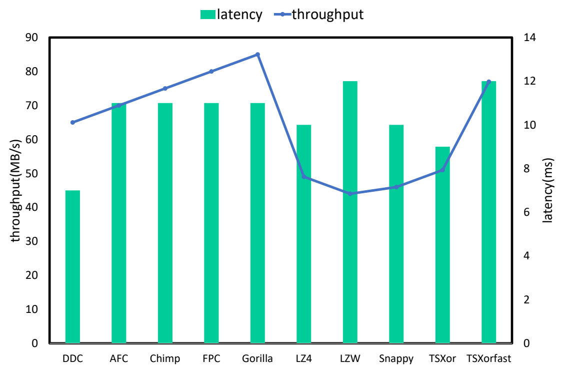

Figure 11. Throughput and Latency of DDC in supporting different databases

VI. CONCLUSIONS

We propose DDC, a dynamic dictionary compression algorithm for floating-point time-series data. It synergistically combines XOR similarity analysis with general compression principles to eliminate redundancy. Key innovations include: 1) Dynamic mantissa bit allocation leveraging data precision

TABLE III
COMPRESSION TIME(MS)/DECOMPRESSION TIME(MS)

	DDC	**AFC**	**Chimp**	**FPC**	**Gorilla**	**LZ4**	**LZW**	**Snappy**	**TSXor**	**TSXor fast**
city-temp	306\92	184\104	132\115	93\97	70\113	47\38	158\117	21\43	502\33	118\41
Stock_UK	5539\3784	4631\3086	3317\3745	3366\4414	2935\3344	1947\1781	5214\4058	1624\1858	9622\1195	2822\1238
Stock_USA	7400\4613	5775\5046	4409\5404	4315\5414	4600\5206	2611\2432	7427\6063	1850\2390	11605\1929	3527\1977
Stock_DE	1157\998	1269\1078	1003\1122	1017\1150	1137\1034	613\521	1766\1442	409\540	2539\453	802\483
AMPds	6500\5935	4199\3092	3953\4527	6463\6185	2928\3716	3460\3417	12510\8590	1624\3619	8957\2322	3033\1866
IR-bio-temp	185251\188663	115757\142408	92007\144001	210318\207223	85046\117781	123773\120187	472476\382187	123254\162973	264706\98098	87581\59926
Wind-dir	349303\202318	243341\222710	219520\254108	172733\230006	229721\221107	165917\117307	305184\265374	170745\133164	558269\83937	159060\94967
PM10-dust	1792\1480	1163\1189	1304\1204	1945\1973	759\714	4010\1421	4070\3353	3128\2224	1912\880	806\564
Dew-point-temp	18314\24475	10287\16092	7948\18109	26465\33557	7059\12899	14997\17291	39628\36358	16404\15501	31062\11037	9184\7740
Stack-ETFs	1668\1050	1494\1259	957\1021	836\1302	1040\978	2120\823	2386\1629	2106\985	4221\432	805\381
Stack-stocks	9397\6322	7609\6700	4942\5842	4735\6814	5675\5951	12189\4192	12454\8964	11205\5757	19953\2289	4179\2112
Air-pressure	70992\65913	37649\49829	32798\54317	80584\82349	27258\45118	55426\47707	195399\149877	46719\63246	86694\37897	33699\25075
Basel-wind	37\14	21\13	12\14	10\12	17\17	13\6	27\21	9\6	50\12	7\7
Basel-temp	10\4	9\6	6\4	4\6	3\10	4\2	10\7	2\5	20\4	3\2
Bitcoin-price	0\1	0\0	0\0	0\0	0\0	0\0	0\1	0\0	0\0	0\0
Bird-migration	1\1	1\1	1\1	1\0	1\1	1\0	1\2	0\1	2\0	1\0
Air-sensor	1\0	1\0	0\0	0\1	0\0	0\0	0\0	0\0	1\0	0\0
AMMMO-IoT	104\85	71\60	87\71	75\75	53\57	66\45	170\111	26\58	305\57	57\53
AMMMO-Server	108\73	73\67	73\71	68\84	52\50	54\34	152\110	29\44	288\44	48\28
AMMMO-UCR	41\33	43\35	43\34	33\36	28\35	26\14	66\45	11\15	177\14	23\11

limitations; 2) Time series similarity detection through trailing bit patterns using an adaptive dictionary; 3) Three optimized matching strategies enabling dynamic pattern updates and optimal selection. Experimental evaluations demonstrate DDC achieves superior compression ratios while maintaining speed efficiency. Moreover, when integrated into the VictoriaMetrics time-series database, DDC has significantly improved insertion performance and reduced disk usage.

ACKNOWLEDGMENT

This work was supported by Jiangsu Province's Special Project for Building a Manufacturing Powerhouse, and the Special Program on Industrial Foundation Re-engineering and High-quality Development of Manufacturing Industry by Ministry of Industry and Information Technology (No.DZBM2024-31-0022).

REFERENCES

[1] Jensen S K, Pedersen T B, Thomsen C. Time series management systems: A survey[J]. IEEE Transactions on Knowledge and Data Engineering, 2017, 29(11): 2581-2600.
[2] Ted D, Ellen F. Time Series Databases: New Ways to Store and Access Data[J]. https://www.academia.edu/29891282/Time_Series_Databases_New_Ways_to_Store_and_Access_Data
[3] Pelkonen T, Franklin S, Teller J, et al. Gorilla: A fast, scalable, in-memory time series database[J]. Proceedings of the VLDB Endowment, 2015, 8(12): 1816-1827.
[4] Burtscher M, Ratanaworabhan P. FPC: A high-speed compressor for double-precision floating-point data[J]. IEEE transactions on computers, 2008, 58(1): 18-31.
[5] Bruno A, Nardini F M, Pibiri G E, et al. Tsxor: A simple time series compression algorithm[C]//String Processing and Information Retrieval: 28th International Symposium, SPIRE 2021, Lille, France, October 4–6, 2021, Proceedings 28. Springer International Publishing, 2021: 217-223.
[6] Ziv J, Lempel A. A universal algorithm for sequential data compression[J]. IEEE Transactions on information theory, 1977, 23(3): 337-343.
[7] Ziv J, Lempel A. Compression of individual sequences via variable-rate coding[J]. IEEE transactions on Information Theory, 2003, 24(5): 530-536.
[8] Welch T A. A technique for high-performance data compression[J]. Computer, 1984, 17(06): 8-19.
[9] Y. Collet. 2011. LZ4: Extremely fast compression algorithm. https://lz4.github.io/lz4/
[10] Deutsch P. DEFLATE compressed data format specification version 1.3[R]. 1996.
[11] RB C. STL: A seasonal-trend decomposition procedure based on loess[J]. J Off Stat, 1990, 6: 3-73.
[12] Fulcher B D, Little M A, Jones N S. Highly comparative time-series analysis: the empirical structure of time series and their methods[J]. Journal of the Royal Society Interface, 2013, 10(83): 20130048.
[13] Liakos P, Papakonstantinopoulou K, Kotidis Y. Chimp: efficient lossless floating point compression for time series databases[J]. Proceedings of the VLDB Endowment, 2022, 15(11): 3058-3070.
[14] Chen H, Liu L, Meng J, et al. AFC: An adaptive lossless floating-point compression algorithm in time series database[J]. Information Sciences, 2024, 654: 119847.
[15] VictoriaMetrics. 2022. VictoriaMetrics: simple & reliable monitoring for everyone. https://victoriametrics.com/
[16] Google Inc. 2011. Snappy: A fast compressor/decompressor. https://github.com/google/snappy
[17] City-temp.2020. https://www.kaggle.com/sudalairajkumar/daily-temperature-of-major-cities
[18] Stock Exchange Datasets.2020. https://zenodo.org/record/3886895#.YbM_6ZuxVqu.
[19] meteoblue datasets.2023. https://www.meteoblue.com/en/weather/archive/export/basel_switzerland
[20] InfluxDB 2.0 Sample Data.2025. https://github.com/influxdata/influxdb2-sample-data
[21] National Ecological Observatory Network (NEON). 2021. 2D wind speed and direction (DP1.00001.001). https://doi.org/10.48443/S9YA-ZC81
[22] National Ecological Observatory Network (NEON). 2021. Barometric pressure (DP1.00004.001). https://doi.org/10.48443/RXR7-PP32
[23] National Ecological Observatory Network (NEON). 2021. Dust and particulate size distribution (DP1.00017.001). https://doi.org/10.48443/4E6X-V373
[24] National Ecological Observatory Network (NEON). 2021. IR biological temperature (DP1.00005.001). https://doi.org/10.48443/JNWY-B177
[25] National Ecological Observatory Network (NEON). 2021. Relative humidity above water on-buoy (DP1.20271.001). https://doi.org/10.48443/Z99V-0502
[26] TBD, Tbd, in: TBD, TBD, 2020, p. TBD. (AMMMO)
[27] X. Yu, Y. Peng, F. Li, S. Wang, X. Shen, H. Mai, Y. Xie, Two-level data compression using machine learning in time series database, in: 2020 IEEE 36th International Conference on Data Engineering (ICDE), IEEE, 2020, pp. 1333–1344.(AMPods)
[28] S. Makonin, B. Ellert, I. V. Bajić, F. Popowich, Electricity, water, and natural gas consumption of a residential house in canada from 2012 to 2014, Scientiffc data 3 (1) (2016) 1–12.
[29] Huge stock market dataset. https://www.kaggle.com/borismarjanovic/pricevolume-data-for-all-us-stocks-etfs
[30] Kahan W. IEEE standard 754 for binary floating-point arithmetic[J]. Lecture Notes on the Status of IEEE, 1996, 754(94720-1776): 11.