Improving Time Series Data Compression in Apache IoTDB

Yuxin Tang
Renmin University of China
yuxintang@ruc.edu.cn

Feng Zhang*
Renmin University of China
fengzhang@ruc.edu.cn

Jiawei Guan
Renmin University of China
guanjw@ruc.edu.cn

Yuan Tian
Timecho
yuan.tian@timecho.com

Xiangdong Huang
NERCBDS, EIRI, Tsinghua University
huangxdong@tsinghua.edu.cn

Chen Wang
NERCBDS, EIRI, Tsinghua University
wang_chen@tsinghua.edu.cn

Jianmin Wang
NERCBDS, EIRI, Tsinghua University
jimwang@tsinghua.edu.cn

Xiaoyong Du
Renmin University of China
duyong@ruc.edu.cn

ABSTRACT

Time series data are generated on an unprecedented scale across various domains. Although traditional compression techniques reduce storage costs, they typically require full decompression before querying, leading to increased latency and higher resource consumption. Homomorphic compression (HC), which enables direct computation on the compressed data without decompression, shows the potential for both reduced storage and improved query performance. However, the unique complexities of time series data pose challenges that current HC methods have yet to adequately address. In this paper, we introduce HC theory in the time series domain, transformatively enabling HC to time series database queries. Building on our theory, we develop *CompressIoTDB* – a novel homomorphic compression framework integrated into Apache IoTDB. By leveraging our proposed CompColumn structure, our framework supports a wide range of query operators, including filtering, aggregation, and window-based functions, all while maintaining data in its compressed form. Furthermore, we incorporate system-level optimizations such as late decompression and dynamic auxiliary management to further boost query efficiency. Extensive experiments show that CompressIoTDB significantly enhances query processing for time series data, achieving an average throughput improvement of 53.4% and memory usage reduction of 20%.

PVLDB Reference Format:
Yuxin Tang, Feng Zhang, Jiawei Guan, Yuan Tian, Xiangdong Huang, Chen Wang, Jianmin Wang, and Xiaoyong Du. Improving Time Series Data Compression in Apache IoTDB. PVLDB, 18(10): 3406 - 3420, 2025.
doi:10.14778/3748191.3748204

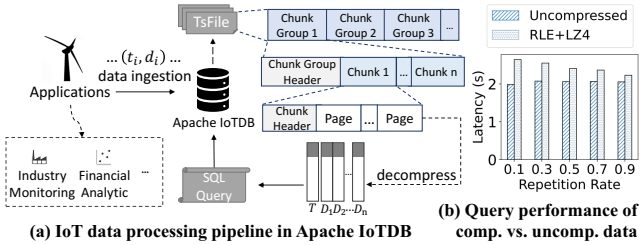
PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/yuxin370/CompressIoTDB.

*Feng Zhang is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 18, No. 10 ISSN 2150-8097.
doi:10.14778/3748191.3748204

1 INTRODUCTION

Time series data are critical in numerous fields, including Internet of Things (IoT), finance [28, 53, 59, 74], healthcare [23, 64, 75, 91], and industrial monitoring [17, 86]. Their chronological structure and exponential growth driven by connected devices and real-time streams necessitate efficient management and querying [14, 24–27, 41–43, 63, 72, 81, 85, 95, 100]. Compression techniques have been used by most time series management systems, such as Apache IoTDB [9], InfluxDB [40], and OpenTSDB [65], to minimize storage costs and transmission bandwidth. However, these compression methods generally require full decompression before querying, leading to significant delays and increased resource consumption, particularly when dealing with large datasets.



(a) IoT data processing pipeline in Apache IoTDB (b) Query performance of comp. vs. uncomp. data

Figure 1: IoT data processing in Apache IoTDB.

Figure 1a illustrates a simplified IoT data processing pipeline in Apache IoTDB, which recently ranked first in the TPCx-IoT benchmark [79]. IoTDB ingests time series data from various IoT applications. These data, featured by high-frequency, redundancy, and regularity, are then compressed using light-weight methods (e.g., RLE) and stored in the TsFile format. For example, railway systems generate 300 billion data points daily, requiring 5TB uncompressed, which is infeasible. TsFile reduces this by up to 95%. However, query execution still requires full decompression, causing significant overhead. As shown in Figure 1b and discussed in § 5.3, while compression cuts disk usage by over 90%, it increases query latency by 15.8% due to decompression costs—highlighting the need for direct querying on compressed data.

Fortunately, homomorphic compression (HC) [36] emerges as a transformative solution, enabling computations directly on compressed text data without decompression. Combining HC with time series data management presents three key advantages: ① It reduces query latency. Since operations can be performed directly on compressed data, the system avoids the overhead of decompression,

significantly speeding up query execution times. ② It improves memory and storage efficiency, and thus enhances system scalability. Keeping data compressed throughout the query process minimizes memory usage, allowing large datasets to be processed without straining system resources. ③ HC offers a unified theory framework for direct computation on compressed data, shifting the paradigm from empirical optimizations to a principled design methodology. Therefore, this paper investigates how to enable HC in time series databases (TSDBs).

Although HC has been well-studied, its application to time series data remains largely unexplored. The unique complexities of time series data – such as high-frequency sampling, large scale, temporal alignment, and timestamp-dependent query patterns [1] – pose challenges that current HC methods do not adequately address. ① Current HC prioritizes general-purpose compression algorithms (e.g., LZW), with limited support for light-weight schemes tailored for time series data. ② Current HC supports only basic operations (e.g., extract). While decompression overhead consumes up to 65.0% of total query time (§ 5.4.2), direct timestamp-dependent queries on compressed time series data are still under-explored. ③ IoT datasets can contain over 90% null values [80] due to misaligned sampling. Current HC, treating data as byte streams, overlook time series complexities like null bitmap management. As a result, there is a pressing need for new solutions to bridge HC and TSDBs. While prior efforts [66, 92, 96] have explored direct computation within streaming systems, they overlook TSDB-specific needs: high compression, inherent data complexity, and specialized query semantics. Moreover, existing approaches lack a unified theoretical model, calling for new TSDB-specific innovations, as detailed in § 2.2.

In this paper, we present a novel HC framework tailored for time series data processing, enabling HC to time series database queries. Building on HC, we introduce a homomorphic query framework for time series data, called *CompressIoTDB*. Our approach, integrated into Apache IoTDB, introduces a compressed data structure, CompColumn, to support direct computation on compressed time series data.By incorporating light-weight compression schemes such as Run-Length Encoding (RLE) [33], Dictionary Encoding [83], and Ts_2Diff [84], it provides robust support for a wide range of query operators, including filtering, aggregation, and window-based functions, all while maintaining the data compressed. We incorporate system-level optimizations to manage auxiliary data structures such as null bitmaps. Together, these techniques significantly improve query performance and system scalability, offering a practical solution for large-scale time series data management.

We evaluate CompressIoTDB on the IoT-benchmark and five real-world datasets with diverse scales and characteristics. Results show a 53.4% average throughput improvement and 20.0% memory reduction, demonstrating the effectiveness of our approach. The main contributions of this paper are as follows: ① We propose a theoretical model specialized for time series data that redefines querying on compressed data and provides formal validation of its performance advantages. ② We design a unified and modular data structure, CompColumn, to manage compressed time series data efficiently. It supports key time series database query operators and ensures seamless interaction with both the storage layer and query engine. ③ We develop CompressIoTDB, a homomorphic compression framework integrated into Apache IoTDB, enabling efficient query execution on compressed time series data.

[1]"Timestamp-dependent query patterns" refer to operations on time series data that explicitly rely on temporal relationships or constraints tied to timestamps, such as timestamp-based joins.

2 MOTIVATION

2.1 Problem Definition

Industrial IoT sensors generate high-volume real-time time series data. In time series data, each data point consists of a timestamp t and a data value d, formally represented as (t, d). A time series S is a sequence of such data points ordered by time, formally defined as $S = \langle (t_1, d_1), (t_2, d_2), \ldots, (t_n, d_n) \rangle$, where t_i is a timestamp and d_i is the associated value, for $1 \leq i \leq n$. For simplicity, we denote the timestamp sequence as T and the value sequence as D. Thus, a time series S can be represented as $S = (T, D)$, where $T = \langle t_1, \ldots, t_n \rangle$ and $D = \langle d_1, \ldots, d_n \rangle$. TSDBs, optimized for this structure, typically separate time series into two columns: a time column T and a value column D, each stored in a compressed format. Based on this structure, we define six basic query operators on time series data [2, 12, 80]. Depending on the operator, computations are performed on time columns T, data columns D, or both.

Filter: Filter operators apply conditions to filter data, either on time column T or value column D.

Timestamp-based join: The timestamp-based join operator merges two time series $S_1 = (T_1, D_1)$, $S_2 = (T_2, D_2)$ by unifying their timestamps: $S = \{(T, D_1, D_2) | T = T_1 \cup T_2\}$, where timestamps from both series align. Unmatched timestamps are preserved with null values for missing data (e.g., $(t_1, d_1, null)$ if $(t_1, d_1) \in S_1$ and $t_1 \notin T_2$).

Aggregation: Aggregation operators apply functions $f : D \rightarrow \mathbb{R}$ to the value column to yield a rational number. Common aggregation functions include `sum`, `average`, `variance`, `max`, `min`, and `count`.

Group by sliding window: This operator groups time series into successive time windows. Formally, $G(S, start, end, window)$ groups the series S into windows between the specified $start$ and end times, with a defined window size $window$. It is often used alongside aggregation functions.

Expression: Expression operators perform computations on value columns, involving both unary and binary operators. Unary operators include `negative`, `regular match`, and `null check`. Binary operators include arithmetic ($+, -, \times, \div$), comparison ($>, <, \geq, \leq, =, \neq$), and logical operations (`and`, `or`).

Slicing: This operator extracts a subset of the time series based on position. It contains two parameters: 1) `offset`, which defines the starting position, and 2) `limit`, which specifies the number of data points to include in the slice.

Queries, involving various operators, require upfront decompression, introducing latency and memory bottlenecks, thus calling for optimized methods for efficient execution on compressed data.

2.2 Revisiting Existing Compression Solutions

Time series compression. Time series compression [4, 6, 15, 38, 44, 55–57, 67, 71, 87], which aims to conserve storage while maintaining crucial data characteristics for effective retrieval and analysis, can be typically categorized as either previous-value-based or model-based. Traditional XOR-based compression algorithms [16, 54, 55, 67, 88], as well as delta-based compression methods [78, 84], typically lack

support for direct querying over compressed data due to sequential dependencies. There are also model-based approaches [18–20, 29, 44, 45, 51, 56, 70, 87, 89] that provide efficient compression, but they are often lossy and may not satisfy the precision needs of industrial applications. We focus on lossless lightweight compression algorithms in this work, which have also been shown to be effective for query-friendly compression in recent studies [5].

Querying compressed data in databases. Efficient querying of compressed time series data employs strategies such as columnar storage for selective decompression [31, 48, 69, 84], index structures with skip pointers [9, 34, 82], and query-aware metadata [77, 85]. Despite these advancements, the decompression overhead is still a bottleneck. Many studies have explored direct computation on compressed data [7, 21, 30, 32, 35, 36, 39, 61, 66, 73, 76, 92–94, 96]. And some studies focus on enabling direct querying of compressed data in columnar databases [3, 49, 68]. Designed for general-purpose columnar databases, these approaches do not prioritize light-weight algorithms suitable for time series compression, and do not take the complexities of time series data organization and timestamp-dependent queries into consideration. In this work, we focus on direct computations for timestamp-dependent operators, with optimizations tailored for time series structures.

Difference from compression in stream systems. Direct querying on compressed data in TSDBs differs significantly from stream processing in three key aspects. ① **System design priority**: Stream frameworks favor light-weight compression for low latency over compression ratio, while TSDBs emphasize higher compression ratio for efficient storage and maintenance. ② **Data scope**: Stream processing frameworks typically operate on small sliding windows (e.g. 512-1024 tuples in each window [96, 97]), whereas TSDBs must handle large-scale historical data requiring bulk data analysis. ③ **Query complexity**: TSDBs are optimized for complex timestamp-based queries (e.g., timestamp-based join), while stream systems focus on simpler queries on real-time streams. These differences necessitate specific design for TSDBs. Our work introduces lightweight schemes, timestamp-dependent operators and TSDB-tailored optimizations with both formal model and practical implementation, bridging the gap between high compression ratios and efficient querying.

2.3 Compressed Time Series Data Direct Processing in Apache IoTDB

Apache IoTDB [9] is a state-of-the-art open-source time series database developed in Java. It operates on the TsFile format [98], optimized for sequential time series data management. It employs a dual-layer compression strategy, combining light-weight algorithms for column encoding with general-purpose heavyweight compression algorithms to minimize the data footprint. It is promising to enable direct processing of compressed time series in IoTDB. However, developing a query framework that supports direct querying of compressed data presents significant challenges, as discussed in Section 1. To address these challenges, we introduce CompressIoTDB, a novel solution designed to facilitate efficient querying over compressed time series data.

Basic Idea. *The basic idea of CompressIoTDB is to enable efficient, direct computation on key time series database operators in compressed time series data queries. By framing direct computation as an algebraic homomorphism problem, we shift the paradigm from empirical optimizations to a mathematically grounded design methodology, providing performance guarantees while supporting a broad range of compression algorithms and TSDB operators.*

Novelties. To address the aforementioned challenges in Section 1, we propose the following novel designs:

- **Theoretical framework for homomorphic compression on time series data (§ 3)**. We present the first formal framework that models the entire query process over compressed time series data. It captures complex time series data semantics and highly timestamp-dependent operators, and provides guidance for compression method selection and system design. Through theoretical proof, we guarantee the validity of homomorphic queries.
- **Homomorphic query framework with modular design (§ 4.2 and 4.3)**. We present a query framework built on HC that enables direct querying of compressed time series data. By introducing a novel CompColumn structure, our approach manages compressed time series data efficiently in memory while supporting a wide range of key time series database operators such as window-based aggregation.
- **System-level optimizations (§ 4.4)**. We introduce system-level optimizations to further enhance query performance. These include dynamic auxiliary encoding for handling nulls and lazy deletions without compromising compression, and late decompression that defers general-purpose decompression until the data is accessed. These optimizations reduce data access overhead, significantly improving query throughput and latency.

3 THEORETICAL FRAMEWORK FOR TIME SERIES DATA

3.1 Definition of Homomorphic Query on Time Series Data

Homomorphic algebra system for time series data. We define the uncompressed and compressed algebra systems as (S_u, Π) and (S_c, Θ), where S_u and S_c represent uncompressed and compressed time series, respectively. The sets $\Pi = \{\mathcal{F}, \mathcal{J}, \mathcal{A}, \mathcal{G}, \mathcal{E}, \mathcal{S}\}$ and $\Theta = \{\mathscr{F}, \mathscr{J}, \mathscr{A}, \mathscr{G}, \mathscr{E}, \mathscr{S}\}$ refer to six basic operators for uncompressed and compressed time series, corresponding one-to-one. Formally, a compression algorithm that satisfies the paradigm of *operating directly on compressed data* can be expressed as a mapping $\varphi : S_u \rightarrow S_c$. If for any $op \in \Pi$ and the corresponding operator $op' \in \Theta$, we have $\varphi^{-1}(op'(c_1, \ldots, c_n)) = op(\varphi^{-1}(c_1), \ldots, \varphi^{-1}(c_n))$, where $c_1, \ldots, c_n \in S_c$. We call φ **homomorphic compression (HC)**.

Time series compression algorithms can be specified as four basic encoding transformations: delta, repeat, bit-packing, and dictionary [22], formally represented as $C = \{T_{delta}, T_{repeat}, T_{pack}, T_{dic}\}$. light-weight time-series-specific compression algorithms typically use these basic components to encode IoT sensor data into encoded bit-array: $\varphi = T_k \circ T_{k-1} \circ \ldots \circ T_1, T_j \in C$. For example, Gorilla [67] is a combination of T_{delta}, T_{repeat}, and T_{pack}. Table 1 shows how

different homomorphic operators interact with each encoding components. Specifically, T_{delta} supports aggregation and expression computations via its telescoping sum property; random access to an arbitrary point, however, is restricted. In contrast, compression methods using T_{repeat}, T_{pack}, and T_{dic} typically allow partial random access or localized computation, which enables direct evaluation of operators in the operator set Π.

Table 1: Operator–encoding component matrix

Operator	T_{delta}	T_{repeat}	T_{pack}	T_{dic}
Filter	–	✓	✓	✓
Timestamp-based Join	–	✓	✓	✓
Aggregation	∗	✓	✓	✓
Group by Sliding Window	–	✓	✓	✓
Expression	∗	✓	✓	✓
Slicing	–	✓	✓	✓

✓: Direct support; ∗: Conditional support; –: Limited or no support.

Process of compressed series querying in TSDBs. Following [36, 37], we represent the distribution of a multi-step process and the probability of an event during the process using the following syntax: $Distribution = \{output : Process\}$. And we use $y \leftarrow f(x)$ to denote y as the output of function $f(x)$. The traditional query process on a compressed time series is denoted as:

$$I_u(c) = \left\{ u : \begin{array}{l} u_0 \leftarrow Decomp(c), u_1 \leftarrow Restore(u_0), \\ u_2 \leftarrow op_1(u_1), \ldots, u \leftarrow op_n(u_{n-1}) \end{array} \middle| \begin{array}{l} i \in \{1, \ldots, n\} \\ op_i \in \Pi \\ c \in S_c \end{array} \right\},$$

where *Decomp* refers to data decompression, and *Restore* refers to the restoration of auxiliary structures (e.g., deletion lists or null bitmaps). This equation indicates that to process compressed time series data, traditional database systems involve three steps: 1) decompressing the compressed data; 2) restoring auxiliary information to decompressed data based on auxiliary structures; and 3) executing queries on the decompressed data. A more concise representation is $I_u(c) = (Q_u \circ R_u \circ U)(c) = Q_u(R_u(U(c)))$, where U represents the decompression process, R_u refers to the auxiliary structure restoration on uncompressed time series data, and Q_u denotes querying on uncompressed time series data.

We define homomorphic query based on the definition of HC.

Definition 3.1 (Homomorphic Query). Given HC $\varphi : S_u \to S_c$, let $Q_u = \langle op_1, \ldots, op_n \rangle$, $op_i \in \Pi$, be a query on S_u, and $Q_c = \langle op'_1, \ldots, op'_n \rangle$, $op'_i \in \Theta$, represents a query on S_c. R_u and R_c refer to auxiliary restoration for S_u and S_c, respectively. If

$$\varphi^{-1}((Q_c \circ R_c)(c_1, \ldots, c_n)) = (Q_u \circ R_u)\left(\varphi^{-1}(c_1), \ldots, \varphi^{-1}(c_n)\right),$$

where $c_1, \ldots, c_n \in S_c$, then Q_c is called a homomorphic query (HQ).

3.2 Properties of Homomorphic Query

While homomorphic query builds a mapping from the operations on compressed time series data to those on uncompressed data, a valid homomorphic query should satisfy the following property:

Definition 3.2 (Direct Homomorphic Query). Given HC mapping $\varphi : S_u \to S_c$, and homomorphic query Q_c, if there involves no computation fragment in the form of $op'(u), op' \in \Theta$, where $u \in S_u$ refers to any uncompressed data segment in the query process, we consider Q_c to be direct.

This definition stipulates that direct homomorphic queries operate exclusively on compressed data, without requiring decompression at any point. For queries that necessitate decompression at certain operators, we call it **Partially Homomorphic Query** (for a formal definition, please see Appendix A). While sacrificing full directness, partial homomorphism balances correctness with reduced decompression overhead for operators that support homomorphic mapping, offering practical performance gains without complexity of full homomorphic.

Definition 3.3 (Effective Restore). Given query Q and HC mapping $\varphi : S_u \to S_c$, let R_u and R_c denote the auxiliary restore processes on uncompressed and compressed time series data such that $\varphi(R_u) = R_c$. If for any $u \in S_u$, the cost of restoring the compressed data, $Cost(R_c(c))$, $c \in S_c$, is less than that of the uncompressed data, $Cost(R_u(u))$, we define R_c as an effective restore.

Definition 3.4 (Effective Homomorphic Query). Given query Q and HC mapping $\varphi : S_u \to S_c$, let query on uncompressed time series data be $Q_u = \langle op_1, \ldots, op_n \rangle$, $op_i \in \Pi$, and the corresponding homomorphic query on compressed time series data be $Q_c = \langle op'_1, \ldots, op'_n \rangle$, $op'_i \in \Theta$, where $\varphi(op_i) = op'_i$, $i \in \{1, \ldots, n\}$. If for any $u \in S_u$, we have $Cost(Q_c(c)) < Cost(Q_u(u))$, where $c = \varphi(u)$, we define Q_c as an effective homomorphic query.

Definition 3.4 highlights that effective homomorphic queries reduce operation overhead. While homomorphic queries are expected to achieve performance improvement through reduced I/O overhead, memory usage, and redundant computations, some compression algorithms may introduce additional overhead due to the need for maintaining extra data structures, making direct queries on compressed time series data slower than those on uncompressed data. Such query is considered ineffective and is referred to as an **Ineffective Homomorphic Query**.

Lemma 3.5. *Given time series S_u, for any query:*

$$Q_u(u_0) = \left\{ u_n : \begin{array}{l} u_1 \leftarrow op_1(u_0), u_2 \leftarrow op_2(u_1), \\ \ldots, u_n \leftarrow op_n(u_{n-1}) \end{array} \middle| \begin{array}{l} i \in \{0, \ldots, n\} \\ op_i \in \Pi \\ u_i \in S_u \end{array} \right\},$$

we have $Size(u_0) \geq Size(u_1) \geq \ldots \geq Size(u_{n-1}) \geq Size(u_n)$, where $Size(u)$ denotes the size (in bytes) of time series u.

Proof. Let $\Pi = \{\mathcal{F}, \mathcal{J}, \mathcal{A}, \mathcal{G}, \mathcal{E}, \mathcal{S}\}$ denote the operator set. For time series $u \in S_u$, we analyze each operator: 1) Filter ($\mathcal{F}$). A filter $\mathcal{F}(u) = u'$ removes data points violating condition. Since $u' \subseteq u$, $Size(u') \leq Size(u)$. 2) Timestamp-based join ($\mathcal{J}$). For $u_1 = (T_1, D_1)$ and $u_2 = (T_2, D_2)$, $\mathcal{J}(u_1, u_2) = \{(T, D_1, D_2) \mid T = T_1 \cup T_2\}$. Since $T_1 \cup T_2 \subseteq T_1 + T_2$ (timestamps may overlap), $Size(S) \leq Size(S_1) + Size(S_2)$. 3) Aggregation ($\mathcal{A}$). An aggregation $\mathcal{A}(u) = d \in \mathbb{R}$ maps a series to a scalar. As d requires constant storage, we have $Size(d) \leq Size(u)$. 4) Group by sliding window ($\mathcal{G}$). Let $G(S, start, end, window)$ partition u into subseries $\{u_1, \ldots, u_m\}$. If followed by aggregation (e.g., $\mathcal{A}(u_i)$), each group reduces to a scalar: $Size(\{\mathcal{A}(u_1), \ldots, \mathcal{A}(u_m)\}) \leq Size(u)$. If no aggregation, $Size(G(u)) = Size(u)$. 5) Expression ($\mathcal{E}$). Expression operators transform values without altering timestamps. Since no new data points are added and value transformations preserve cardinality, we have $Size(u') = Size(u)$. 6) Slicing ($\mathcal{S}$). $\mathcal{S}(u, offset, limit) = u'$ extracts a contiguous subset. As $u' \subseteq u$, $Size(u') \leq Size(u)$. Since

for any operator sequence $op_1, \ldots, op_n \in \Pi$, each step satisfies $Size(u_k) \leq Size(u_{k-1})$. By reduction,

$$Size(u_0) \geq Size(u_1) \geq \ldots \geq Size(u_n).$$

□

Lemma 3.5 indicates that the total data size decreases monotonically as query operations are performed, which is a common characteristic in real-world queries.

Proposition 3.6. *Given compressed time series S_c, an auxiliary restore process R_u and a query $Q_u = \langle op_1, op_2, \ldots, op_n \rangle$, $op_i \in \Pi$ on uncompressed time series, along with an effective auxiliary restore R_c and an effective homomorphic query $Q_c = \langle op'_1, op'_2, \ldots, op'_m \rangle$, $op'_i \in \Theta$ on compressed time series, there exists a mapping $\varphi : S_u \rightarrow S_c$ such that $\varphi^{-1}((Q_c \circ R_c)(c)) = (Q_u \circ R_u)(\varphi^{-1}(c))$. we have $Cost(I_u(c)) \geq Cost((I_c(c)))$, where $I_u(c) = (Q_u \circ R_u \circ U)(c)$ and $I_c(c) = (I'_u \circ Q_c \circ R_c)(c)$, $c \in S_c$.*

Proof. Given $Cost(I_u(c))$ and $Cost(I_c(c))$, we have

$$\begin{aligned}
&\text{Cost}(I_u(c)) - \text{Cost}(I_c(c)) \\
&= (\text{Cost}(U(c)) - \text{Cost}(U(c_j))) + (\text{Cost}(R_u(u)) - \text{Cost}(R_c(c))) \\
&+ \left(\text{Cost}\left(\langle op_1, op_2, \ldots, op_j \rangle(u_0)\right) - \text{Cost}\left(\langle op'_1, op'_2, \ldots, op'_j \rangle(c_0)\right)\right).
\end{aligned}$$

According to Lemma 3.5, we have $Size(c) \geq Size(c_j)$, which implies that $Cost(U(c)) \geq Cost(U(c_j))$. Furthermore, by Definition 3.4, we obtain the inequality $Cost\left(\langle op_1, op_2, \ldots, op_j \rangle(u_0)\right) \geq Cost\left(\langle op'_1, op'_2, \ldots, op'_j \rangle(c_0)\right)$. By Definition 3.3, it follows that $Cost(R_u(u)) \geq Cost(R_c(c))$. Thus, we conclude that

$$Cost(I_u(c)) \geq Cost(I_c(c)).$$

□

Due to space constraints, a concise proof is provided here, with the details presented in Appendix C. Proposition 3.6 identifies three cost-determining factors for homomorphic and traditional queries: **time series data decompression**, **auxiliary restoration**, and **operator computation**. Although partial homomorphic queries cannot avoid time series data decompression, they delay decompression until necessary, reducing the decompressed data sizes. This minimizes overhead, enabling effective homomorphic queries to outperform traditional ones. Even ineffective homomorphic queries show significant potential by offsetting operational costs through reduced decompression and I/O overhead.

4 COMPRESSIOTDB

4.1 Overview of System Modules

Guided by the theoretical framework established in Section 3, we propose a novel solution, called CompressIoTDB. It consists of three modules: **Data Structure Module**, **Operator Module**, and **Optimization Module**, as shown in Figure 2. These modules enable direct computation on compressed time series data within IoTDB's query layer. The data structure module serves as the system's foundation, providing essential structures that ensure *directness* (Definition 3.2) for the operator and optimization modules. The operator module implements *effective homomorphic queries* (Definition 3.4) by facilitating direct computation on compressed time series data for key time series operators, as listed in § 2.1. The optimization module enhances the overall performance by accelerating the construction of compressed time series data structures and data transmission via *effective restore* (Definition 3.3). For implementation details, please refer to Appendix D.

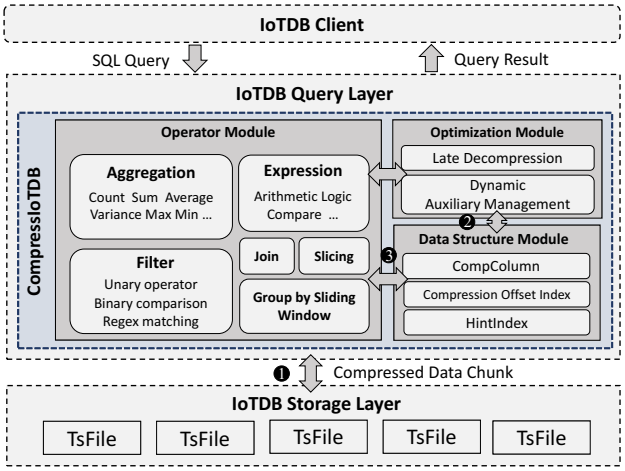


Figure 2: Framework of CompressIoTDB

Workflow. The workflow of CompressIoTDB proceeds in three stages, as shown in Figure 2. ❶ CompressIoTDB loads compressed time series data chunks from the storage layer into the chunk cache. ❷ It retrieves compressed time series data from the chunk cache and constructs CompColumn in two phases: 1) *late decompression for TsFile* (§ 4.4.2), which defers general-purpose decompression until the data is actually accessed; 2) *dynamic auxiliary management* (§ 4.4.1), which avoids light-weight decompression and uses a dynamic encoding strategy to restore auxiliary in a compact form. ❸ It executes homomorphic operators directly on the compressed time series data stored in CompColumn (§ 4.3), leveraging the *Compression Offset Index* (§ 4.2.2) and *HintIndex* (§ 4.2.3) for efficient data access. Intermediate results are passed between operators as CompColumns, and the final query results are returned to the client in uncompressed form.

Compression algorithm selection. We use three different compression algorithms for our homomorphic query framework: 1) the repeat-based RLE algorithm, suitable for a variety of data types, 2) the Dictionary encoding algorithm, commonly used for text data, and 3) the delta-based Ts_2Diff algorithm, suitable for numerical data. In Apache IoTDB, Dictionary encoding combines dictionary-based and run-length encoding by first encoding strings with dictionary and then compressing the result with RLE.

4.2 Data Structure Module

The data structure module provides key structures for storing and accessing compressed time series data, especially the **CompColumn** structure, which ensures the *directness* of CompressIoTDB. The primary goals of this design are to maximize performance while minimizing memory usage.

Design concept. Homomorphic queries improve performance by reducing I/O costs and avoiding decompression overhead. However,

enabling direct computation on compressed data requires query operators to be aware of and adapt to the compression scheme, necessitating extensive code modifications and engine disruptions [1]. To address this, we propose the *CompColumn* data structure, which efficiently manages compressed time series data and provides unified interfaces for operators. These interfaces abstract compression details, enabling direct computation (e.g., writing, slicing, reversing, and (de)serialization) without exposing algorithm-specific logic, ensuring efficient data access and manipulation. To further streamline data access, we incorporate a *Compression Offset Index* and a *HintIndex* within CompColumn.

Listing 1: CompColumn definition in CompressIoTDB

```
class CompColumn implements Column{
public:
    CompColumn(int arrayOffset, int positionCount,
        Column[] values, int[] patternOffsetIndex);
    public ColumnEncoding getEncoding();
    // reading
    Object getObject(int position);
    Pair<Column[], int[]> getCompBlocks();
    // slicing
    Column getRegion(int positionOffset, int length);
    Column subColumn(int fromIndex);
    // reversing
    void reverse();
private:
    Column[] values; // compression block array
    int[] compressionOffsetIndex;
    int hintIndex;
};
class CompColumnBuilder implements ColumnBuilder {
    // writing
    CompColumnBuilder writeCompressionBlock(
        Column value, int logicPositionCount);
    ColumnBuilder write(Column column, int index);
};
class CompColumnEncoder implements ColumnEncoder {
    // deserialization
    Column readColumn(ByteBuffer input,
        TSDataType dataType, int positionCount);
    // serialization
    void writeColumn(DataOutputStream output,
        Column column);
};
```

4.2.1 CompColumn Design. CompColumn is designed as a modular class that inherits from the abstract class `Column`, which provides a unified data representation in Apache IoTDB. As shown in Listing 1, in CompColumn, each compression block is stored as a `Column`, and a compressed time series is constructed as an array of these `Columns`, referred to as `values`. The two key data structures within CompColumn, `compressionOffsetIndex` (hereafter denoted as `coIndex`) and `hintIndex`, are crucial for enabling fast access to compressed time series data. The `getEncoding` function returns the compression algorithm used for the stored values. The `getObject` and `getCompBlocks` functions offer data access at different granularities, either at the individual data point level or at the compression block level. Functions like `getRegion` and `subColumn` allow for slicing operations on CompColumn, and `reverse` reverses the order of data stored in `values`. Two helper classes, `CompColumnBuilder` and `CompColumnEncoder`, handle the writing and (de)serialization processes of CompColumn. CompColumn's modular design enables operators to access both compressed and uncompressed data via unified interfaces, allowing them to focus on computation. This design also allows new compression schemes to be easily integrated by inheriting from the Column class and implementing required interfaces.

4.2.2 Compression Offset Index. A key distinction between compressed and uncompressed data is that uncompressed data supports random access, while compressed data, despite retaining some structural organization, lacks efficient random access capabilities. This often necessitates traversing the entire compressed dataset to retrieve a single tuple, resulting in significant overhead. To achieve fast compressed time series data locating, we implement offset indexing between compressed and uncompressed data.

The compression offset index is structured as a two-tiered mapping. The first tier consists of block-level entries that map uncompressed data blocks to their compressed counterparts. Each block stores a reference to the starting position of the corresponding compressed segment. The second tier contains fine-grained offsets within each block, mapping specific uncompressed tuples to their relative position within the compressed block. This allows for efficient narrowing down of the search scope during data retrieval. During access, we first map through the offset index to the target segments within the compressed data, and then conduct a narrow-scoped search to retrieve the desired tuple or compression block. For example, as shown in Figure 3, to fetch data at position $targetOffset = 18$, we first consult the compression offset index and locate at `coIndex[3]`, with $HintIndex = 3$ (explained further in § 4.2.3). This index points to a compression block spanning from position 17 to 22. Then, we retrieve data from the second compression block, i.e., `Values[3]`. Since the value array is RLE-encoded (where each block represents a run of repeated values), the target value at position 18 is 7.

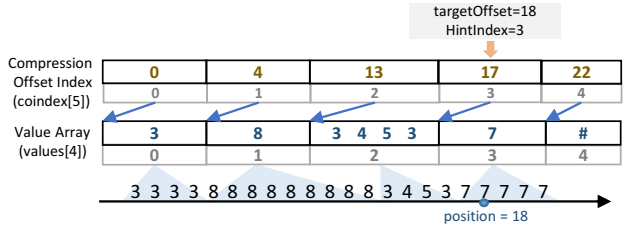


Figure 3: An example of the CompColumn for RLE

4.2.3 HintIndex. In time series database queries, data is often accessed sequentially during scans. To optimize access, CompColumn employs a *HintIndex*, a 4-byte integer that stores the last accessed position in the compression offset index. When retrieving data, CompColumn first checks if the *HintIndex* falls within the segment containing the target data. If it does, CompColumn directly accesses and returns the data. If not, the system determines whether the target data lies beyond the current indexed segment. If so, it continues traversing the compression offset index from the current position and updates the *HintIndex*. Otherwise, traversal begins from the start. This design avoids traversing the index from scratch on each access [47]. For example, when handling large-scale time series data exceeding memory capacity, slicing is necessary. Assume that CompColumn is split into fixed-length sub-columns (e.g., 8,000 points) for memory efficiency. *hintIndex* is initialized to 0. When accessing the first $(0, 8000)$ interval, we directly retrieve the starting position. Upon reaching the 8,000th position, traversal identifies index 100 (assuming `coIndex[100]` maps to the original uncompressed segment 7,800–8,100). We then update the *hintIndex* to 100

and perform compressed data slicing at this boundary. For next requests for (8,001, 16,000), we leverage the *hintIndex* value 100 as the starting point for index lookup, eliminating full index traversal and achieving $O(1)$ query complexity through stateful index pointers. Experimental results show that the *HintIndex* hits the correct segment in most cases, yielding a notable 11.7% improvement in throughput (detailed in § 5.4).

4.3 Operator Module

The operator module provides support for direct computation on key operators in TSDBs, with a focus on implementing effective homomorphic query operators that ensure *directness* and *effective homomorphic query*. We implement six core operators as detailed in Section 2.1. Using homomorphic query techniques, CompressIoTDB avoids decompression, effectively utilizing the compact information in the compressed time series data to reduce redundant computations and enhance query performance.

4.3.1 Operator Design. In CompressIoTDB, we provide full support for RLE and Dictionary encoding. Queries with operators from $\Pi = \{\mathcal{F}, \mathcal{J}, \mathcal{A}, \mathcal{G}, \mathcal{E}, \mathcal{S}\}$ are mapped to a fully homomorphic queries, while others are handled as partial homomorphic queries. For Ts_2Diff, we support aggregation and expression operators. Operations are pushed down to allow direct access to compressed data, minimizing transmission overhead [90]. For queries involving columns compressed using different algorithms, each column is read sequentially through unified interface as compression block and processed according to its specific algorithm.

Filter operator. The value filter operator, supporting unary predicates, binary comparisons, and regular expressions, are evaluated run- or dictionary-wise. For RLE algorithm, we scan RLE pattern blocks, where repeated data points are computed only once to reduce redundant calculations. For dictionary algorithm, we probe the dictionary directly and reuse the resulting bitmap efficiently.

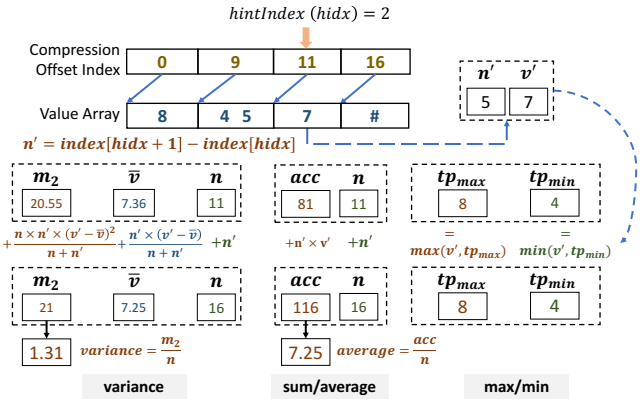


Figure 4: Homomorphic aggregation on RLE-encoded data

Aggregation operator. We implement common aggregation operators, including average, variance, sum, max, min, and count, processing compressed data incrementally using temporary state accumulators. Figure 4 illustrates homomorphic aggregation over RLE-compressed data, where cumulative variables are updated during traversal to avoid loading all data at once. For `variance`, three variables are maintained: m_2 (sum of squared deviations from the mean), $\bar{v}$ (current mean), and n (number of processed data points). The final variance is derived from m_2. For `sum` and `average`, we track the running sum of the series, while for `max/min`, we maintain the current minimum or maximum value. Each time the operator receives the next data run, represented as (n', v'), the cumulative variables are updated incrementally. This allows efficient aggregation without iterating through every individual values. For Ts_2Diff, aggregations are computed by applying formulas to each compression block. For instance, the block sum is derived as $\Sigma_{i=0}^{n}((n-i)d_i)$, where n is the block size and d_i is the i-th delta value. For brevity, we skip further details here.

Expression operator. Expression operators encompass arithmetic, logical, and comparison operations. For unary operations, RLE-compressed data is processed by traversing RLE patterns, computing each repeated value only once. Dictionary encoding traverses the dictionary table without modifying the actual data sequence. In Ts_2Diff, addition and subtraction operations are performed exclusively on base values, avoiding iterating through all data points.

Algorithm 1: Homomorphic Join with RLE-Encoding

Input: *blocks*: data block array, *rows*: selected join-row index list.
Output: *resultBuilder*: builders for output columns.

```
1  for each column index j in blocks do
2      builder ← resultBuilder.get(j) ;
3      CompColumn ← blocks.getColumn(j) ;
4      if CompColumn is RLE then
           /* Process each run in the RLE column          */
5          for each run r in CompColumn do
6              subset ← { k ∈ rows | k ∈ r.range} ;
7              if r represents a single repeated value then
8                  for each seq in subset do
9                      builder.writePattern(seq = null_seq ? null :
                         r.value, |seq|) ;
10             else
11                 vals ← [ r.getValue(k − r.start) or null | k ∈
                     subset ] ;
12                 builder.writePattern(vals, |subset|) ;
13     else
           /* Traditional join                            */
14 return resultBuilder;
```

Timestamp-based join operator. In TSDBs, data is typically joined using timestamp alignment. Aligned data is pre-aligned on disk during writes, shifting alignment costs to ingestion and eliminating query-time operations. For unaligned data, joins may introduce null values We dynamically encode the inserted nulls while compacting the joined series without requiring fully decompression. Algorithm 1 shows an example of homomorphic joins on RLE-encoded data via column-wise processing. Given selected row index list, for each column, we first initialize output buffers and load compressed data (Lines 1-3). On RLE-encoded columns (Line 4), we process runs (Line 5) by: ① Identifying overlapping row ranges (Lines 6). ② For constant runs, we directly write uniform values or nulls (Lines 7-9); For non-constant runs, we extract values sequentially (Lines 10-12). Uncompressed columns use standard

row-wise joins (Line 13) and results are return as a builder (Line 14). The algorithm's ideal time complexity is $O(RC)$, where R and C are the number of RLE runs and columns, outperforming traditional solution ($O(NC)$, N is the number of selected rows) when $R \ll N$.

Group by sliding window operator. This operator is often used in conjunction with aggregation operators. Based on the intervals specified by time windows, we divide the data columns into small chunks in compressed format. We perform direct calculations on the compressed time series data within each of these chunks.

Slicing operator. Slicing enables efficient sequential scans of large historical time series by processing data in batches to avoid memory overload. This operator directly identifies the `startOffset` and `endOffset` using the *HintIndex*. It then extracts the corresponding sub-array for `values` and incrementally updates the `Compression Offset Index` during traversal.

Besides above core operators, we implement two more basic operations to support a broader range of queries. 1) **Reversal** is performed when queries require results in order of time. By default, Apache IoTDB returns query results in ascending time order. When a `ORDER BY TIME DESC` clause is used, the results are reversed. 2) **(De)Serialization** is used for data transmission. By serializing CompColumn into a compact byte stream, it reduces transmission overhead. The serialized format varies by encoding algorithm.

4.3.2 Running Example of Homomorphic Query. Using RLE-encoded time series $s = \langle (4,3),(9,8),(5,7),(4,\langle 3,4,5,3\rangle)\rangle$ as depicted in Figure 3, we illustrate the query processing by query: `SELECT s/2 FROM series WHERE s > 3 OFFSET 11 LIMIT 4`.

① **CompColumn construction and filter push down.** We first load time series data from the storage layer into memory as Compressed Data Chunks. it is constructed into CompColumn, restoring 32-bit integers via bit-unpacking first. In the `CompColumnBuilder`, we pack each pattern into a compression block (`values`) within CompColumn. At the same time, we calculate each block's start offset to build the compression offset index, with `coIndex[0]` initialized to 0. Notably, the filter operator is pushed down to CompColumn construction phase to reduce the overhead of CompColumn building and data transmission. Specifically, for the first pattern $(4,3)$, since 3 fails the filter, this pattern is filtered out and will not participate in the construction. For the second pattern $(9,8)$, since $8 > 3$, it is retained, and thus we initialize `values[0]` to $\langle 8\rangle$ and set `coIndex[1]` to 9. For the pattern $(4,\langle 3,4,5,3\rangle)$, batch filtering is impossible, requiring per-value checks. Among these, only 4 and 5 satisfy the filtering condition; hence, the pattern is transformed into $(2,\langle 4,5\rangle)$. We set `values[1]` to $\langle 4,5\rangle$, and set `coIndex[2]` to 11. For the last pattern (5,7), the computation procedure is the same. After filtering, the final `values` array is $\langle 8,\langle 4,5\rangle,7\rangle$, with `coIndex` initialized to $\langle 0,9,11,16\rangle$. The hint index is initialized to 0.

② **Expression operator.** First, we retrieve the first pattern ($v = 8$) in CompColumn. Instead of computing run lengths, we directly perform calculations on the value, yielding $v = 8/2 = 4$. Subsequently, for the next non-run-length-encoded pattern, we process each data point individually; the last value $v = 7$ undergoes the same computation to produce $v = 3.5$. This yields a compressed values array $\langle 4,\langle 2,2.5\rangle,3.5\rangle$, while `coIndex` remains unchanged.

③ **Slicing operator.** To retrieve 5 data points at offset 11, we finally execute the slicing operator. With *HintIndex* initialized to 0, we first locate the offset 11 via `CoIndex`. This yields index = 1 (`coIndex[1]` = 9 and `coIndex[2]` = 11). Consequently, we discard `values[0]` and update `coIndex[1]` to $11-11+1 = 1$, set $hintindex = 1$, and retain only the last element of `values[1]`. For end offset $11+4 = 15$, we traverse from $hintindex$ to avoid full scans, identifying index = 2 (`coIndex[2]` = 11, `coIndex[3]` = 16). The `coIndex[2]` is updated to `coIndex[1] + 15 - coIndex[2]` = 5. The resulting value array becomes $\langle 2.5,3.5\rangle$, with the reconstructed `coIndex` as $\langle 0,1,5\rangle$. Given that **Group by sliding window operator** partitions time series similarly to the slicing operator, using time windows instead of offsets, further details of this operator are omitted for brevity.

④ **Aggregation operator.** Consider query `SELECT variance(s) FROM series WHERE s > 3`, we execute aggregation operator on CompColumn constructed after filtering in ①, using incremental temporary state accumulators. With n, v', and m_2 both initialized to 0, we proceeds as follows: Initially, we iterate over CompColumn, fetching the first pattern ($v' = 8$). The n' is computed as `coIndex[1] − coIndex[0]` = 9. Then, using the formula shown in Figure 4 for incremental computation, we obtain $m_2 = 0$, $\overline{v} = 8$, and $n = 9$. Next, we move to the following pattern, which is not in run-length format. Each value is processed incrementally with $n' = 1$, resulting in $m_2 = 20.55$, $\overline{v} = 7.36$, and $n = 11$, as shown in Figure 4. For the last pattern $v' = 7$, n' is computed as `coIndex[3] − coIndex[2]` = 5. As a result, we get $m_2 = 21$, $\overline{v} = 7.25$, and $n = 14$, yielding a variance of 1.31.

4.4 Optimization Module

The objective of the optimization module is to facilitate *effective restore* and *effective homomorphic queries* in CompressIoTDB by reducing data transmission and reading overhead. This is achieved through two key optimizations: 1) dynamic auxiliary management, and 2) late decompression for TsFile.

4.4.1 Dynamic Auxiliary Management. Time series data can be either unaligned (each column has its own time column) or aligned (columns share a time column), as show in Figure 5 (a) and (c). To handle nulls in aligned formats, databases typically use a compact layout where non-null values are stored contiguously, with nulls tracked via a *bitmap* [1, 10, 11]. Additionally, lazy deletion strategies are employed to avoid physical modifications by maintaining a *deletion list* that records the positions of deleted data.

Analysis. This compact layout and lazy deletion strategy, while optimizing storage and I/O, introduce complexity for homomorphic queries. Nulls must be scattered into their correct positions using the bitmap, leading to overhead during series scan. Lazy deletion requires checking the deletion lists, further complicating data access. These techniques disrupt the structure of compressed data, which negatively impacts the performance of homomorphic queries.

Design. Instead of decompressing the entire dataset for each query, we introduce a *dynamic encoding strategy* that maintains data in a compact form while handling auxiliary structures, such as nulls and deletion lists. For unaligned data, we directly traverse each RLE pattern, and adjust its run-length by skipping deleted entries, never decompress the data. For aligned data, we use dynamic encoding to handle nulls without disturbing the compressed data structure. First, based on the deletion list, we slice the null sequence into runs

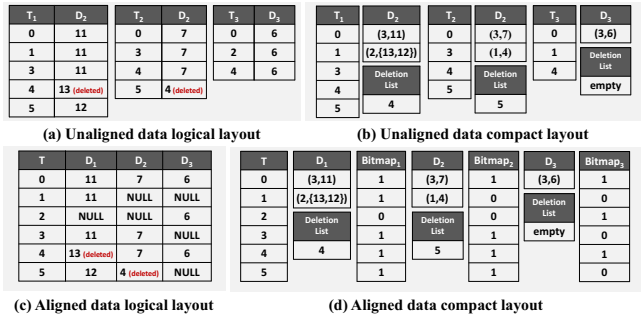


Figure 5: Compact layout example using RLE encoding

that align with each RLE pattern. Then, we encode and merge continuous null runs directly back into the compressed data, preserving its structure. For non-continuous nulls, as their insertion disrupting the compressed data, we revert the affected segment to its original format for simplify parsing in subsequent queries. For example, in the unaligned layout (Figure 5 (b)), the deletion list indicates that the entry at timestamp 4 in D_1 has been deleted, so it is removed, resulting in compressed series $D_1 = \langle(3, 11), (1, 12)\rangle$. In the aligned layout (Figure 5 (d)), D_2 initially consists of six-bits bitmap (100111) and RLE data $\langle(3, 7), (1, 4)\rangle$. After checking the deletion list and bitmap, two nulls are inserted and a value is deleted, resulting in compressed $D_2 = \langle(1, 7), (2, NULL), (2, 7)\rangle$; D_3 degrades into $\langle(6, \langle 6\ NULL\ 6\ NULL\ 6\ NULL\rangle)\rangle$, which no longer suits RLE encoding, so we revert it to its original form. This dynamic encoding strategy is applied during the series scan and does not require full de-compression and re-compression, preserving the *directness* of the query process.

4.4.2 Late Decompression for TsFile. To minimize decompression and transmission costs, we employ late decompression for TsFile, delaying general-purpose decompression until data access.

Analysis. As shown in Figure 6 (a), in Apache IoTDB's original approach, compressed data is stored in TsFile, where each page is compressed using light-weight algorithms like RLE or Gorilla, followed by general-purpose compression (e.g., LZ4). However, during queries, the entire chunk undergoes general-purpose decompression even if only a small subset of data is needed. For example, querying the first 200 data points in a chunk requires decompressing all pages within the chunk, leading to wasted CPU cycles.

Design. We address this inefficiency by deferring general-purpose decompression until the series scan phase, as illustrated in Figure 6 (b). Specifically, we defer general-purpose decompression to the series-scan phase: chunks are read in their dual-compressed form and iterated page by page, invoking heavyweight decompression only when the specific page is accessed and bypassing the light-weight layer by directly instantiating CompColumn. This ensures that only the data actually scanned is decompressed, cutting CPU overhead. This feature has been integrated into the Apache TsFile[2] storage layer, enhancing overall system performance in Apache IoTDB. Additionally, we encapsulate decompression logic within CompColumn to support partially homomorphic queries,

[2]https://github.com/apache/tsfile

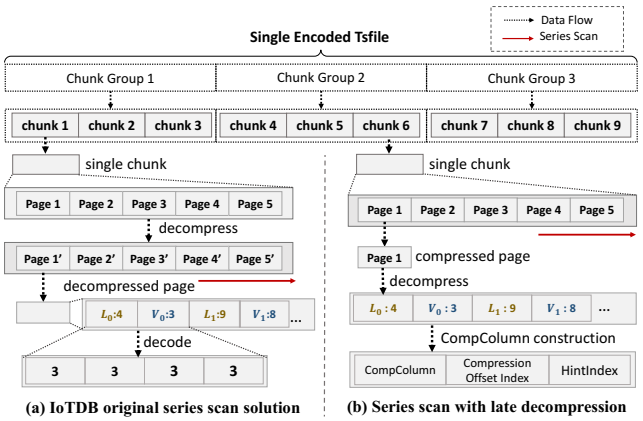


Figure 6: Illustration of late decompression strategy

exposes uncompressed views for operators that do not support direct computation on compressed time series data. This design keeps data compressed as long as possible, minimizing transmission costs by reducing the volume of transferred data.

5 EVALUATION

5.1 Experimental Setup

Baselines. We evaluate CompressIoTDB with three baselines, Uncompressed, CompressIoTDB-NoLate, and IoTDB. Uncompressed stores and queries data without compression. CompressIoTDB-NoLate represents CompressIoTDB without the late decompression optimization. IoTDB refers to the original Apache IoTDB, without any of our enhancements. We use two light-weight compressions: Run-Length Encoding (RLE) for numerical data and Dictionary Encoding for string data. Meanwhile, for delta-based Ts_2Diff, we conduct queries with expression and aggregation operators to demonstrate its performance. All light-weight compressed data is further processed with LZ4 to improve compression efficiency.

Datasets. We evaluate CompressIoTDB using five open-source real-world time series datasets as well as synthetic datasets with diverse characteristics generated by IoT-benchmark, thereby assessing the system under both practical conditions and varying dataset configurations. For details on the data generation algorithms and parameters, refer to Appendix B. We extend IoT-benchmark to support expression queries. Table 2 provides dataset statistics. These datasets have been widely used in previous studies [8, 52, 84, 87, 92].

Table 2: Datasets

Name	Attr #	Length
Weather Forecast (WF) [99]	6	910,576
AMPds [60]	11	10,490,860
Smart Grid (SG) [46]	5	100,000,000
Linear Road (LR) [13]	6	108,437,193
Computer Monitor (CM) [62]	4	144,370,688
IoT-benchmark [58, 84]	5	$10^5 - 10^9$

Benchmark queries. We use ten queries derived from real-world applications, combining various operators discussed in § 3.1. The detailed applications of these queries are provided in Appendix E.For

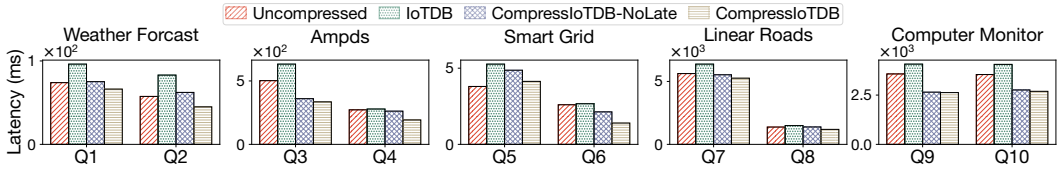


Figure 7: Latency on real world datasets

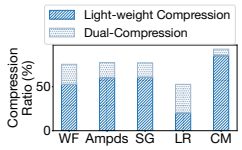


Figure 8: CPR of real world datasets

Table 3: Queries

Query	Detail
Q1	SELECT AVG (9 * bottom_temperature / 5 + 32) AS AvgBottomTemp_F FROM root.air.wf WHERE bottom_temperature > 20;
Q2	SELECT AVG (wind_level), AVG (wind_direction) FROM root.air.wf GROUP BY ((2013-09-01T06:00:56.000+08:00, 2013-9-01T06:16:00.000+08:00], 1m);
Q3	SELECT STDDEV (I) AS CurrentStandardDeviation, VARIANCE (I) AS CurrentVariance FROM root.amp;
Q4	SELECT SUM (S) AS TotalApparentEnergy_kVAh, SUM(P) AS TotalRealPower_kWh FROM root.amp;
Q5	SELECT AVG (plug) FROM root.sg GROUP BY ((2013-09-01T06:00:56.000+08:00, 2013-9-02T12:05:00.000+08:00], 1d);
Q6	SELECT SUM (house) FROM root.sg GROUP BY ((2013-09-01T06:00:56.000+08:00, 2013-9-02T12:05:00.000+08:00], 1d);
Q7	SELECT VARIANCE (direction) FROM root.lr WHERE lane = 0 GROUP BY ((2013-09-01T06:00:56.000+08:00, 2013-9-02T12:05:00.000+08:00], 1d);
Q8	SELECT AVG (speed) FROM root.lr GROUP BY ((2013-09-01T06:00:56.000+08:00, 2013-9-02T12:05:00.000+08:00], 1d);
Q9	SELECT VARIANCE (cpu) FROM root.cm GROUP BY ((2013-09-01T06:00:56.000+08:00, 2013-09-02T22:07:06.688+08:00], 1d);
Q10	SELECT VARIANCE (priorities) FROM root.cm GROUP BY ((2013-09-01T06:00:56.000+08:00, 2013-09-02T22:07:06.688+08:00], 1d);
QT1	SELECT AGG (v1), ... , AGG (v5) FROM data WHER time > ? AND time < ?;
QT2	SELECT AGG (v1), ... , AGG (v5) FROM data WHERE v1 op ? AND ... AND v5 op ?;
QT3	SELECT EXP (v1), ... , EXP (v5) FROM data WHERE v1 op ? AND ... AND v5 op ?;

IoT-benchmark, we run three basic queries [50, 58] to compare the baselines. Specifically, the three queries are 1) aggregation with a time filter, 2) aggregation with value filters, and 3) expression with value filters. Each query involves data from five sensors. Details of each query are provided in Table 3.

Platform. We perform experiments on a server equipped with an Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz and 64 GB of RAM.

5.2 Overall Evaluation

5.2.1 Performance. We explore the overall performance on five real-world datasets. Specifically, we evaluate the latency of CompressIoTDB by measuring the number of queries processed within a given time period. Each query in Table 3 is executed 200 times, with the first 50 rounds used for system warm-up. We then measure the total query time for the subsequent 150 rounds to calculate the average latency. The results are presented in Figure 7.

These results lead to several key conclusions. ① CompressIoTDB consistently accelerates queries, reducing latency by 33.1% (53.4% throughput gain) over IoTDB and 20.3% (29.9% throughput gain) against Uncompressed, highlighting its dual benefit of minimizing decompression overhead while enhancing compressed data processing. ② The performance gains correlate with compression efficiency. For instance, Ampds (highest compression ratio) achieves 39.0% latency reduction over IoTDB, while Linear Roads (20% compression ratio) shows modest gains (18.9%). This is because datasets with higher compression ratios present more opportunities for I/O optimization and efficient computation of compressed data. ③ CompressIoTDB outperforms its variant without late decompression (CompressIoTDB-NoLate) by 14.5% in latency, demonstrating its effectiveness at reducing unnecessary decompression and improving overall efficiency.

5.2.2 Space Saving and Late Decompression. Figure 8 presents the compression ratios for each real-world dataset, calculated as: Compression Ratio (CPR) $= 1 - \frac{\text{size of compressed data}}{\text{size of uncompressed data}}$. The bottom dark blue bars in Figure 8 represent the compression ratios achieved by light-weight compression, while the upper light bars show the additional compression from applying general-purpose compression to the light-weight-compressed data. Several key observations can be made. ① For most datasets, the majority of space reduction results from the light-weight compression algorithm, with general-purpose compression contributing relatively little. This validates our approach of performing direct queries on light-weight-compressed data while delaying decompression for general-purpose compression. ② The impact of late decompression is proportional to the contribution of general-purpose compression. For example, in the Computer Monitor dataset, general-purpose compression contributes only 7.5% (to a total of 92.7%), so late decompression yields only a 1.7% speedup. In contrast, for the Linear Roads dataset, the light-weight compression alone gives 20%, but general compression boosts this to 52.7%. Without the late decompression strategy, the cost of decompression outweighs the performance benefits of homomorphic querying, causing CompressIoTDB-NoLate to perform worse than Uncompressed.

5.3 Macro-Benchmark Evaluation

Building on our evaluation of CompressIoTDB on real-world datasets, we further assess its performance via macro-benchmarks. Specifically, we analyze three basic queries across datasets with varying repetition rates [3] and sizes, generated using the IoT-benchmark. The datasets comprise monotonically non-decreasing time series with a mean value of 1×10^4, repetition rates ranging from 0 to 1, and lengths up to 1.5×10^8. Each dataset simulates five sensors per device with FLOAT data (generation algorithm in Appendix B)."

[3]Repetition rate refers to the ratio of consecutive repeated data in the dataset. Lengths of consecutive repeated data sequences are not correlated with repetition rates.

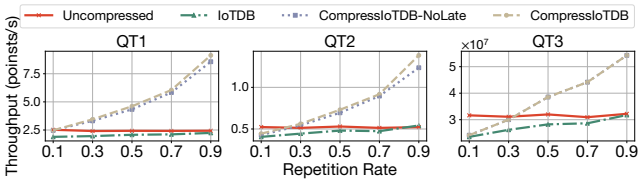


Figure 9: Throughput across varying repetition rates

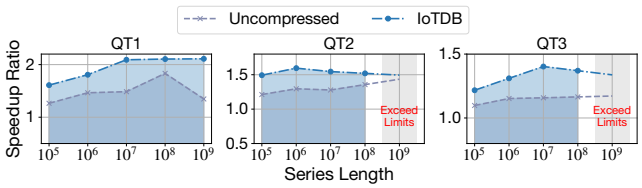


Figure 10: Speedup ratios on datasets of varying series length

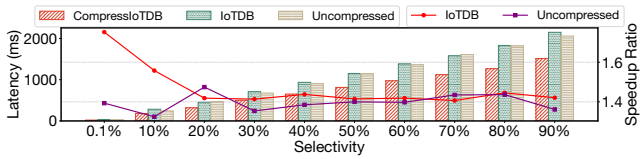


Figure 11: Performance on queries with multi-selectivity

5.3.1 Evaluation on Datasets with Varying Repetition Rates. Empirical analyses indicate a strong correlation between the performance of CompressIoTDB and the dataset's compression ratio. Specifically, datasets with higher repetition rates, which are conducive to Run-Length Encoding (RLE), yield better performance. This outcome is expected as low repetition reduces RLE effectiveness, incurring additional overheads in data maintenance during homomorphic queries. Figure 9 illustrates the query latency and throughput results for datasets with a length of 1×10^8 and varying repetition rates. We draw several key observations. ① Higher repetition rates boost CompressIoTDB's throughput significantly, while IoTDB sees modest gains and Uncompressed remains stable. ② CompressIoTDB outperforms IoTDB by 36.9% on average (up to 75.5% at 0.9 repetition) and surpasses CompressIoTDB-NoLate by 4.4%, underscoring late decompression's efficacy across all repetition rates. ③ At low repetition rates, CompressIoTDB slightly lags Uncompressed due to minimal RLE compression and auxiliary overhead. However, such cases are rare in practice, as administrators optimize compression strategies based on data traits.

5.3.2 Evaluation on Datasets with Varying Size. We conduct experiments on datasets of different series length, from 10^5 to 10^9, all with a repetition rate of 0.5. The speedup ratios of CompressIoTDB compared to Uncompressed and IoTDB are shown in Figure 10. The speedup ratio is defined as: $\text{Speedup Ratio} = \frac{\text{latency of baseline}}{\text{latency of CompressIoTDB}}$. We have the following observations. ① CompressIoTDB improves performance across all series lengths of the datasets. CompressIoTDB achieves an average speedup of 48.0% compared to IoTDB and 35.4% compared to Uncompressed. ② While IoTDB's I/O and decompression overheads grow with data size, CompressIoTDB's time-range query (QT1) performance improves with larger datasets, as late decompression minimizes unnecessary processing, maintaining stable decompression and data transfer costs. However, for Uncompressed, the rising I/O costs are offset by avoiding decompression, narrowing CompressIoTDB's advantage at scale. ③ CompressIoTDB maintains stable performance for full data queries. For queries without time-range filters (QT2, QT3), where the amount of queried data scales with the dataset size, the performance improvement of CompressIoTDB remains consistent relative to IoTDB and Uncompressed, demonstrating strong scalability. ④ When series length reaches 10^9, IoTDB times out (60s threshold) on queries on full series (QT2, QT3).

We further conduct experiments on sequence lengths ranging from 1×10^8 to 1×10^9. As shown in Table 4, IoTDB fails to support query types QT2 and QT3 at 7×10^8 due to scalability constraints. In contrast, while the Uncompressed incurs significant I/O overhead, it remains functional by avoiding the critical decompression cost. Our proposed CompressIoTDB, while maintaining data compression to conserve storage and memory resources, achieves an average latency reduction of 7.3% compared to Uncompressed. This advancement effectively pushes the upper bound of system performance by harmonizing efficient compression with rapid query execution.

Table 4: Micro-analysis on datasets of varying series length

Series Length	QT1		QT2		QT3	
	Uncomp.	IoTDB	Uncomp.	IoTDB	Uncomp.	IoTDB
2×10^8	1.46	2.02	1.72	1.55	1.47	1.40
5×10^8	1.25	1.43	1.35	1.29	1.21	1.20
7×10^8	1.21	1.38	1.04	–	1.00	–
1×10^9	1.35	2.11	–	–	–	–

Speedup ratios compared to baseline methods; "–" denotes timeout cases.

5.3.3 Evaluation of Queries with Multi-Selectivity. We conduct experiments on a dataset of size 10^8 with a repetition rate of 0.5, executing time-range queries (QT1) with selectivity ranging from 0.1% to 90%. The latency and speedup ratios, as defined in § 5.3.2, are illustrated in Figure 11. Our findings are as follows. ① As selectivity increases, CompressIoTDB effectively keeps latency from rising too sharply, maintaining a considerable speedup across all selectivity. It achieves a 32.4% latency reduction compared to IoTDB and 28.3% improvement over the Uncompressed baseline, demonstrating the effectiveness of our method. ② Under extreme low selectivity conditions (0.1%), CompressIoTDB attains a peak 42.9% latency reduction versus IoTDB. This is because our late decompression effectively limits unnecessary decompression (explained in detail in § 5.3.2). ③ At extremely low selectivity, the speedup over Uncompressed is less pronounced than over IoTDB. This is because uncompressed data avoids decompression overhead. Although uncompressed storage incurs significant I/O costs, the query bottleneck remains tied to decompression operations. Thus, our method, which optimizes decompression efficiency, yields smaller relative gains in low-selectivity scenarios where decompression dominates execution time.

5.4 Detailed Analysis

5.4.1 Evaluation of HintIndex. As discussed in § 4.2.2, CompressIoTDB leverages *HintIndex* to record the last accessed position, avoiding unnecessary traversal of the index from the start during sequential data access. We conduct an ablation study to evaluate the

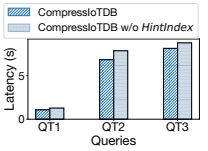


Figure 12: Effects of HintIndex

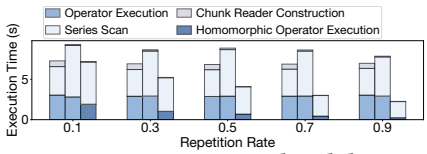


Figure 13: Execution time breakdown on datasets of varying repetition rates

performance gains attributed to *HintIndex*. As shown in Figure 12, *HintIndex* provides an average performance improvement of 11.7%. Furthermore, it can be observed that the absolute time saved by *HintIndex* is positively correlated with the amount of data accessed during the query. For queries without time-range filters (QT2, QT3), where the queried data volume is larger, *HintIndex* provides more substantial time savings.

5.4.2 Execution Time Breakdown. We evaluate the execution time for each phase across datasets with varying repetition rates and a length of 1×10^8, using query "`SELECT VARIANCE(*) FROM data`". The query execution is divided into three phases: chunk reader construction, series scan, and operator execution. For detailed explanation of query phases, please refer to Appendix G.Figure 13 presents the breakdown of execution time for Uncompressed (left bar), IoTDB (middle bar), and CompressIoTDB (right bar). The key observations are as follows. ① Chunk reader speedup: Our approach delivers a 20.8× average speedup over the Uncompressed by reduced data-transfer costs, and a 4.4× speedup over IoTDB thanks to late decompression that defers decompression to the series scan phase. ② Series scan dominates query execution time. It accounts for 48.2% of runtime on Uncompressed, 65.0% on IoTDB, and 79.5% in CompressIoTDB due to its deferred decompression to this phase. Despite the higher share, we still cut overall latency by 39% versus IoTDB by skipping light-weight decompression and efficiently managing auxiliary structure restoration, highlighting the effectiveness of our optimizations. ③ CompressIoTDB's homomorphic operator execution runs 5.5× faster average than Uncompressed. This speedup increases as the repetition rate increases, reaching 12.1× at a repetition rate of 0.9, underscoring the effectiveness of CompressIoTDB for homomorphic queries.

5.4.3 Memory Usage. We measure the memory usage of the CompColumn structure using the compression ratio (CPR = CompColumn size/uncompressed size) under the same query and dataset settings as in § 5.4.2. Table 5 shows IoTDB's memory use stays stable across repetition rates, while CompColumn achieves a 20% reduction on average. However, for CompressIoTDB, a low repetition rate can result in higher memory usage. This is because, under low repetition rate, the RLE-compressed data closely resembles raw data, and additional data structures introduced by CompColumn to manage compression lead to increased memory overhead. Nevertheless, as illustrated in Figure 13, even at a repetition rate of 0.1, CompressIoTDB achieves an 22.7% reduction in latency compared to IoTDB, demonstrating the effectiveness of CompColumn structure.

5.5 Discussion on Capability

We summarize prevalent time series compression schemes, including all encoding algorithms implemented in Apache IoTDB, and how they are combined from the basic components (detailed in § 3), as shown in Table 6. Key observations are outlined below. ① Nearly all these algorithms employ bit-packing to reduce storage overhead. ② Many algorithms compress data based on the repeat encoding, capitalizing on the frequent occurrence of repeated or uniformly incremental values in time series data. ③ Several algorithms are optimized for the delta-encoding. For delta-encoding-based schemes, our analysis (Section 3) reveals limited homomorphic support. In the preceding experiments, due to the delta-based method's insufficient support for certain operators, we employed RLE and Dictionary encoding, compensating for the Dict, Repeat, and Bit-Packing components. To evaluate the efficacy of our approach in delta-based encoding, we implement Ts_2Diff and conduct an evaluation on aggregation query with expressions: "`SELECT AGG(EXP(*)) FROM data`", using the same dataset described in Section 5.3.3. Results show that our method achieves a 43.2% reduction in latency compared to IoTDB and 18.9% to Uncompressed baselines, demonstrating its performance advantages.

Table 5: Memory usage (GB) of CompColumn

Repetition Rate	0.1	0.3	0.5	0.7	0.9	Average
IoTDB	3.08	3.08	3.08	3.08	3.08	3.08
CompressIoTDB	3.90	3.28	2.44	1.69	0.99	2.46
CPR	1.27	1.07	0.79	0.55	0.32	0.80

Table 6: Basic components of time series compression

Comp.	ZigZag	RLE	Bitmap	Dict.	Ts_2Diff	Gorilla	Chimp	RLBE
Dict	–	–	✓	✓	–	–	–	–
Delta	–	–	–	–	✓	✓	✓	✓
Repeat	–	✓	–	✓	–	✓	✓	✓
Pack	✓	✓	✓	✓	✓	✓	✓	✓

Remark: CompressIoTDB maintains compatibility with mainstream database systems. Detailed discussions and preliminary experimental results are provided in Appendix F.

6 CONCLUSION

In this paper, we present a novel homomorphic compression framework for time series data. Based on a formal model, we propose CompressIoTDB. It supports homomorphic computation for key time series database operators with unified modular design and system-level optimizations, significantly reducing decompression overhead and I/O costs. Our experimental results show that CompressIoTDB achieves significant performance improvement.
Supplemental Materials: The code and an appendix are available at https://github.com/yuxin370/CompressIoTDB/tree/master.

ACKNOWLEDGMENTS

This work is supported by the National Key R&D Program of China (No. 2024YFB3309601), National Natural Science Foundation of China (No. 62322213, 62461146205, and 92267203), and Beijing Nova Program (No. 20230484397 and 20220484137). Yuxin Tang, Feng Zhang, Jiawei Guan, and Xiaoyong Du are with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China. Feng Zhang is the corresponding author of this paper.

REFERENCES

[1] Abadi and Daniel. 2007. Column Stores for Wide and Sparse Data., In Conference on Innovative Data Systems Research. *CIDR 2007 - 3rd Biennial Conference on Innovative Data Systems Research*, 292–297.

[2] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. 2003. Aurora: a data stream management system. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (San Diego, California) *(SIGMOD '03)*. Association for Computing Machinery, New York, NY, USA, 666. https://doi.org/10.1145/872757.872855

[3] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (Chicago, IL, USA) *(SIGMOD '06)*. Association for Computing Machinery, New York, NY, USA, 671–682. https://doi.org/10.1145/1142473.1142548

[4] Colin Adams, Luis Alonso, Benjamin Atkin, John P. Banning, Sumeer Bhola, Richard W. Buskens, Ming Chen, Xi Chen, Yoo Chung, Qin Jia, Nick Sakharov, George Talbot, Nick Taylor, and Adam Tart. 2020. Monarch: Google's Planet-Scale In-Memory Time Series Database. *Proceedings of the VLDB Endowment* 13 (2020), 3181 – 3194.

[5] Azim Afroozeh and Peter Boncz. 2023. The FastLanes Compression Layout: Decoding > 100 Billion Integers per Second with Scalar Code. *Proc. VLDB Endow.* 16, 9 (May 2023), 2132–2144. https://doi.org/10.14778/3598581.3598587

[6] Azim Afroozeh, Leonardo X. Kuffo, and Peter Boncz. 2023. ALP: Adaptive Lossless floating-Point Compression. *Proc. ACM Manag. Data* 1, 4, Article 230 (Dec. 2023), 26 pages. https://doi.org/10.1145/3626717

[7] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. 2015. Succinct: enabling queries on compressed data. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA) *(NSDI'15)*. USENIX Association, USA, 337–350.

[8] Yihao Ang, Qiang Huang, Yifan Bao, Anthony K. H. Tung, and Zhiyong Huang. 2023. TSGBench: Time Series Generation Benchmark. *Proc. VLDB Endow.* 17, 3 (nov 2023), 305–318. https://doi.org/10.14778/3632093.3632097

[9] Apache IoTDB. 2024. https://iotdb.apache.org/. Accessed: 2025-06-28.

[10] Apache ORC. 2024. https://orc.apache.org/. Accessed: 2025-06-28.

[11] Apache Parquet. 2024. https://parquet.apache.org/. Accessed: 2025-06-28.

[12] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal* 15, 2 (June 2006), 121–142. https://doi.org/10.1007/s00778-004-0147-z

[13] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. 2004. Linear road: a stream data management benchmark *(VLDB '04)*. VLDB Endowment, 480–491.

[14] Hillel Avni, Alisher Aliev, Oren Amor, Aharon Avitzur, Ilan Bronshtein, Eli Ginot, Shay Goikhman, Eliezer Levy, Idan Levy, Fuyang Lu, Liran Mishali, Yeqin Mo, Nir Pachter, Dima Sivov, Vinoth Veeraraghavan, Vladi Vexler, Lei Wang, and Peng Wang. 2020. Industrial-strength OLTP using main memory and many cores. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3099–3111. https://doi.org/10.14778/3415478.3415537

[15] Bruno Barbarioli, Gabriel Mersy, Stavros Sintos, and Sanjay Krishnan. 2023. Hierarchical Residual Encoding for Multiresolution Time Series Compression. *Proc. ACM Manag. Data* 1, 1, Article 99 (May 2023), 26 pages. https://doi.org/10.1145/3588953

[16] Davis Blalock, Samuel Madden, and John Guttag. 2018. Sprintz: Time series compression for the internet of things. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2, 3 (2018), 1–23.

[17] Radu Boncea and Ioan Bacivarov. 2016. A system architecture for monitoring the reliability of iot. In *Proceedings of the 15th International Conference on Quality and Dependability*. 143–150.

[18] Martin Burtscher and Paruj Ratanaworabhan. 2007. High Throughput Compression of Double-Precision Floating-Point Data. In *2007 Data Compression Conference (DCC'07)*. 293–302. https://doi.org/10.1109/DCC.2007.44

[19] Martin Burtscher and Paruj Ratanaworabhan. 2009. FPC: A High-Speed Compressor for Double-Precision Floating-Point Data. *IEEE Trans. Comput.* 58, 1 (2009), 18–31. https://doi.org/10.1109/TC.2008.131

[20] Shubham Chandak, Kedar Tatwawadi, Chengtao Wen, Lingyun Wang, Juan Aparicio Ojea, and Tsachy Weissman. 2020. LFZip: Lossy Compression of Multivariate Floating-Point Time Series Data via Improved Prediction. In *2020 Data Compression Conference (DCC)*. 342–351. https://doi.org/10.1109/DCC47342.2020.00042

[21] Zheng Chen, Feng Zhang, JiaWei Guan, Jidong Zhai, Xipeng Shen, Huanchen Zhang, Wentong Shu, and Xiaoyong Du. 2023. CompressGraph: Efficient Parallel Graph Analytics with Rule-Based Compression. *Proc. ACM Manag. Data* 1, 1, Article 4 (May 2023), 31 pages. https://doi.org/10.1145/3588684

[22] Giacomo Chiarot and Claudio Silvestri. 2023. Time Series Compression Survey. *ACM Comput. Surv.* 55, 10, Article 198 (Feb. 2023), 32 pages. https://doi.org/10.1145/3560814

[23] Andrew A Cook, Göksel Mısırlı, and Zhong Fan. 2019. Anomaly detection for IoT time-series data: A survey. *IEEE Internet of Things Journal* 7, 7 (2019), 6481–6494.

[24] Rui Ding, Qiang Wang, Yingnong Dang, Qiang Fu, Haidong Zhang, and Dongmei Zhang. 2015. Yading: Fast clustering of large-scale time series data. *Proceedings of the VLDB Endowment* 8, 5 (2015), 473–484.

[25] Xiaoou Ding, Yingze Li, Hongzhi Wang, Chen Wang, Yida Liu, and Jianmin Wang. 2024. TSDDISCOVER: Discovering Data Dependency for Time Series Data. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 3668–3681. https://doi.org/10.1109/ICDE60146.2024.00282

[26] Xiaoou Ding, Yichen Song, Hongzhi Wang, Chen Wang, and Donghua Yang. 2024. MTSClean: Efficient Constraint-based Cleaning for Multi-Dimensional Time Series Data. *Proc. VLDB Endow.* 17, 13 (2024), 4840–4852. https://www.vldb.org/pvldb/vol17/p4840-wang.pdf

[27] Xiaoou Ding, Yichen Song, Hongzhi Wang, Donghua Yang, Chen Wang, and Jianmin Wang. 2024. Clean4TSDB: A Data Cleaning Tool for Time Series Databases. *Proc. VLDB Endow.* 17, 12 (2024), 4377–4380. https://doi.org/10.14778/3685800.3685879

[28] Zengyu Ding, Gang Mei, Salvatore Cuomo, Yixuan Li, and Nengxiong Xu. 2020. Comparison of estimating missing values in iot time series data using different interpolation algorithms. *International Journal of Parallel Programming* 48 (2020), 534–548.

[29] Frank Eichinger, Pavel Efros, Stamatis Karnouskos, and Klemens Böhm. 2015. A time-series compression technique and its application to the smart grid. *The VLDB Journal* 24, 2 (April 2015), 193–218. https://doi.org/10.1007/s00778-014-0368-8

[30] Wenfei Fan, Jianzhong Li, Xin Wang, and Yinghui Wu. 2012. Query preserving graph compression. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 157–168. https://doi.org/10.1145/2213836.2213855

[31] Chenguang Fang, Shaoxu Song, Haoquan Guan, Xiangdong Huang, Chen Wang, and Jianmin Wang. 2023. Grouping time series for efficient columnar storage. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.

[32] Paolo Ferragina, Rodrigo González, Gonzalo Navarro, and Rossano Venturini. 2009. Compressed text indexes: From theory to practice. *ACM J. Exp. Algorithmics* 13, Article 12 (Feb. 2009), 31 pages. https://doi.org/10.1145/1412228.1455268

[33] S. Golomb. 1966. Run-length encodings (Corresp.). *IEEE Transactions on Information Theory* 12, 3 (1966), 399–401. https://doi.org/10.1109/TIT.1966.1053907

[34] Adrián Gómez-Brandón, José R Paramá, Kevin Villalobos, Arantza Illarramendi, and Nieves R Brisaboa. 2021. Lossless compression of industrial time series with direct access. *Computers in Industry* 132 (2021), 103503.

[35] G. Graefe and L.D. Shapiro. 1991. Data compression and database performance. In *[Proceedings] 1991 Symposium on Applied Computing*. 22–27. https://doi.org/10.1109/SOAC.1991.143840

[36] Jiawei Guan, Feng Zhang, Siqi Ma, Kuangyu Chen, Yihua Hu, Yuxing Chen, Anqun Pan, and Xiaoyong Du. 2023. Homomorphic Compression: Making Text Processing on Compression Unlimited. *Proc. ACM Manag. Data* 1, 4, Article 271 (dec 2023), 28 pages. https://doi.org/10.1145/3626765

[37] Shai Halevi. 2017. *Homomorphic Encryption*. Springer International Publishing, Cham, 219–276. https://doi.org/10.1007/978-3-319-57048-8_5

[38] Sven Hielke Hepkema, Azim Afroozeh, Lotte Felius, Peter Boncz, and Stefan Manegold. 2025. G-ALP: Rethinking Light-weight Encodings for GPUs. In *Proceedings of the 21st International Workshop on Data Management on New Hardware, DaMoN 2025*.

[39] Aaron Hurst, Daniel E. Lucani, and Qi Zhang. 2024. PairwiseHist: Fast, Accurate and Space-Efficient Approximate Query Processing with Data Compression. *Proc. VLDB Endow.* 17, 6 (May 2024), 1432–1445. https://doi.org/10.14778/3648160.3648181

[40] InfluxDB 2024. https://www.influxdata.com/. Accessed: 2025-06-28.

[41] INTERNET OF THINGS MARKET ANALYSIS - 2032. 2024. https://www.fortunebusinessinsights.com/industry-reports/internet-of-things-iot-market-100307. Accessed: 2025-06-28.

[42] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. 2017. Time Series Management Systems: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 29, 11 (2017), 2581–2600. https://doi.org/10.1109/TKDE.2017.2740932

[43] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. 2018. Modelardb: Modular model-based time series management with spark and cassandra. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1688–1701.

[44] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. 2018. ModelarDB: modular model-based time series management with spark and cassandra. *Proc. VLDB Endow.* 11, 11 (July 2018), 1688–1701. https://doi.org/10.14778/3236187.3236215

[45] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. 2021. Scalable Model-Based Management of Correlated Dimensional Time Series in

ModelarDB+. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 1380–1391. https://doi.org/10.1109/ICDE51399.2021.00123

[46] Zbigniew Jerzak and Holger Ziekow. 2014. The DEBS 2014 grand challenge. In *Distributed Event-Based Systems*. https://api.semanticscholar.org/CorpusID:17908409

[47] Yunhong Ji, Wentao Huang, and Xuan Zhou. 2024. HeterMM: applying in-DRAM index to heterogeneous memory-based key-value stores. *Frontiers of Computer Science* 18, 4, Article 184612 (2024). https://doi.org/10.1007/s11704-024-3713-0

[48] Hao Jiang, Chunwei Liu, Qi Jin, John Paparrizos, and Aaron J. Elmore. 2020. PIDS: attribute decomposition for improved compression and query performance in columnar storage. *Proc. VLDB Endow.* 13, 6 (Feb. 2020), 925–938. https://doi.org/10.14778/3380750.3380761

[49] Hao Jiang, Chunwei Liu, John Paparrizos, Andrew A. Chien, Jihong Ma, and Aaron J. Elmore. 2021. Good to the Last Bit: Data-Driven Encoding with CodecDB. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 843–856. https://doi.org/10.1145/3448016.3457283

[50] Abdelouahab Khelifati, Mourad Khayati, Anton Dignös, Djellel Difallah, and Philippe Cudré-Mauroux. 2023. TSM-Bench: Benchmarking Time Series Database Systems for Monitoring Applications. *Proc. VLDB Endow.* 16, 11 (jul 2023), 3363–3376. https://doi.org/10.14778/3611479.3611532

[51] Xenophon Kitsios, Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. 2023. Sim-Piece: Highly Accurate Piecewise Linear Approximation through Similar Segment Merging. *Proc. VLDB Endow.* 16, 8 (April 2023), 1910–1922. https://doi.org/10.14778/3594512.3594521

[52] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter R. Pietzuch. 2016. SABER: Window-Based Hybrid Stream Processing for Heterogeneous Architectures. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 555–569. https://doi.org/10.1145/2882903.2882906

[53] Raghavendra Kumar, Pardeep Kumar, and Yugal Kumar. 2020. Time series data prediction using IoT and machine learning technique. *Procedia computer science* 167 (2020), 373–381.

[54] Ruiyuan Li, Zheng Li, Yi Wu, Chao Chen, and Yu Zheng. 2023. Elf: Erasing-Based Lossless Floating-Point Compression. *Proc. VLDB Endow.* 16, 7 (March 2023), 1763–1776. https://doi.org/10.14778/3587136.3587149

[55] Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. 2022. Chimp: efficient lossless floating point compression for time series databases. *Proc. VLDB Endow.* 15, 11 (July 2022), 3058–3070. https://doi.org/10.14778/3551793.3551852

[56] Chunbin Lin, Etienne Boursier, and Yannis Papakonstantinou. 2020. Plato: approximate analytics over compressed time series with tight deterministic error guarantees. *Proc. VLDB Endow.* 13, 7 (March 2020), 1105–1118. https://doi.org/10.14778/3384345.3384357

[57] Chunwei Liu, Hao Jiang, John Paparrizos, and Aaron J. Elmore. 2021. Decomposed bounded floats for fast compression and queries. *Proc. VLDB Endow.* 14, 11 (July 2021), 2586–2598. https://doi.org/10.14778/3476249.3476305

[58] Rui Liu and Jun Yuan. 2019. Benchmark Time Series Database with IoTDB-Benchmark for IoT Scenarios. *CoRR* abs/1901.08304 (2019). arXiv:1901.08304 http://arxiv.org/abs/1901.08304

[59] Yi Liu, Sahil Garg, Jiangtian Nie, Yang Zhang, Zehui Xiong, Jiawen Kang, and M Shamim Hossain. 2020. Deep anomaly detection for time-series data in industrial IoT: A communication-efficient on-device federated learning approach. *IEEE Internet of Things Journal* 8, 8 (2020), 6348–6358.

[60] Stephen Makonin, Bradley Ellert, Ivan V. Bajić, and Fred Popowich. 2016. Electricity, water, and natural gas consumption of a residential house in Canada from 2012 to 2014. *Scientific Data* 3 (2016). https://api.semanticscholar.org/CorpusID:1406747

[61] Hossein Maserrat and Jian Pei. 2010. Neighbor query friendly compression of social networks. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Washington, DC, USA) *(KDD '10)*. Association for Computing Machinery, New York, NY, USA, 533–542. https://doi.org/10.1145/1835804.1835873

[62] More google cluster data. 2011. https://ai.googleblog.com/2011/11/more-google-cluster-data.html. Accessed: 2025-06-28.

[63] Abdullah Mueen, Suman Nath, and Jie Liu. 2010. Fast approximate correlation for massive time-series data. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 171–182.

[64] Hussain Nizam, Samra Zafar, Zefeng Lv, Fan Wang, and Xiaopeng Hu. 2022. Real-time deep anomaly detection framework for multivariate time-series data in industrial iot. *IEEE Sensors Journal* 22, 23 (2022), 22836–22849.

[65] Open TSDB 2024. http://opentsdb.net/. Accessed: 2025-06-28.

[66] Gennady Pekhimenko, Chuanxiong Guo, Myeongjae Jeon, Peng Huang, and Lidong Zhou. 2018. TerseCades: Efficient Data Compression in Stream Processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 307–320. https://www.usenix.org/conference/atc18/presentation/pekhimenko

[67] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. Gorilla: a fast, scalable, in-memory time series database. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1816–1827. https://doi.org/10.14778/2824032.2824078

[68] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. 2013. DB2 with BLU acceleration: so much more than just a column store. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 1080–1091. https://doi.org/10.14778/2536222.2536233

[69] Vijayshankar Raman and Garret Swart. 2006. How to wring a table dry: entropy compression of relations and querying of compressed relations. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (Seoul, Korea) *(VLDB '06)*. VLDB Endowment, 858–869.

[70] P. Ratanaworabhan, Jian Ke, and M. Burtscher. 2006. Fast lossless compression of scientific floating-point data. In *Data Compression Conference (DCC'06)*. 133–142. https://doi.org/10.1109/DCC.2006.35

[71] Galen Reeves, Jie Liu, Suman Nath, and Feng Zhao. 2009. Managing massive time series streams with multi-scale compressed trickles. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 97–108. https://doi.org/10.14778/1687627.1687639

[72] Sean Rhea, Eric Wang, Edmund Wong, Ethan Atkins, and Nat Storer. 2017. Littletable: A time-series database and its uses. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 125–138.

[73] Kunihiko Sadakane. 2003. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms* 48, 2 (Sept. 2003), 294–313. https://doi.org/10.1016/S0196-6774(03)00087-7

[74] Yasushi Sakurai, Yasuko Matsubara, and Christos Faloutsos. 2015. Mining and forecasting of big time-series data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 919–922.

[75] Arnaldo Sgueglia, Andrea Di Sorbo, Corrado Aaron Visaggio, and Gerardo Canfora. 2022. A systematic literature review of IoT time series anomaly detection solutions. *Future Generation Computer Systems* 134 (2022), 170–186.

[76] Xinyang Shen, Xiaofei Liao, Long Zheng, Yu Huang, Dan Chen, and Hai Jin. 2024. ARCHER: a ReRAM-based accelerator for compressed recommendation systems. *Frontiers of Computer Science* 18, 5, Article 185607 (2024). https://doi.org/10.1007/s11704-023-3397-x

[77] Eugene Siow, Thanassis Tiropanis, Xin Wang, and Wendy Hall. 2018. Tritandb: Time-series rapid internet of things analytics. *arXiv preprint arXiv:1801.07947* (2018).

[78] Julien Spiegel, Patrice Wira, and Gilles Hermann. 2018. A Comparative Experimental Study of Lossless Compression Algorithms for Enhancing Energy Efficiency in Smart Meters. In *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*. 447–452. https://doi.org/10.1109/INDIN.2018.8471921

[79] TPCx-IoI. 2024. https://www.tpc.org/tpcx-iot/. Accessed: 2025-06-28.

[80] Chen Wang, Jialin Qiao, Xiangdong Huang, Shaoxu Song, Haonan Hou, Tian Jiang, Lei Rui, Jianmin Wang, and Jiaguang Sun. 2023. Apache IoTDB: A Time Series Database for IoT Applications. *Proc. ACM Manag. Data* 1, 2, Article 195 (June 2023), 27 pages. https://doi.org/10.1145/3589775

[81] Peng Wang, Haixun Wang, and Wei Wang. 2011. Finding semantics in time series. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 385–396.

[82] Zhiqi Wang, Jin Xue, and Zili Shao. 2021. Heracles: an efficient storage model and data flushing for performance monitoring timeseries. *Proc. VLDB Endow.* 14, 6 (Feb. 2021), 1080–1092. https://doi.org/10.14778/3447689.3447710

[83] Welch. 1984. A Technique for High-Performance Data Compression. *Computer* 17, 6 (1984), 8–19. https://doi.org/10.1109/MC.1984.1659158

[84] Jinzhao Xiao, Yuxiang Huang, Changyu Hu, Shaoxu Song, Xiangdong Huang, and Jianmin Wang. 2022. Time series data encoding for efficient storage: a comparative analysis in Apache IoTDB. *Proc. VLDB Endow.* 15, 10 (jun 2022), 2148–2160. https://doi.org/10.14778/3547305.3547319

[85] Yang Yang, Qiang Cao, and Hong Jiang. 2019. EdgeDB: An efficient time-series database for edge computing. *IEEE Access* 7 (2019), 142295–142307.

[86] Zhongguo Yang, Irshad Ahmed Abbasi, Fahad Algarni, Sikandar Ali, and Mingzhu Zhang. 2021. An iot time series data security model for adversarial attack based on thermometer encoding. *Security and Communication Networks* 2021, 1 (2021), 5537041.

[87] Zehai Yang and Shimin Chen. 2023. MOST: Model-Based Compression with Outlier Storage for Time Series Data. *Proc. ACM Manag. Data* 1, 4, Article 250 (dec 2023), 29 pages. https://doi.org/10.1145/3626737

[88] Yuanyuan Yao, Lu Chen, Ziquan Fang, Yunjun Gao, Christian S. Jensen, and Tianyi Li. 2024. Camel: Efficient Compression of Floating-Point Time Series. *Proc. ACM Manag. Data* 2, 6, Article 227 (Dec. 2024), 26 pages. https://doi.org/10.1145/3698802

[89] Xinyang Yu, Yanqing Peng, Feifei Li, Sheng Wang, Xiaowei Shen, Huijun Mai, and Yue Xie. 2020. Two-Level Data Compression using Machine Learning in Time Series Database. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1333–1344. https://doi.org/10.1109/ICDE48307.2020.00119

[90] Yuchen Yuan, Xiaoyue Feng, Bo Zhang, Pengyi Zhang, and Jie Song. 2024. JAPO: learning join and pushdown order for cloud-native join optimization. *Frontiers of Computer Science* 18, 6, Article 186614 (2024). https://doi.org/10.1007/s11704-024-3937-z

[91] Aoqian Zhang, Shaoxu Song, Jianmin Wang, and Philip S Yu. 2017. Time series data cleaning: From anomaly detection to anomaly repairing. *Proceedings of the VLDB Endowment* 10, 10 (2017), 1046–1057.

[92] Feng Zhang, Lin Yang, Shuhao Zhang, Bingsheng He, Wei Lu, and Xiaoyong Du. 2020. FineStream: Fine-Grained Window-Based Stream Processing on CPU-GPU Integrated Architectures. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 633–647. https://www.usenix.org/conference/atc20/presentation/zhang-feng

[93] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Wenguang Chen. 2018. Zwift: A Programming Framework for High Performance Text Analytics on Compressed Data. In *Proceedings of the 2018 International Conference on Supercomputing* (Beijing, China) *(ICS '18)*. Association for Computing Machinery, New York, NY, USA, 195–206. https://doi.org/10.1145/3205289.3205325

[94] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. 2022. POCLib: A High-Performance Framework for Enabling Near Orthogonal Processing on Compression. *IEEE Transactions on Parallel and Distributed Systems* 33, 2 (2022), 459–475. https://doi.org/10.1109/TPDS.2021.3093234

[95] Feng Zhang, Chenyang Zhang, Jiawei Guan, Qiangjun Zhou, Kuangyu Chen, Xiao Zhang, Bingsheng He, Jidong Zhai, and Xiaoyong Du. 2025. Breaking the Edge: Enabling Efficient Neural Network Inference on Integrated Edge Devices. *IEEE Transactions on Cloud Computing* (2025). https://doi.org/10.1109/TCC.2025.3559346

[96] Yu Zhang, Feng Zhang, Hourun Li, Shuhao Zhang, and Xiaoyong Du. 2023. CompressStreamDB: Fine-Grained Adaptive Stream Processing without Decompression. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 408–422. https://doi.org/10.1109/ICDE55515.2023.00038

[97] Yu Zhang, Feng Zhang, Hourun Li, Shuhao Zhang, Xiaoguang Guo, Yuxing Chen, Anqun Pan, and Xiaoyong Du. 2024. Data-Aware Adaptive Compression for Stream Processing. *IEEE Transactions on Knowledge and Data Engineering* 36, 9 (2024), 4531–4549. https://doi.org/10.1109/TKDE.2024.3377710

[98] Xin Zhao, Jialin Qiao, Xiangdong Huang, Chen Wang, Shaoxu Song, and Jianmin Wang. 2024. Apache TsFile: An IoT-Native Time Series File Format. *Proc. VLDB Endow.* 17, 12 (Nov. 2024), 4064–4076. https://doi.org/10.14778/3685800.3685827

[99] Yu Zheng, Xiuwen Yi, Ming Li, Ruiyuan Li, Zhang Shan, Eric Chang, and Tianrui Li. 2015. Forecasting Fine-Grained Air Quality Based on Big Data. *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2015). https://api.semanticscholar.org/CorpusID:12440971

[100] Yunyue Zhu and Dennis Shasha. 2003. Query by humming: a time series database approach. In *Proc. of SIGMOD*. 675.