International Journal of Software and Informatics, ISSN 1673-7288
http://www.ijsi.org, ijsi@iscas.ac.cn, +86-10-62661048
IJSI, 2025, 15(1): 87–113, doi: 10.21655/ijsi.1673-7288.00344
©2025 by Institute of Software, Chinese Academy of Sciences. All rights reserved.

International Journal of Software and Informatics

Research Article

Short Time Series Group Compression and Merging Method in Apache TsFile

Xingyu Liu (刘星宇)[1], Shaoxu Song (宋韶旭)[1,2,3], Xiangdong Huang (黄向东)[1,2,3], Jianmin Wang (王建民)[1,2,3]

[1] (School of Software, Tsinghua University, Beijing 100084, China)

[2] (National Engineering Research Center for Big Data Software (Tsinghua University), Beijing 100084, China)

[3] (Beijing National Research Center for Information Science and Technology (Tsinghua University), Beijing 100084, China)

Corresponding author: Shaoxu Song, sxsong@tsinghua.edu.cn

Abstract Time series data are widely used in industrial manufacturing, meteorology, electricity, automotive, and other fields, which has promoted the development of time series database management systems. More and more database systems are migrating to the cloud, and the architecture of terminal-edge-cloud collaboration is becoming more common, leading to increasingly large data scales to be processed. In scenarios such as terminal-edge-cloud collaboration and massive time series, a large number of short time series are generated due to short synchronization cycles, frequent data flushing, and other reasons, posing new challenges to database systems. Effective data management and compression methods can significantly improve storage performance, enabling database systems to handle the storage of massive time series. Apache TsFile is a columnar storage file format designed for time series scenarios, playing an important role in database management systems such as Apache IoTDB. In this paper, we propose the group compression and merging methods to be used in Apache TsFile to address the scenario of a large number of short time series, especially in applications with a large number of time series such as industrial Internet of Things. This group compression method fully considers the data characteristics in the short time series scenario, improves the utilization of metadata through device grouping, reduces file index size and short time series, and significantly improves compression efficiency. After validation with real-world datasets, our grouping method shows significant improvements in compression efficiency, reading, writing, file merging, and other aspects, enabling better management of TsFiles in scenarios with short time series.

Keywords data compression; time series data; database; industrial Internet of Things (IIoT)

Citation Liu XY, Song SX, Huang XD, Wang JM. Short time series group compression and merging method in Apache TsFile, *International Journal of Software and Informatics*, 2025, 15(1): 87–113. http://www.ijsi.org/1673-7288/344.htm

This is the English version of the Chinese article “Apache TsFile 中的短时间序列分组压缩及合并方法. 软件学报, 2025, 36(3): 941–961. doi: 10.13328/j.cnki.jos.007277”

Funding items: National Key R&D Program of China (2021YFB3300500); National Natural Science Foundation of China (62232005, 62021002, 62072265, 92267203); Science and Technology Project of the Headquarter of State Grid Corporation of China (5700-202435261A-1-1-ZN)

Received 2024-05-27; Revised 2024-07-16, 2024-08-19; Accepted 2024-08-29; IJSI published online 2025-03-31

Time series data are a data series composed of values collected for certain indicators or measurements that change over time. It has a wide range of applications in industrial manufacturing, meteorology, electricity, automotive, and other fields. In these application scenarios, time series data, characterized by high-frequency collection and continuity, form massive data streams. These data require not only real-time processing capabilities but also efficient storage to meet the needs of real-time monitoring and historical analysis.

Terminal-edge-cloud collaboration[1] is an important trend in the development of database systems in recent years. With the popularity of Internet of Things (IoT) devices and the maturity of cloud computing technology, increasing applications are choosing to distribute data collection, processing, storage, and analysis tasks between local terminal devices and cloud data centers. This terminal-edge-cloud collaboration model can fully utilize the computing and storage capabilities of terminal devices, reduce the burden on cloud servers, and improve the real-time performance of data processing.

Massive time series scenarios[2] are becoming increasingly common. As the number of IoT devices keeps growing, the data scale in various application scenarios is becoming increasingly larger, and the number of time series that database management systems need to process is also increasing, reaching hundreds of millions. The difficult management and storage due to the large number of time series is called the massive series problem or high cardinality problem. As the number of users and stored time series keeps growing, the management and storage of massive time series has become a problem of great practical significance.

In terminal-edge-cloud collaboration scenarios, the terminal/device, edge side, and cloud side need to collaborae to complete tasks. To ensure real-time data performance, the data collected by the terminal/device is integrated by the edge side and needs to be regularly synchronized to the cloud side, and the synchronization period may be very short. The amount of data collected in a short period is small, resulting in a large number of short time series, which are referred to as short series. Correspondingly, long time series refer to long series. In the scenarios involving massive time series, there may also be a large number of short series. This is mainly due to the large number of time series that each distributed node in the database system needs to process in a short period of time, which can lead to frequent data flushing and a large number of fragmented files. Short series can reduce the storage and management efficiency of database systems, posing new challenges to cloud-native database systems.

Facing the huge demand for time series applications, time series databases represented by Apache IoTDB[3] provide excellent solutions for managing time series data. Apache IoTDB is a time series database designed for industrial IoT applications, integrating data collection, storage, and analysis to meet the needs of massive storage, high-speed reading, and complex analysis in industrial IoT and other application scenarios[4]. Apache TsFile[5], the main storage format of Apache IoTDB, is a columnar storage file format designed specifically for time series data. It supports efficient compression, high read/write throughput, and is compatible with multiple frameworks[6].

The main challenges faced by existing database file formats in short series scenarios can be summarized as follows.

(1) Time series data are stored as short series scattered on disks, which not only leads to a large amount of redundant metadata information but also slows down the query efficiency of time series.
(2) Due to unclear features such as periodic changes and repetitive patterns, the compression efficiency of short series is generally low.
(3) Each short series requires independent index information, which occupies a large amount of storage space.

We propose a group compression method in Apache TsFile specially designed for short time series scenarios to address the above challenges. Apache TsFile is currently a key storage format in IoTDB, using columnar storage to manage data and placing file indexes at the end of files to accelerate queries. TsFile uses time series as the basic unit for organizational management, storing data in pages. This non-group management mode performs well for long time series. In short series scenarios, the storage length of the vast majority of short series does not exceed one data page, and the compression performance is poor, which result in a large number of short data pages in the file and low space utilization. With our group compression method, TsFile significantly improves the storage efficiency of short series. Our group compression method mainly includes the following techniques: obtaining device groups by grouping devices and managing storage based on device groups; establishing a two-layer index structure by reusing the same metadata information based on the similarity of time series within the device group and using the two-layer index structure to reduce index storage overhead; concatenating short series into long series by uniformly managing the time series within the device group, thus significantly improving data compression performance.

The main contributions of this paper are described as follows.

(1) Based on real application scenarios, we analyze the characteristics of short series application scenarios and propose a group compression method for addressing this problem.
(2) We design and implement a group compression method in Apache TsFile, which stores short time series with similar features in a centralized manner, significantly improving the storage performance of Apache TsFile in short series scenarios.
(3) We concatenate and store time series with the same measurement name based on the similarity of short time series to reduce the data compression ratio.
(4) According to the similarity of metadata in short series scenarios, we extend the original one-layer index structure of TsFile to a two-layer structure, reducing the storage of redundant metadata and the storage overhead of file indexes.

In this article, we introduce the relevant work in data storage, data indexing, and short series compression in Section 1. The theoretical concepts and basic knowledge are presented in Section 2. Details the group compression method in Apache TsFile are shown in Section 3. We introduce the scenarios with a large number of short time series and analyze the application of the group compression method in these scenarios in Section 4, evaluate and validate the performance and advantages of the proposed method in Section 5. And finally we summarize the entire work in Section 6.

1 Relevant Work

Short time series scenarios bring difficulties such as inefficient data storage and slow query to database systems. In this work, we aim to solve the inefficient storage in two directions: optimizing the storage of metadata information and indexes and improving data storage efficiency through compression.

1.1 Database system and its index

How to build indexes for data tables in files has always been an important issue that database management systems need to address, and how to design the index structure is also crucial. Generally, using indexes can quickly locate the position of data in a file, thereby efficiently performing data reading and writing. In short series scenarios, as the data volume and time series scale increase, the number of indexes also increases, and thus the storage burden of indexes keeps growing.

Apache TsFile places the index of each time series at the end of a file, which allows quick reading of the index information of each series and quick positioning of data blocks. An index can be abstractly viewed as a pair, storing the names of time series and integer pointers representing the positions of file data blocks. As the number of data points in the time series increases, a single index cannot quickly locate the data that needs to be queried, and pointers recording the location of the data will also increase. In the scenarios having massive time series, the number of indexes will increase linearly with the number of time series, which results in a surge in the storage overhead of indexes. In the real scenarios we investigated, the storage proportion of indexes may even reach up to 70%. In this article, we propose a group compression method compatible with the TsFile format, which can reduce the storage burden of indexes by grouping and building second-layer indexes, while ensuring the efficiency of indexes.

InfluxDB[7] is a popular open-source time series database that uses a time structured merge tree as storage engine[8] and it is constantly iterated. The Time Series Index (TSI) of InfluxDB adopts a multi-level indexing strategy. The core idea of TSI is to divide time series data into blocks based on time slots, with each block containing a set of time series data points and corresponding to an index file for quick searching and accessing of the data. However, like TsFile, in this indexing mode, the index size increases linearly with the number of time series, and faces problems such as index size expansion and decreased storage and query performance in the scenarios with massive time series.

To solve the indexing problem, InfluxDB proposed InfluxDB IOx[9]. InfluxDB IOx also uses columnar storage, which has better compression and query performance than row-based storage. In addition, InfluxDB IOx introduces a storage and query strategy based on data partitioning, which divides data into time ranges and other dimensions to improve query performance and reduce index size. Furthermore, InfluxDB IOx adopts a more flexible and scalable index structure, which can better handle the storage and query requirements of large-scale time series data.

Apache HoraeDB[10] is a high-performance, distributed, cloud-native time series database. The basic design concept of HoraeDB is to adopt a hybrid storage format and corresponding query methods to achieve better performance in both processing time series and analyzing scenarios. In classic time series databases, tags or labels are usually indexed in a reverse order, which allows quick positioning of time series based on tags. However, in some scenarios, the cardinality of tags can be very large. At the beginning of the design, HoraeDB aimed to solve the main problem of high indexing overhead in the case of high tag cardinality. HoraeDB used the idea of scanning and pruning in analytical databases to accelerate data indexing.

1.2 Data encoding and compression

In the process of storing time series, encoding and compression methods can significantly reduce storage burden, and thus various forms of encoding and compression are often employed in the storage of time series information. Refs. [11, 12] evaluated the effects of various encoding and compression methods in Apache TsFile and analyzed the changes in encoding effects when time series presented different features. However, they did not analyze the impact of short time series on encoding effects. When a piece of data is scattered and stored in multiple short series, the encoding effects often significantly decrease, which can be explained by the following two reasons. (1) Encoding methods generally utilize the periodic changes and repetitive pattern features in time series, which are less obvious in short series. (2) Each short series needs to separately store the features extracted by the encoding method, which introduces additional overhead. Therefore, the overall encoding and compression effects of short time series generally decrease, and the compression ratio increases. TsFile supports multiple encoding methods[12], such as run-length encoding (RLE)[13], differential encoding, SPRINTZ encoding[14], dictionary

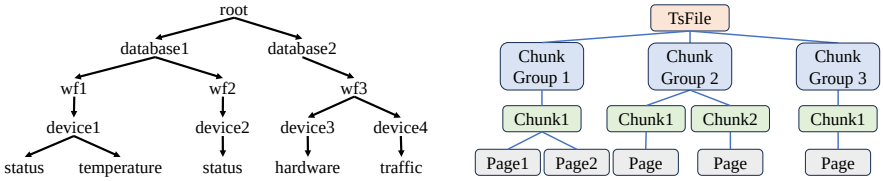
encoding[15], Gorilla encoding[16], and CHIMP encoding[17], all of which are affected by the above reason. Taking the most easily understandable dictionary encoding as an example, when several data points are stored in multiple short series, each short series needs to store a piece of dictionary information and cannot reuse the same key value pairs in the dictionary, which results in a significant decrease in efficiency. In this article, we concatenate time series with the same features within a group to ensure that the compression method can handle a sufficient number of data points, optimizing the overall compression ratio without changing the encoding method.

In addition, some researchers have proposed grouping time series to achieve efficient storage. Fang *et al.*[18] proposed a heuristic algorithm to analyze the similarity of data point distributions in different time series and stored similar time series in a centralized manner to improve storage efficiency. In our method, we focus more on the timestamp similarity of specific time series and can reuse timestamps to reduce storage burden. In addition, Fang *et al.*[19] developed a multi-column compression method, aiming at solving the problem of storage engine space amplification in load-intensive scenarios. The group compression method in our article focuses more on the similarity of metadata such as time series types and names, extracts common features of time series in short series scenarios, and stores time series with common features in a centralized manner to reduce redundant metadata and indexes. The concatenation of short series reduces the data compression ratio and improves the storage efficiency.

2 Basic Knowledge

2.1 Data storage model of TsFile

In Apache TsFile, data are mainly organized and managed in a tree structure[5]. As shown in Figure 1(a), data is stored in a tree structure in Apache TsFile. Starting from the root, each layer from top to bottom consists of a root, several databases, an intermediate node layer, a device layer, and a measurement layer. According to the user's needs, there can be no intermediate nodes or multiple ones. The unique path from the root to the leaf node uniquely represents a time series. For example, the path root.database1.wf1.device1.temperature represents a time series, with the database root.database1, the device root.database1.wf1.device1, and the measurement name of temperature. This time series may store the temperature data collected by a device in a wind power plant. Users can customize a tree structured model based on specific usage scenarios to manage the collected time series data[4].



(a) Logic structure of data in TsFile (b) Storage structure in TsFile

Figure 1 Storage logic and structure of time series in TsFile

The disk storage structure of Apache TsFile corresponds logically to the logical structure of the data. Figure 1(b) shows the storage structure of data in TsFile. A TsFile contains multiple chunk groups, each responsible for storing information about a device, including several chunks. Each chunk stores a time series of data, consisting of metadata information and several pages. The time series in each page is stored in a columnar format, including two columns: a timestamp column and a value column. The two columns are encoded, compressed, and stored separately, thus being separate. A page is the basic unit for TsFile data reading and writing, and it generally requires a complete read of the entire page to parse the data stored within it. However, pages,

chunks, and chunk groups all contain data summaries that describe the characteristics of the stored data, such as the maximum and minimum values of data points, start and end time points, and other information. In this way, during the query process, data summaries can be used for filtering, and the pages that do not need to be read can be skipped.

2.2 Data encoding and compression

Data encoding is related to the efficiency and quality of data storage, and it is a method of reducing storage space usage while ensuring data integrity. In the process of data storage, it is necessary to encode the data, transforming the data from a human readable form to a form that is easy for computers to classify, sort, and compress. That is, a certain algorithm is used to serialize data into a binary byte stream, which usually occupies less space than the original data representation, thus achieving the compression effect. Data compression is the process of reducing data redundancy to save storage space or transmission bandwidth. It can be divided into lossy compression and lossless compression based on whether information is lost.

In the encoding compression method of Apache TsFile, the timestamp column and value column of time series are stored separately. This separated storage logic enables the use of targeted encoding and compression methods for the value column and the isolation and non-interference between different time series. In addition, separating the timestamp column enables targeted encoding and compression of timestamps. For example, using second-order differential encoding to encode timestamps improves the efficiency. However, this separated storage strategy faces challenges in the scenarios involving massive short time series. Since time series are stored separately, even if they have similar metadata (e.g., the same type, encoding method, and actual physical meaning), they still need to be stored separately, which brings much overhead in scenarios with massive time series.

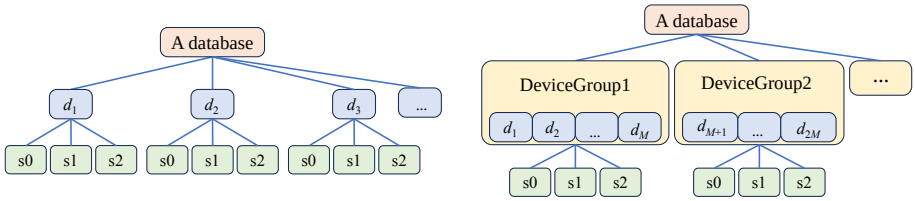
3 Group Compression Method in Apache TsFile

We find that scenarios involving short series have some features by analyzing real-world datasets provided by corporate partners. In these scenarios, the metadata information of time series is regular and similar, which is determined by the characteristics of the real world. In general, devices in application scenarios are standardized and streamlined, and thus there is a large number of devices responsible for the same function, generating the same type of time series data, such as temperature and humidity monitored by meteorological collection devices, motor rotation rates of wind power plants, engine speeds during vehicle operation. The time series collected by devices with the same function may have similar or identical features, such as names, measurement types, and numerical ranges. Therefore, many short series share the same metadata information but are isolated during storage, which results in wasted space.

We propose a group compression method for scenarios involving short series. We can fully utilize duplicate metadata information and improve data storage efficiency with our method. Here we introduce the group compression method from the designs of the model, compression method, and index structure.

3.1 Modeling and the group method

The group compression method adopts a tree model in TsFile for modeling, while modifying the organization of devices. Figure 2 illustrates the storage structures of the non-group method and the group compression method. Integrating information from multiple devices into the same node reduces the number of nodes in the tree structure, thereby reducing storage burden. Meanwhile, each leaf node in the tree structure represents a storage unit of time series on a disk. As the tree structure is pruned, the storage of time series becomes more organized, which reduces the storage burden of indexes.



(a) Storage structure of the non-group method (b) Storage structure of the group compression method

Figure 2 Models of device groups in TsFile

As shown in Figure 2(a), there are several devices in the storage model of the non-group method in TsFile, and the measurement of each device is a leaf node of the tree, corresponding to a time series. Each device is independent of each other and the information of time series is also managed separately. However, according to our survey on actual user scenarios, a large number of devices have similar functions and consistent information of time series. Therefore, we optimize the organizational management of the devices in a targeted manner. As shown in Figure 2(b), we classify several devices into a group, i.e., a device group. The upper limit of the number of devices within a device group is the size of the device group, also known as parameter M in the graph. A device group contains several different devices which share identical measurement information. In this way, devices within a device group do not need to store the same information multiple times but share measurement information, which reduces redundant metadata.

For multiple devices that need to be grouped, we adopt a simple grouping method, which is composed of two steps: (1) analyzing the measurement information of each device and putting devices with completely consistent measurement information into the same set; (2) dividing each set into several device groups based on the size of the device group, i.e., the upper limit of the number of devices in the group. In this way, in a device group, time series with the same measurement name for all devices have completely consistent metadata information and can share duplicate metadata during storage to reduce the storage of redundant metadata.

This group compression method needs to address some problems. The first is the data storage problem, i.e., how to store time series with the same measurement name for multiple devices. The second is the data indexing problem, i.e., how to confirm which device the stored fragment of time series belongs to before grouping. We introduce the compression method in Section 3.2 to address the data storage problem and the two-layer indexing method in Section 3.3 to solve the data indexing problem.

3.2 Concatenation of short series for compression

In our group model, several similar devices are grouped together for unified management. Because of the group method, the measurements with the same name in a device group have good properties and identical features, including data type, encoding method, and compression method. However, their timestamps may not be aligned. Considering these properties, we propose a compression method for time series by concatenation to further improve storage efficiency and address the low compression efficiency of short series.

As shown in Figure 3, two devices, namely device1 and device2, need to be stored. Both devices have two time series with the same measurement names: s1 and s2. In the non-group storage model, four time series corresponding to eight arrays need to be stored, and their metadata information is not shared with each other, with storage units isolated from each other. In our group compression method, TsFile can recognize the similarity between the two devices and place them in the same storage group (group1). We concatenate measurements with the same

name to compress the time series in this device group. The specific operation for concatenating and storing time series with the same measurement name is as follows:

(1) selecting a measurement with the same name, such as s1;
(2) traversing both devices (device1 and device2) in the device group in sequence and retrieving the timestamp and value columns of the measurement;
(3) concatenating the timestamp column and the value column separately according to the traversal order;
(4) applying corresponding encoding or compression methods to s1 and s2 when storing to TsFile.

For example, s1 with a 32-bit signed integer data type can be encoded by the default second-order differential encoding and compressed by the LZ4[20] method, while s2 with a 64 bits floating-point data type can be encoded by the default Gorilla[16] method and then compressed by the LZ4 method. According to this process, all time series with the same measurement name can be concatenated. In this way, the time series with the same measurement name for multiple devices are concatenated into one time series within the device group. Moreover, this group compression method does not change the encoding and compression methods used from time series, but rather transforms the encoding and compression process of multiple short series isolated from each other into that of long series.

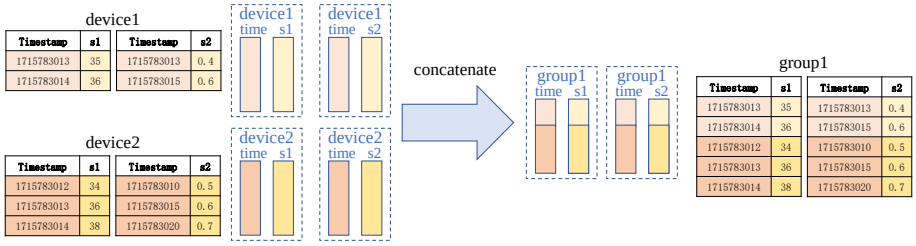


Figure 3 Concatenation of time series with the same measurement name in the device group

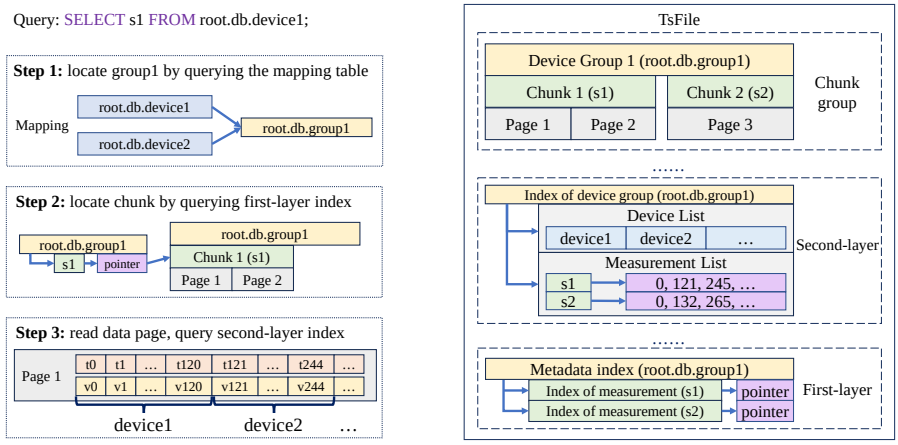
In scenarios with a large number of time series and frequent occurrence of short series, this group compression method can effectively reduce the actual number of stored time series units and improve the encoding and compression efficiency by concatenating short series into long series. The concatenation of time series with the same measurement name solves the difficulty of the high compression ratio of data pages in the scenarios of massive time series. Due to the concatenation of multiple short series into long series, the compression method is more likely to identify repetitive patterns in value or timestamp columns, which can reduce the compression ratio and improve the storage efficiency. In addition, the method proposed in this paper can fully utilize the storage structure of TsFile. For the storage engine of TsFile, the main information for storing a time series is the metadata information of time series (name, data type, etc.), timestamp column, and value column. After concatenation of time series with the same measurement name, a new time series can be obtained, which contains the main information of these three elements and can be directly stored by the storage engine of TsFile. It naturally requires storing additional information for this method. For example, in the example of Figure 3, if only two concatenated time series are known, it is impossible to restore the four time series before concatenation. Therefore, we extend the indexing method to locate which device each row of data belongs to. This part is detailed in Section 3.3.

3.3 Two-layer index structure and query rewriting

After group compression, the same information of the devices in TsFile is fully utilized, while the corresponding independence is also lost. In the query framework that is originally not

grouped in TsFile, the storage of different devices is isolated from each other, and the storage of chunks can be quickly located by indexing. In the group compression method we propose, multiple devices share chunks and require additional information to assist in localization. Therefore, we extend the original one-layer index structure to a two-layer index structure and design a query rewriting scheme to ensure the effectiveness of the group compression method. In this section, we introduce the query rewriting scheme, elaborate on the specific implementation of the index structure, and analyze the reasons for the success of this method.

The query rewriting scheme runs mainly relying on two types of data structures: one is a mapping table of the device-to-device group contained in the file, which can quickly locate the device group where the device is; the second is a two-layer index structure, where the first-layer index can locate the position of chunks, and the second-layer index can locate the position of data points in pages. We illustrate a complete query example in Figure 4(a).



（a） Query flow （b） File structure of TsFile

Figure 4 Query rewriting scheme and extended file structure of TsFile

In the example in Figure 4(a), the user wants to query the measurement s1 of root.db.device1 and sends the corresponding SQL statement. After the query statement is received, the first step is to locate the device, that is, to determine which device group device1 is located in. The application of the mapping table enables the quick location of the device group where device1 is located as root.db.group1. In this way, the query originally targeting root.db.device1 is redirected to the device group root.db.group1. The second step is to find the storage location of the device group. To achieve this, it is necessary to use a second-layer index to locate the corresponding time series. In this example, the measurement s1 needs to be queried. By searching for the file pointer via the first-layer index, the position of the time series root.db.group1.s1 in the device group can be quickly found, and its chunk can be obtained. After the chunk is obtained, the data pages within it can be read. The third step is to locate the data points of the device being searched in the data page. The data page contains two columns, one is the timestamp column t, and the other is the value column v. Through second-layer indexing, it can be confirmed that the device group group1 contains several devices: device1 and device2. It is also possible to determine all measurements within the device group, including s1 and s2. The index of the data point corresponding to s1 can be found. In this example, the index of the found data point is $\{0, 121, 245, \ldots\}$, which represents the index of the first data point of each device. For example, the data point index of device1 is 0, and that of device2 is 121. Therefore, it can be determined that the data points of device1 are from (t0, v0) to (t120, v120). In a similar way, the data point

positions of each device can be determined. On the contrary, when any data point is read, the device to which it belongs can also be found based on the index of that data point. Finally, by processing the 121 data points, the query results can be obtained.

To achieve this query rewriting, we need to improve the file structure of TsFile to support the two data structures mentioned above. The mapping table can be reconstructed based on the second-layer index, and thus it does not need to be stored in a file, and only the second-layer index needs to be stored. We save the added second-layer index data in TsFile Metadata, as shown in Figure 4(b). The second-layer index contains the structural information of all device groups in the file, storing a list of devices within the group and a list of measurements within the group, with each measurement pointing to a data point index. In this way, the index structure provides two layers under our group compression method. The first layer is the index of each time series in the device group, which relies on the native index structure of TsFile. The second layer is the index list of data points for each measurement in the device group. By using these two layers of indexes, we can quickly locate the position of a certain segment of data in any time series in the file.

The group encoding method divides the index structure into two layers without reducing the number of indexes. However, it reduces the storage overhead, mainly because it reduces the data storage content of strings with more bytes. We confirm this result through formula derivation, and the symbols used and their definitions are listed in Table 1. Among them, the device name, measurement name, and index pointer are represented by d, s, and p, respectively. The complete path of a time series is obtained by concatenating the device name and measurement name with an English period, that is, $d.s$. We use bytes to measure the mean length occupied by storing each variable, which is represented by the symbol L. N_d and N_s represent the number of devices and the number of measurements, respectively. The symbol M represents the size of the device group, which is the upper limit of the number of devices within the device group.

Table 1 Definitions of symbols in equations

Variable	Symbol	Definition
Name of device	d	A string describing a device path
Measurement name	s	A string describing a measurement name path, and $d.s$ refers to a complete path
Index pointer	p	An integer representing the location of data in a file
Mean length of device name	L_d	Mean number of bytes of the device name string
Mean length of measurement name	L_s	Mean number of bytes of the measurement name string
Mean length of index pointer	L_p	Number of bytes of index pointer
Size of device group	M	Maximum devices in each device group
Total number of devices	N_d	Number of devices in a file
Number of measurements with the same name	N_s	Number of measurements for each device

The index of time series generally includes two parts. One is the complete path of time series that concatenates the device name with the measurement name to obtain $d.s$, which is stored as a string in TsFile. The second is the index pointer p, which is usually an INT-type integer that points to the position of a chunk in the file. This type of a pair $(d.s, p)$ can locate the data position of a time series in the file, and the storage overhead of the index is the total number of bytes in this pair. For the convenience of calculating the total length of the strings, the average number of bytes is uniformly used here. We assume that a file contains N_d devices, each with only N_s measurements, and use the device name, measurement name, and mean length of index pointer to measure the storage overhead of the index. Here we demonstrate that when the device group size M is sufficiently large, the indexing overhead of the group method

in real-world scenarios is lower.

Lemma 1. The index storage overhead in the non-group method is directly proportional to the number of time series, device names, measurements, and the mean length of index pointers.

A file contains N_d devices, each of which has only N_s measurements, resulting in a total of $N_d \cdot N_s$ time series. If the non-group method is used, the complete path needs to be stored for each time series, and a file pointer needs to be saved. Therefore, the total storage overhead is the number of time series multiplied by the total byte length of the index pair, which is $N_d \cdot N_s \cdot (L_d + L_s + L_P)$ bytes.

Lemma 2. In the case of $N_s > 1, M > 2$, the storage overhead of device names and measurement names in the group method is lower than that in the non-group method. The index pointer storage overhead of the group method is always higher than that of the non-group method.

In the case of using the group compression method, the index has two layers and generates $\lceil N_d/M \rceil$ device groups. The first-layer index needs to record the time series of each device group, and thus the number of time series that need to be recorded is reduced. Only $\lceil N_d/M \rceil \cdot N_s$ time series need to be recorded, and the total overhead is $\lceil N_d/M \rceil \cdot N_s \cdot (L_d + L_s + L_p)$ bytes. The second-layer index needs to simultaneously record the device list, measurement list, and data point index. The total overhead of the device list and of the measurement list is $N_d \cdot L_d$ and $\lceil N_d/M \rceil \cdot N_s \cdot L_s$, respectively. The total overhead of the data point index is at most $\lceil N_d/M \rceil \cdot N_s \cdot M \cdot L_p$ bytes, and the total overhead of the second-layer index is $\lceil N_d/M \rceil \cdot N_s \cdot (M \cdot L_p + L_s) + N_d \cdot L_d$ bytes. Therefore, the total overhead of the group compression method is the sum of the two-layer index overhead, which is at most $\lceil N_d/M \rceil \cdot N_s \cdot (L_d + 2L_s + (M+1)L_p) + N_d \cdot L_d$ bytes. We summarize the storage overhead of the two index methods in Table 2. As long as the inequality $N_s \cdot M > N_s + M$ is satisfied, the storage overhead of device names in the group compression method will be lower. As long as $M > 2$ is satisfied, the storage overhead of measurement names in the group compression method will be lower. The group compression method always has more index pointers, which increases the overhead by $N_d \cdot N_s \cdot L_p/M$. According to the specific definitions of the variables, $N_s > 1$ generally holds, and in this case, the proposition $N_s \cdot M > N_s + M$ is implied by the assumption $M > 2$.

Table 2 Storage overhead of two index methods

Item	Symbol	Storage overhead of non-group method	Storage overhead of group compression method	Approximate difference
Mean length of device name	L_d	$N_d \cdot N_s \cdot L_d$	$(\lceil N_d/M \rceil \cdot N_s + N_d) \cdot L_d$	$((1 - \frac{1}{M})N_s - 1)N_d \cdot L_d$
Mean length of measurement name	L_s	$N_d \cdot N_s \cdot L_s$	$2\lceil N_d/M \rceil \cdot N_s \cdot L_s$	$(1 - \frac{2}{M})N_d \cdot N_s \cdot L_s$
Mean length of index pointer	L_p	$N_d \cdot N_s \cdot L_p$	$\lceil N_d/M \rceil \cdot N_s \cdot (M+1)L_p$	$-(\frac{1}{M} + \frac{M+1}{N_d})N_d \cdot N_s \cdot L_p$

Corollary 1. Compared with the non-group method, whether the group method reduces the index size is independent from the device number N_d.

We can observe in Table 2 that both the index overhead of the non-group method and that of the group compression method contain the positive integer N_d term. Therefore, dividing the three terms by N_d at the same time does not affect their size relationship, and thus the corollary holds.

Theorem 1. If $\left(\left(1-\frac{1}{M}\right)N_s-1\right)L_d+\left(1-\frac{2}{M}\right)N_s\cdot L_s>\left(\frac{1}{M}+\frac{M+1}{N_d}\right)N_s\cdot L_p$, then the index storage overhead of the group compression method is lower than that of the non-group method.

We can draw this conclusion from the approximate difference in Table 2. Here, the approximate difference refers to the estimation of the index overhead of the non-group method minus that of the group compression method. Since the inequality $N_d/M \leq \lceil N_d/M \rceil < N_d/M+1$ holds, we can scale the size of each index. We reduce the overhead estimate of reducing device names and measurement names and increase the overhead estimate of index pointers, and then obtain the inequality in the theorem by subtracting. In practical situations, the inequality in the theorem generally holds, which can be explained by two main reasons. (1) The constant term related to the device group size M at the left side of the inequality is larger. (2) Strings generally occupy more storage space than integers, meaning that the parameter values related to byte count at the left side of the inequality are larger. For ease of understanding, we provide a practical example in Example 1.

Example 1. Application of the group compression method and the non-group method in the estimation of the index storage overhead of a vehicle dataset. Assuming the number of devices we need to store is $N_d = 6700$ and each device has $N_s = 27$ time series, the storage of each device name, measurement name, and integer index requires an average of $L_d = 72$ bytes, $L_s = 8$ bytes, and $L_p = 4$ bytes, respectively.

(1) The non-group method requires storing indexes of $6700 \times 27 = 180900$ time series, consuming 180900×80 B $= 14472$ KB of space. The storage of index pointers requires 180900×4 B $= 724$ KB of space. The final non-group method consumes a total of 15,196 KB of space.

(2) In the case of using the group compression method, assuming that each group can store the information of up to $M = 100$ devices, there are $6700 \div 100 = 67$ device groups after merging. The first-layer index requires recording $67 \times 27 = 1809$ time series, with a string storage overhead of 1809×80 B $= 145$ KB and an index pointer storage overhead of 1809×4 B $= 8$ KB. In the second-layer index, a device list of 6,700 devices needs to be stored, with the overhead of 6700×72 B $= 483$ KB. In addition, a measurement list of 1,809 series needs to be stored, with the overhead of 1809×8 B $= 15$ KB, and the overhead of data point index pointers is $1809 \times 100 \times 4$ B $= 724$ KB. The total consumption of the group compression method is 1,373 KB of space, a reduction of 91%.

The theoretical estimate is highly consistent with the experiment results.

The group compression method can effectively reduce the index size, which is attributed to the fact that the full utilization of the same features of a large number of devices reduces the storage content of time series names. In addition, in the above derivation, we assume that each time series has only one file pointer for its first-layer index. In TsFile, the number of file pointers for each time series is related to the number of data pages, which enables the use of multiple indexes to quickly locate data in scenarios with large amounts of data. However, the analysis of real-world datasets shows that the data volume in the scenarios involving short time series is very small, generally no more than 200 lines, which means they only have one data page in the file. At the same time, when the device group size M is not too large, the concatenated long time series only has one or two data pages, and thus the actual situation is consistent with the theoretical derivation. Even in scenarios where short series are longer, the first-layer index contains multiple file pointers, and the additional pointer storage overhead is much lower than the string storage overhead reduced by the group method. Therefore, the group method can still effectively reduce the index size. However, if the stored sequence is very long and the number

of devices and time series in a TsFile is small, the inequality in Theorem 1 may not hold. In this case, the index overhead of the non-group method will be lower than that of the group method, and such scenarios without short series are not suitable for the application of the group compression method. We discuss this situation in Section 4.

4 Merging of Short Series Files

We have detailed the group compression method and explained the philosophy of model design, the compression of short series by concatenation, and how to build indexes and rewrite queries. The group compression method is applicable for the scenarios with a large number of short series, while this method requires additional processing overhead, such as the time overhead of time series grouping and the additional time overhead of query rewriting. When the database management system can merge short series into long series, the number of short series will gradually decrease, and the storage disadvantage will no longer be obvious. At this time, the group compression method is gradually no longer needed. Therefore, in this section, we provide a detailed introduction to the generation and merging process of short series files, as well as how to choose the group compression method at the appropriate time.

4.1 Generation of short series files and data synchronization

We find that there are a large number of short series in many scenarios by analyzing real-world datasets provided by corporate partners. Short series have fewer data points than long series, which results in low storage efficiency. There are various reasons for the generation of short series, in which two are common. (1) Short data synchronization cycles results in file series containing only data within a short time window. (2) In the scenario of massive time series, there are many concurrent requests and frequent data flushing, which result in a large number of short series being stored in the same file. We detail the two scenarios here.

Example 2. Terminal-edge-cloud collaboration. In practical scenarios, it is often necessary to collaborate with the terminal/device, edge side, and cloud side to complete data collection, cleaning, management, and analysis. For example, Apache IoTDB, which uses the TsFile format, manages data through a distributed structure and can be flexibly deployed on both cloud and edge sides, realizing seamless connection by cloud synchronization tools. A company has multiple factories, each with several workshops. At this time, IoTDB can be integrated into the three-layer IoT platform of the company, factory, and workshop to achieve unified device debugging and control. In this scenario, the IoT devices on the workshop side are responsible for collecting data, and IoTDB C++ clients can be deployed to collect data under limited hardware resources. The factory layer collects terminal/device data and completes cleaning work, and it can perform business analysis on the data. The cloud side can monitor and analyze factory equipment. IoT devices collect data 24 h without interruption. In order to synchronize data in a timely manner, the synchronization period may be very short, such as once every minute. At this point, the device data per minute only includes the data points within that minute in the synchronized file, which is relatively small. Therefore, the large amount of data generated within 24 hours is divided into 1,440 fragmented files. After receiving these fragmented files, the edge-side data nodes need to reorganize and merge them.

Example 3. Massive time series. In a scenario of massive time series, the number of time series can be very large. For example, an IoT company uses Apache IoTDB to record 27 indicators from over 7 million vehicles of a brand during actual operation, with a total of over 200 million time series. Despite using several distributed nodes, each node still needs to handle a large number of write requests of time series. In general, IoTDB will cache the data written to each time series in memory and flush it to the disk when the data volume is sufficient. However,

in scenarios with massive time series, especially during peak request periods, the average cache allocated to each time series is small, and thus data have to be immediately flushed to disk once the cache is full. Therefore, there are numerous time series written into the same TsFile, while each time series is very short, with an average of only 80 data points and a corresponding time span of about 4 minutes.

In Example 2 and Example 3, there are a large number of short series, while the reasons and problems caused are slightly different. In Example 2, there is a data synchronization problem. The data synchronization of TsFile is determined by the specific implementation of the database management system. For example, in Apache IoTDB, the data synchronization methods depend on the user's configuration, including two types: (1) batch synchronization, which is completed through the transfer of TsFile; (2) stream synchronization, which generally involves sending synchronization tasks, such as sending write plans to ensure the synchronization of new data, and using the method of sending TsFile to synchronize more data when there is a backlog of tasks. In brief, the basic unit of data synchronization in the TsFile format is the data itself rather than a single TsFile. The file format does not limit the methods of data synchronization, and the specific methods used for data synchronization depend on the user's configuration. In Example 2, the terminal/device side continuously generates time series data, while during synchronization, the data points are divided into multiple short series for transmission in different time periods. The edge side will receive a large number of fragmented files, which need to be sorted and merged.

In Example 3, short series are generated on the local server, and there is no timing synchronization problem, while there is a consistency maintenance problem between copies. In Example 3, the user configures two copies for each IoTDB data node, with the two data nodes serving as backup for each other. At this point, the two nodes mainly synchronize the newly added data by synchronously writing to the plan, and the synchronization process does not directly use the TsFile format. The main problem that needs to be solved in this scenario is the presence of a large number of short series in local files, which results in high storage pressure. In this scenario, when the cache is insufficient and data in memory must be flushed to disk, the database management system is unable to read historical data and integrate it into long series in a short period of time. It directly flushes time series with only a small amount of data in the cache to disk, resulting in a large number of short series in TsFile.

4.2 Merging of short series files

The merging of TsFile mainly has three objectives.

(1) Integrating out-of-order data to avoid temporary out-of-order merging and sorting during queries, thus improving query performance.
(2) Integrating data from the same time series to increase the size of chunks and avoid multiple queries and reads of chunks, thus improving query performance.
(3) Reducing the number of files, which has certain benefits.

For example, in IoTDB using the TsFile format, each file corresponds to a TsFile resource memory structure, and fewer files correspond to less memory usage. As the merging progresses, fragmented files that store a large number of short series will be merged into files containing long series in TsFile.

Similar to data synchronization, the merging of TsFiles also depends on the specific implementation of the database system. For example, the merging process of TsFiles in IoTDB can be simply divided into three stages.

(1) File selection: selecting the files that need to be merged.
(2) Merging scheduling: selecting merging tasks based on the current system state.

(3) Merging execution: reading the files to be merged, sorting, organizing, and merging the files in memory, and then writing the new files to the disk.

TsFile adopts a strategy of separating sequential and out-of-order data storage. In general, the data points stored in a time series in TsFile are in an ascending order, and there is no out-of-order data. Out-of-order data are additionally recorded in out-of-order files. In the merging of fragmented short series files, out-of-order data are not our research focus. Therefore, we mainly consider the merging of sequential files in this section. The merging of short series files here mainly refers to sequential files in TsFile, where data points in the same time series are arranged in an ascending order.

The merging process of TsFiles with the non-group method is the following:

(1) Read the files to be merged into memory and lock the corresponding files.
(2) Record the merging information, such as the type of merging, the files to be merged, and the target merged file according to the rules in the system configuration.
(3) Sort and merge different segments of the same time series in the files to be merged to obtain a new time series.
(4) Write the merged time series into the target file.
(5) Replace the TsFile resources of the files to be merged with the TsFile resources of the new file in memory, release the lock of the files to be merged, and delete the files to be merged. If necessary, the merging logs are finally deleted.

The process of merging files in TsFile with the group compression method is only different in step (3). Since the group method compresses the time series of multiple devices into the same device group, step (3) needs to be rewritten as follows: (3.1) split the time series of device groups in the files to be merged into the time series of multiple devices, which is equivalent to decompressing the concatenated long series into several short series; (3.2) sort and merge different segments of the decompressed time series to obtain a new time series; (3.3) apply the group compression method again for the merged time series.

In addition, if necessary, step (3.3) can also omit the use of the group compression method, so that several compressed fragment files in TsFile to be merged can be read and finally merged to obtain a non-group file. This method is effective when short series gradually disappear and the group method is no longer needed. We detail the application of the group method in the merging of short series in Section 4.3.

4.3 Application of the group compression method in merging of short series

As long as there are enough devices with the same measurement in terms of name and data type, redundant metadata can be fully removed through the group compression method to improve storage efficiency. This group compression method mainly reduces the storage overhead in two ways:

(1) Concatenating short series into long series to improve the compression efficiency of data pages.
(2) Establishing a second-layer index and removing redundant series metadata information from the first-layer index.

Both ways require the data stored in TsFile to have certain features. The former requires a large number of short series of the same data type to exist in the data. The latter requires a sufficient number of time series within the file and a sufficiently large device group size, meaning that there are enough devices sharing the same metadata information. Therefore, when the data no longer satisfies these two types of features, the selection should be based on the actual situation and the most efficient storage method should be adopted. The practical applications of

the group compression method in Examples 2 and 3 are shown in Figures 5 and 6, respectively.

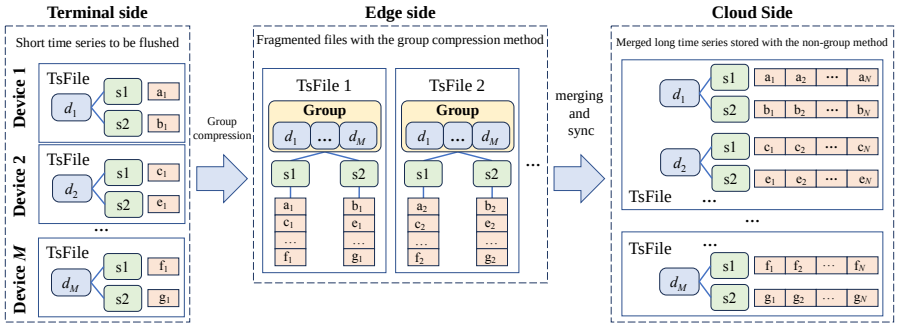


Figure 5 Application of the group compression method in the terminal-edge-cloud collaboration scenario

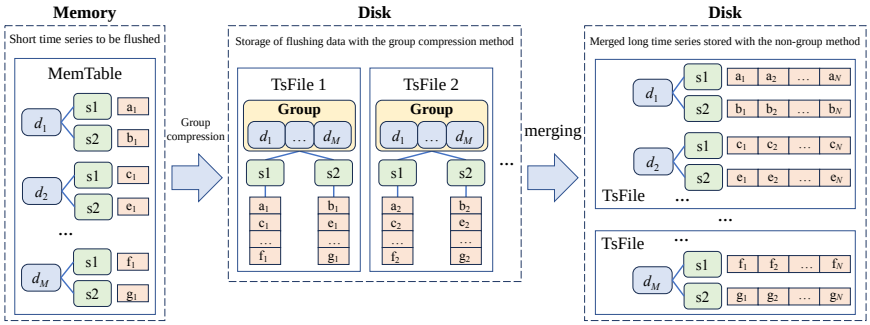


Figure 6 Application of the group compression methods in the massive time series scenario

In Example 2, we presented the terminal-edge-cloud collaboration. In this scenario, the group compression method can play a role in data synchronization and cloud side storage, as shown in Figure 5. The data synchronization at the terminal/device side generates a large number of fragmented files, each of which contains a small number of data points. At this time, the TsFile data stored with the non-group method occupies much space. After these fragmented files waiting for synchronization are sent to the edge side, efficient storage can be achieved through group compression to reduce file size. The amount of data at the edge side increases over time, and the number of short series gradually increases. At this point, file merging is necessary. The edge side can merge fragmented files into TsFiles by group compression. Each device continuously generates data over time, and these time series will gradually merge into long series at the edge side. Due to the upper limit of the size of each TsFile, after the series becomes longer, a file may no longer be able to accommodate data from multiple devices, and these devices need to be stored separately in different TsFiles. In this way, as fragmented files are merged, the time series stored in a single TsFile will gradually decrease, and the time series length will gradually increase. When it reaches a certain level, the group compression method is no longer applicable. At this point, the non-group method can be used to store long series in TsFile. Finally, the cloud side uses the non-group method to store long series of each device. In addition, the merging of fragmented files during this process may also occur at the cloud side. For example, if the newly added data at the edge side have not been merged into long series, but a synchronization request is received, the fragmented files may be synchronized to the cloud side. The cloud side may integrate the fragmented files into long series files that are stored with the non-group method at an appropriate time.

In Example 3, we presented the massive time series. In this scenario, the main reason for the generation of short series is frequent data flushing caused by insufficient memory. In response to this problem, the group compression method can play a role in data flushing and storing short series, as shown in Figure 6. When the cache is insufficient and data in memory must be flushed to the disk, the group compression method can be used to complete the writing. The number of disk writes is reduced and the amount of data written is lower too. Because the group compression method processes multiple devices at once, it can reduce the number of file write operations. At the same time, this method reduces the compression ratio and decreases the size of data pages and indexes, thereby reducing the amount of serialized data written. In addition, the memory pressure in this scenario is high, and the additional overhead introduced by the group compression method is minimal, which can effectively ensure the performance. This is because the group compression method fully utilizes the similar metadata information of time series, and thus the metadata within the device group in memory is reused. At the same time, concatenating and storing short series can greatly improve the storage efficiency. During the low period of write requests, the server has sufficient idle computing power to merge files, gradually merging short series into long series. Similar to the analysis of Example 2, the merging process of TsFile gradually reduces the number of time series in the same file and increases the length of each time series. When the length of a time series reaches a certain level, the group compression method is no longer applicable. At this point, the non-group method can be used to store long series during merging in TsFile.

Overall, the group compression method can be flexibly used based on the specific situation of short series generation. The method we proposed in this paper plays an important role in scenarios such as short series storage and data synchronization. We compare the performance of the group compression method and the non-group method regarding various indicators in detail in Section 5 and analyze the impact of series merging on the compression effect. In addition, we analyze under what circumstances we should switch from the group compression method to the non-group method, comprehensively considering the compression, query, write and other operations efficiency.

5 Experimental Analysis

In this section, we analyze the performance of the group compression method in TsFile from the following aspects:

(1) Space compression ratio.
(2) Reading performance.
(3) Writing performance.
(4) Merging performance of fragmented files.

We chose Apache IoTDB as the database management system and conducted experiments with the group compression method of Apache TsFile. The configuration used in the experiments is as follows: Intel® Cores™ i9-13900HX CPU@2.20 GHz with 24 cores, 16 GB memory, and 1 TB solid-state drive.

5.1 Experiment design

We validate the effectiveness of the proposed method on two real-world datasets. Table 3 lists the basic information of the datasets. The vehicle dataset is a large amount of data collected from vehicles of a brand during actual operation, with a total of over 7 million devices and more than 200 million time series. Its main data types are 32-bit signed integers and 64-bit floating-point numbers. In this experiment, we extract real data from some of the devices during a certain period of time. The ship dataset is composed by the operational data collected

during ship navigation testing, with the main data type being text and storing multiple navigation parameters in the JSON format. In this experiment, we extract real data from some of the devices during a certain period of time.

Table 3 Dataset for evaluations

Dataset	Mean number of devices per file	Mean number of time series per file	Total lines	Main data types
Vehicle dataset	6,772	182,844	14,901,948	INT32, DOUBLE
Ship dataset	68,018	136,036	408,108	TEXT

We use real datasets collected in various scenarios for evaluations. We organize these data through the group compression method, rewrite them in the TsFile format, and compare them with that obtained with the non-group method. Our evaluations mainly involves three aspects:

(1) File compression efficiency.
(2) File reading and writing efficiency.
(3) Time consumption and compression efficiency changes during the merging of fragmented files.

By comparing the sizes of two files, we can intuitively obtain the compression ratio of the applied method. Here, we define compression ratio as the compressed file size divided by the original file size. Therefore, this ratio is generally a number less than 1. Furthermore, we analyze in detail the proportion of each part in the file structure, which can further explain how the group compression method improves compression efficiency. Next, by comparing the file querying and writing rates of the non-group and group compression methods, we can analyze the file reading and writing efficiency of the two methods. Finally, we design an experiment merging fragmented files, in which the dataset is fragmented and stored in several fragmented files to simulate scenarios of short time series generation in TsFile. By comparing the file merging time of the non-group and group compression methods, we can analyze the efficiency of the two methods in merging fragmented files. In addition, we analyze the changes in file compression efficiency changes during the merging process, as mentioned in Section 4, which helps us analyze the appropriate time of choosing methods.

5.2 Compression efficiency of the group compression method

For each dataset, we use different device group sizes for compression and compare the file compression effect. On the one hand, this is beneficial for us to select appropriate parameters for subsequent experiments, and on the other hand, it is beneficial for analyzing the impact of device group size on the encoding efficiency of the group compression method. We not only analyze the impact of changes in device group size on the overall file but also delve into the changes in various components of the file, mainly the compression efficiency of data pages and the file index size.

Firstly, we evaluate the change in total file size after using the group compression method, with the results shown in Figure 7. We find that even if the device group size is only 2, which means compressing at most two devices in each group, the group compression method can effectively reduce the total file size. The compression ratio is used to evaluate the compression efficiency of files, and it is defined as the size of the file after application of the group compression method divided by the size of the original file. As shown in Figure 7, as the size of the device group increases, the total file size gradually decreases. When the size of the device group reaches around 100, the file compression ratio tends to become stable. In addition, the optimal compression ratio of files converges to different values on different datasets, which we believe is related to the inherent characteristics of the datasets. To explain this point and confirm how the compression efficiency of the group compression method is improved, we further analyze

the changes in data pages and indexes in the file.

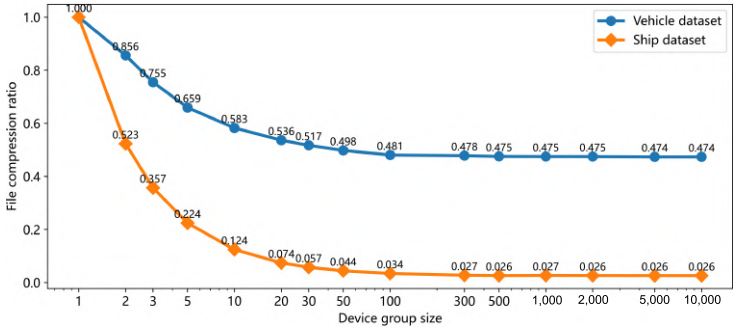


Figure 7 Compression ratio changes of files under different device group sizes

We conduct in-depth analysis from two aspects: data page compression and index compression. We analyze the total size of data pages and indexes in each file for different device group sizes. Due to the varying sizes of different datasets, we normalize the total size data to analyze the changes in compression ratio. Specifically, we divide the sizes of the grouped data pages by the size before grouping to obtain a compression ratio less than 1, and then analyze the changes in this compression ratio. Furthermore, we perform the same normalization process on the index size. The changes in the total data page size and index size are shown in Figures 8 and 9, respectively.

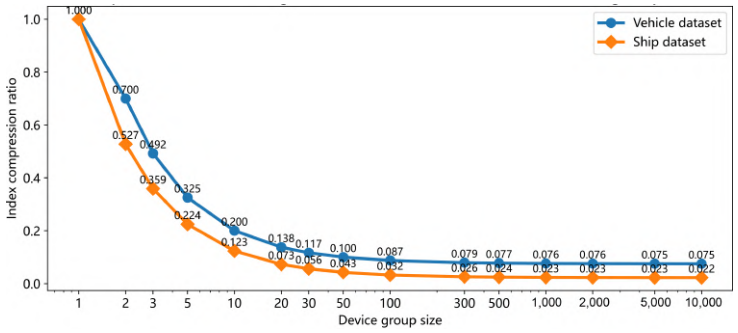


Figure 8 Compression ratio changes of data pages under different device group sizes

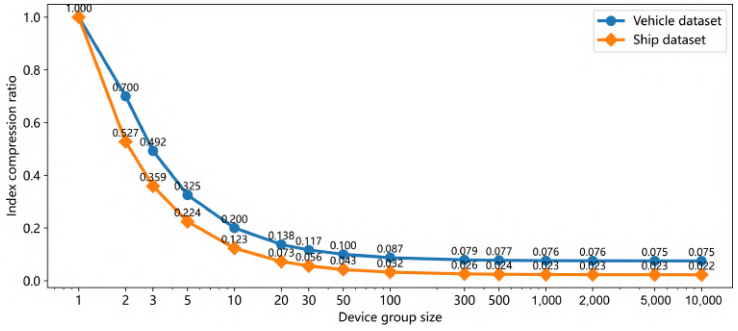


Figure 9 Compression ratio changes of indexes under different device group sizes

As shown in Figure 8, the application of the group compression method decreases the size of data pages in the file, and the decrease is related to the size of the specific dataset and

device group. As the device group size gradually increases, the compression ratio of data pages gradually decreases and tends to become stable. On different datasets, the compression ratio decreases and converges to different values, which we believe is related to the inherent data types, features, and encoding methods of the datasets. Firstly, the ship dataset stores navigation logs in the form of strings, which have a uniform format and contain a large number of duplicate values. Secondly, during the experiment, these strings are stored in TsFile through dictionary encoding. Therefore, dictionary encoding fully exploits the duplicate features of these logs. When data are scattered across a large number of short series, each short series needs to independently store the dictionary information, which results in high redundancy. When short series are concatenated with the group compression method, the data from multiple devices share the same dictionary information, which significantly improves the encoding efficiency. Therefore, the final compression ratio of the ship dataset becomes smaller. The vehicle dataset mainly consists of integers and floating-point numbers, and the reduction in compression ratio caused by concatenation of short series is limited.

As shown in Figure 9, on different datasets, as the device group size increases, the file index size gradually decreases and eventually becomes stable. This indicates that our group compression method can effectively reduce the storage overhead of indexes. In addition, we can see that when the device group size is 100, the compression ratio of indexes on the vehicle dataset is 8.7%, and the index size is reduced by 91.3%, which is consistent with the theoretical value calculated in Example 1.

In addition, the group compression method can reduce the storage proportion of indexes. We analyze the changes in the proportion of indexes in the total file size, and the results are shown in Figure 10. As mentioned above, the group compression method fully utilizes the metadata of multiple devices but does not reduce the number of index pointers or the size of data pages before compression. Instead, it reduces storage burden by utilizing their similarity. For data pages, we improve compression ratio and reduce storage costs by concatenating short series into long series. For indexes, we divide indexes into two layers to reduce redundant data and lower storage costs. However, the sizes of data pages and indexes are constantly changing and exhibit differences across different datasets. Therefore, we analyze the proportion of index size in file size to determine which part dominates in file storage.

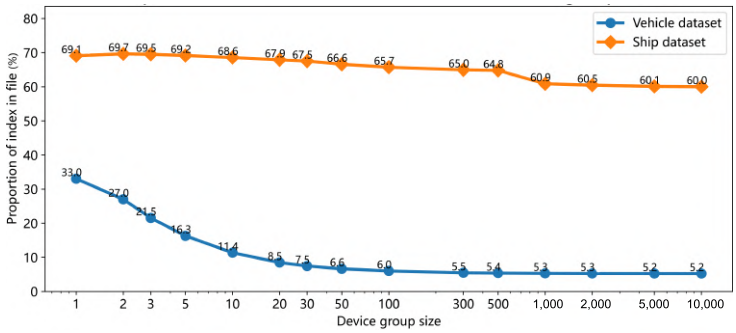


Figure 10 Proportion of indexes in file under different device group sizes

As shown in Figure 10, the storage proportion of indexes gradually decreases as the device group size increases, indicating that the portion of unit storage space reserved for data pages is gradually increasing. The proportion of indexes on different datasets shows significant differences. For example, the proportion of index storage on the ship dataset is significantly higher, always above 60%, while that on the vehicle dataset decreases from 33.0% to 5.2%. We believe this is due to the data characteristics of the ship dataset. As shown in Table 1, although

both datasets have tens of thousands of time series in the same file, the vehicle dataset has about 14.9 million lines and the ship dataset has only about 400 thousand lines. This indicates that the vehicle dataset and the ship dataset have 109 and 3 data points per time series on average, respectively. Although the number of lines per time series is small in the ship dataset, the character text length of each data point is long, with an average length of about 430 bytes, while each data point in the vehicle dataset only requires 4 bytes (INT32) or 8 bytes (DOUBLE) for storage. This difference results in a higher proportion of metadata in the storage of the ship dataset in TsFile. In addition, in the ship dataset, the UID information of certain devices is directly stored in the time series name, which results in longer time series names and is also reflected in the index size.

Overall, by analyzing the changes in the total file size, we confirm that the group compression method can effectively reduce storage space, particularly when the device group size is about 100. By analyzing the changes in various parts of the file, we find that the group compression method mainly improves the space efficiency from two aspects:

(1) Reduction of indexes. The use of second-layer indexing reduces the number of strings required for storage and lowers the storage overhead for indexing time series in the file.
(2) Optimization of compression ratio of time series. Due to concatenation of multiple short series with high compression ratios into long series with low compression ratios, the compression ratio of data pages in chunks is further reduced, saving much space.

Through the experiment in this section, we confirm that on the selected datasets, when the device group size is set as 100, the group compression method can achieve good compression effects and reflect its unique characteristics. Therefore, we select the device group size of 100 for analyzing the file query, writing, and fragmented file merging.

5.3 Query efficiency

The implementation of the group compression method is associated with the TsFile format and does not depend on specific database management systems. Therefore, we mainly analyze the query efficiency when using the TsFile-native query interface. In this experiment, we measure the query time consumption of the TsFile using the non-group method and the TsFile using the group compression method under a fixed number of queries. The time series to query are randomly and uniformly selected from the dataset, and each query requires reading all the data of the selected time series. To further confirm the query efficiency in real-world scenarios, we conduct query tests in two scenarios: without cache and with cache. In the scenario without cache, each query must read the file to obtain the query result. In the scenario with cache, we simulate the caching implementation of a database management system. When the cache contains the information that needs to be queried, the historical query results in the cache are directly returned without the need to read the file again. For the size of the cache, we refer to the default configuration in IoTDB, where each dataset can be fully cached, meaning the cache size is sufficient to store all query history data. The results are shown in Figures 11 and 12.

Figures 11 and 12 show the comparison of the query time consumption in TsFile between two methods on the vehicle dataset and the ship dataset, respectively. Meanwhile, the impact of different time series lengths on query performance is analyzed. The horizontal axis in the plots represents the average number of data points per time series, i.e., the length of time series. For example, in this scenario where the number of data points is only 100, time series can be considered short time series. As the length of time series gradually increases, the time consumed by queries also increases because the number of data points read by each query increases.

In the case of using cache, the group compression method outperforms the non-group method. This is mainly because the smallest unit of TsFile query is data page. After receiving a

query request, TsFile will locate the position of the chunk based on the index, and then process data pages sequentially to obtain the query result. The advantage of the group compression method is that the time series data of multiple devices are stored in the same data page. Correspondingly, the time series data of multiple devices are cached. In this experiment, the device group size is 100, which means that after the TsFile with the group compression method is read, the time series data with the same measurement name of up to 100 devices can be cached each time, which greatly improves the cache hit rate. The non-group method can only cache the data of the time series after each TsFile read, and thus the cache hit rate is low.

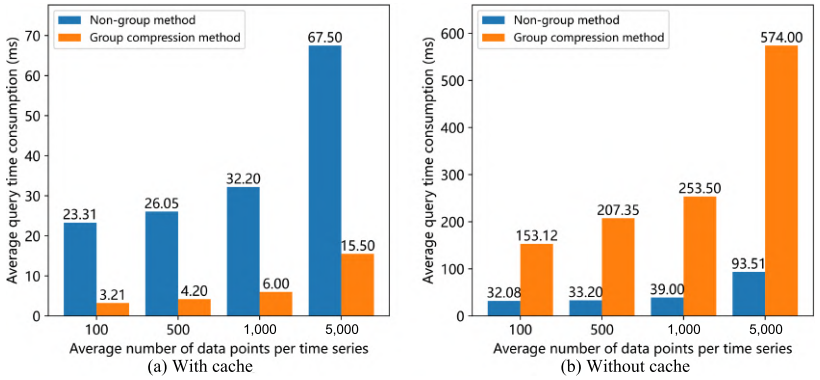


Figure 11 Comparison of the query time consumption on the vehicle dataset

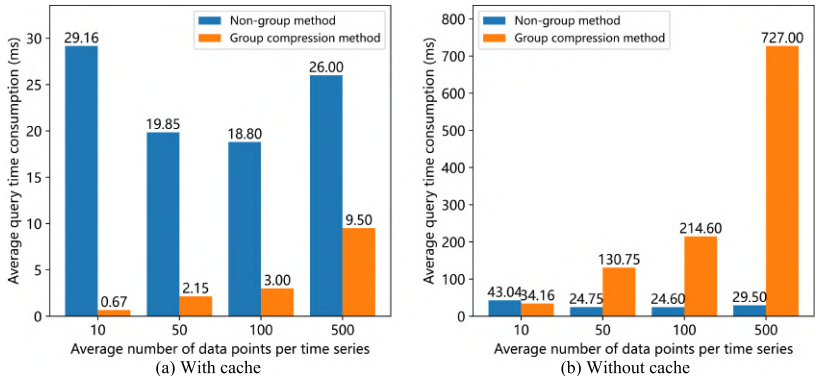


Figure 12 Comparison of the query time consumption on the ship dataset

In the case of not using cache, each query request requires reading TsFile, which increases the time consumption of each method. Compared with the non-group method, the group compression method takes more time to query, which is related to the fact that the minimum unit of TsFile query is data page. Since the group compression method concatenates short series with the same name from multiple devices into long series and stores them in data pages, the size of data pages will become larger. TsFile reads at least one data page at a time, and must read the entire data page completely. Accordingly, the group compression method always needs to read more bytes of data, which increases the time consumption. The time increase of the group compression method varies on different datasets, which can be explained with two main reasons.

(1) As shown in Figure 7, the ship dataset has a significantly lower compression ratio of data pages than the vehicle dataset, and thus shows a significant reduction in the size of data

pages and a relatively smaller number of bytes to be read.

(2) Data pages will limit the amount of data stored in each page. When setting the upper limit of data points, data exceeding the limit will be stored in the next page. The vehicle dataset has significantly a higher average number of lines than the ship dataset, which means that when short series are concatenated into long series, the number of time series that can be accommodated in the same data page is smaller.

Overall, in application scenarios with cache, the group compression method has higher query efficiency, which is attributed to the increased cache hit rate caused by the large amount of information in data pages. In the application scenarios without cache, the query of the group compression method will slow down.

5.4 Writing efficiency

As mentioned in Section 5.3, the group compression method does not depend on specific database systems. Therefore, in this experiment, we only use the TsFile-native write interface to measure the writing efficiency. In this experiment, we measure the time required to write TsFile by using both non-group and group compression methods for the same data. The results are shown in Figure 13.

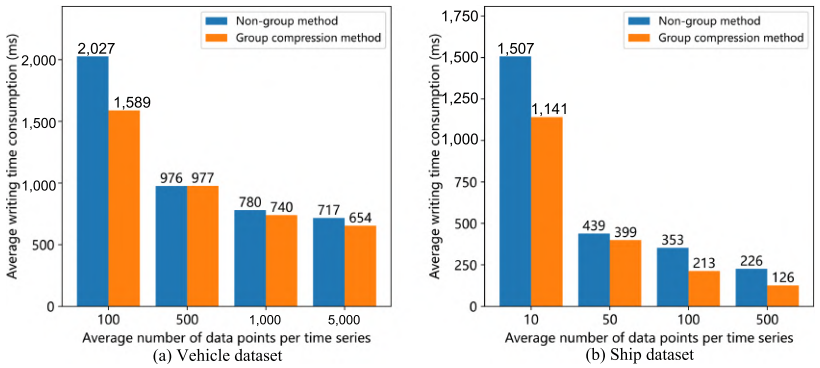


Figure 13 Comparison of the time consumption of writing on different datasets

As shown in Figure 13, the time consumption of writing with the group compression method is slightly lower than that with the non-group method. This result can be explained with two reasons.

(1) The calls to write interface are fewer. With the group compression method, only fewer write interfaces need to be called. After obtaining the data that needs to be written, the group compression method concatenates the time series with the same name of multiple devices in the same device group and writes them all at once.

(2) The data compression ratio is lower, requiring less data to be written. After application of the group compression method, short series will be concatenated into long series in memory. According to the analysis of file compression efficiency above, the compression ratio of long series is lower, and the space occupied after compression is less, and thus less data needs to be written.

Overall, using the group compression method to write short time series can improve the writing efficiency in TsFile.

5.5 Merging of fragmented files in TsFile

In this experiment, we measure the efficiency of file merging by using both non-group and group compression methods. We break down the non-group TsFile on each dataset into several

fragmented files to simulate the large number of small TsFiles generated by terminal devices. Next, we store the fragmented files separately by using non-grouping and group compression methods to prepare for merging. Finally, we use the two methods to merge the corresponding TsFiles and measure the time consumption of each stage. Considering the different hierarchical structures of time series in datasets, we summarize the fragmentation of each dataset in Table 4.

Table 4 Fragmentation of different datasets

Dataset	Limited number of devices per file	Limited number of lines per time series	Number of time series per file	Number of fragmented files
Vehicle dataset	68	80	680	100
Ship dataset	681	3	1,362	100

The number of time series for each device in the ship dataset is small, while each data point occupies more bytes. When the number of fragmented files is the same, the number of devices and total time series in each fragmented file is greater. We measure the time consumed in each stage of the file merging process, and the results are shown in Figure 14.

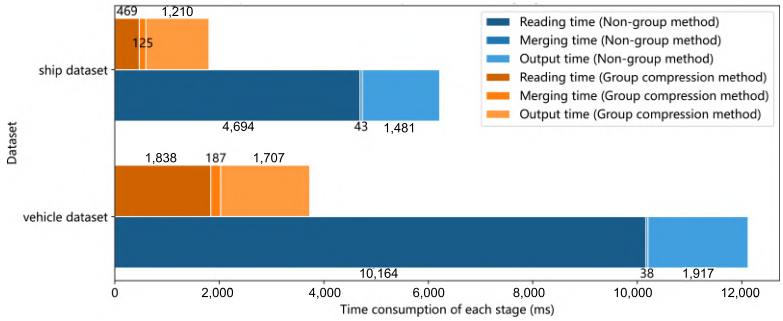


Figure 14 Comparison on the time consumed for the merging of different datasets

As shown in Figure 14, two methods are used for each dataset, resulting in two bar charts. Each bar chart is divided into three parts, which correspond to the time consumed by the three steps of the merging process. The three steps are: reading fragmented files, merging and organizing data in memory, and outputting the merged data to a new TsFile.

The group compression method consumes shorter total time, which is about 30% that of the non-group method. The time consumption mainly concentrates in file reading and input. Generally speaking, the reading of fragmented files takes the most time, followed by the output of merged results, and the shortest time is spent on merging and organizing in memory. After application of the group compression method, the time required for file reading is significantly reduced. This is attributed to the fact that the group compression method encodes time series with the same measurement name of multiple devices into the same chunk. Each call to the query interface can read more data, making it faster. In addition, the time consumed by merging and organizing data in memory only accounts a small part of the entire process. Nevertheless, the group compression method consumes 2–3 times more time than the non-group method. This is mainly because the group compression method requires data to be re-grouped and time series with the same measurement name in the same device group to be re-concatenated, while the non-group method only needs to merge multiple fragments of these time series. Therefore, the group compression method naturally takes longer time in data merging and sorting, while its overall time consumption is less.

In practical scenarios, the merging process does not only occur once. Instead, the merging progresses with the arrival of new data. To analyze the space optimization of the group

compression method during the gradual merging process, we analyze the changes in file size during the merging process. The results are shown in Figures 15 and 16, where the average length of time series is proportional to the number of fragments merged. In other words, longer time series indicate that more fragmented files are merged, and the change in the horizontal axis in the plots represents the gradual merging process of fragmented files. As the merging progresses, the length of time series in the file keeps growing, and the short time series gradually disappears, becoming a long time series. At this time, the storage efficiency of the non-group method will gradually improve.

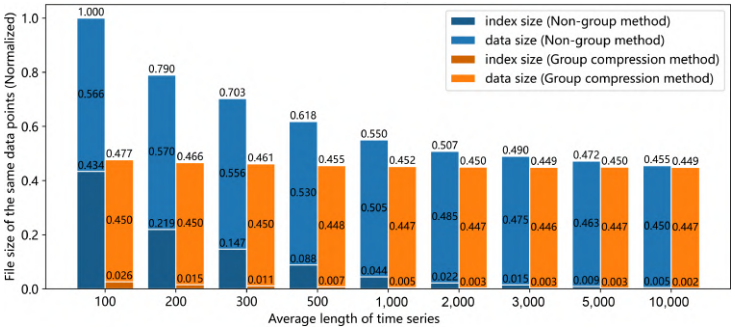


Figure 15 Changes in file size during merging on the vehicle dataset

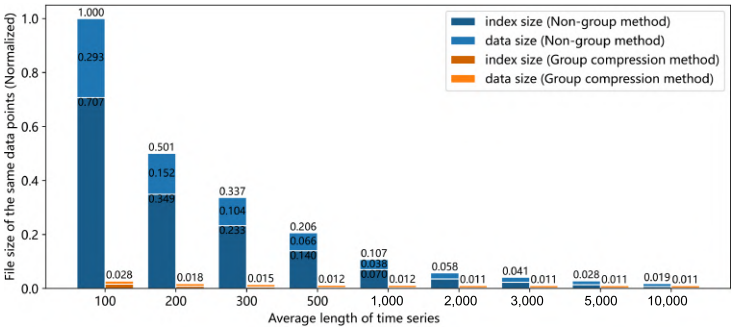


Figure 16 Changes in file size during merging on the ship dataset

For the convenience of comparing changes in file size, in Figures 15 and 16, we set the vertical axis to the normalized file size, which is the ratio of the absolute file size to the number of data points it stores. In this way, we can intuitively see that when storing the same number of data points, the longer time series and fewer time series in the file correspond to the smaller the size of TsFile and the higher storage efficiency. After application of the group compression method, the size of TsFile is relatively stable. With the gradual merging of fragments, the mean length of time series keeps increasing, and the corresponding relative file size decreases. After 100 fragmented files are merged, the efficiency of the non-group method and the group method tend to be consistent. In addition, since the ship dataset mainly contains ship log data stored in strings by dictionary encoding, its compression ratio of long time series is very low, and the file size will significantly decrease after merging.

6 Summary and Prospects

This article comprehensively elaborates on the design and implementation of the group compression methods in Apache TsFile. We delved into the challenges faced by TsFile in the

scenarios of short time series, identified the features of short time series, and fully utilized these features to introduce the group compression method.

Our group compression method can fully exploit the highly repetitive feature of device metadata information in scenarios of short time series for targeted optimization. We addressed challenges such as low compression efficiency of a large number of short series, high proportion of indexes, and metadata redundancy, and optimized the compression ratio of Apache TsFile by proposing this method. Furthermore, we determined the reasons for the optimized compression effect through experiments and confirmed that the group compression method improves both data page compression and index compression. In addition, we verified through evaluations that our method balances both writing and reading performance, and has advantages in the merging of fragmented files. As short series are gradually merged, the advantages of the group compression method gradually decrease. At this point, the non-group method can be used for storage in TsFile to ensure the query performance.

It needs further research and improvement for this work. Further efforts can be made to optimize file writing at the edge side in the scenario of terminal-edge-cloud collaboration, such as designing an efficient streaming writing and compression method to improve efficiency. A feasible TsFile file merging strategy can be studied to accelerate the data organization during file merging. The support of user-defined pre-grouping strategies can be considered, such as automatically grouping based on commonly used aggregate queries to improve data management and query efficiency. In addition, our group compression method improves data compression efficiency by concatenating short time series, which is universal and has the potential to be further extended to more file formats.

References

[1] Wu DP, Zhang PN, Wang RY. Smart Internet of Things aided by “terminal-edge-cloud” cooperation. Chinese Journal on Internet of Things, 2018, 2(3): 21–28 (in Chinese with English abstract).

[2] Qiao JL. Research on file structure design and optimization of time series database management system for Internet of Things [Ph.D. Thesis]. Beijing: Tsinghua University, 2021 (in Chinese with English abstract).

[3] Wang C, Qiao JL, Huang XD, Song SX, Hou HN, Jiang T, Rui L, Wang JM, Sun JG. Apache IoTDB: A time series database for IoT applications. Proc. of the ACM on Management of Data, 2023, 1(2): 195.

[4] Apache IoTDB. https://iotdb.apache.org/

[5] Zhao X, Qiao JL, Huang XD, Wang C, Song SX, Wang JM. Apache TsFile: An IoT-native time series file format. Proc. of the VLDB Endowment, 2024, 17(12): 4064–4076.

[6] TsFile. https://tsfile.apache.org/

[7] Influxdata. 2024. https://docs.influxdata.com/influxdb/clustered/

[8] The new InfluxDB storage engine: Time structured merge tree. 2015. https://www.influxdata.com/blog/new-storage-engine-time-structured-merge-tree/

[9] Announcing InfluxDB IOx—The future core of InfluxDB built with rust and arrow. 2020. https://www.influxdata.com/blog/announcing-influxdb-iox/

[10] Apache HoraeDB. https://horaedb.apache.org/

[11] Xiao JZ, Huang YX, Hu CY, Hu CY, Song SX, Huang XD, Wang JM. Time series data encoding for efficient storage: A comparative analysis in Apache IoTDB. Proc. of the VLDB Endowment, 2022, 15(10): 2148–2160.

[12] Xia TR, Xiao JZ, Huang YX, Hu CY, Song SX, Huang XD, Wang JM. Time series data encoding in Apache IoTDB: Comparative analysis and recommendation. The VLDB Journal, 2024, 33(3): 727–752.

[13] Golomb S. Run-length encodings (Corresp.). IEEE Trans. on Information Theory, 1966, 12(3): 399–401.

[14] Blalock D, Madden S, Guttag J. Sprintz: Time series compression for the Internet of Things. Proc. of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies, 2018, 2(3): 93.

[15] Welch TA. A technique for high-performance data compression. Computer, 1984, 17(6): 8–19.

[16] Li C, Markl V, Pelkonen T, Franklin S, Teller J, Cavallaro P, Huang Q, Meza J, Veeraraghavan K. Gorilla: A fast, scalable, in-memory time series database. Proc. of the VLDB Endowment, 2015, 8(12): 1816–1827.

[17] Liakos P, Papakonstantinopoulou K, Kotidis Y. Chimp: Efficient lossless floating point compression for time series databases. Proc. of the VLDB Endowment, 2022, 15(11): 3058–3070.

[18] Fang C, Song SX, Guan HQ, Huang XD, Wang C, Wang JM. Grouping time series for efficient columnar storage. Proc. of the ACM on Management of Data, 2023, 1(1): 23.

[19] Fang CG, Chen ZJ, Song SX, Huang XD, Wang C, Wang JM. On reducing space amplification with multi-column compaction in Apache IoTDB. Proc. of the VLDB Endowment, 2024, 17(11): 2974–2986.

[20] Bartík M, Ubik S, Kubalik P. LZ4 compression algorithm on FPGA. Proc. of the 2015 IEEE Int'l Conf. on Electronics, Circuits, and Systems. Cairo: IEEE, 2015. 179–182. [doi: 10.1109/ICECS.2015.7440278]



Xingyu Liu, master's degree candidate. His research interest is time series databases.



Xiangdong Huang, Ph.D., associate professor. His research interests include industrial data management and distributed storage systems.



Shaoxu Song, Ph.D., associate professor, doctoral supervisor. His research interests include databases, data quality, time series data cleaning, and big data integration.



Jianmin Wang, Ph.D., professor, doctoral supervisor. His research interests include databases, workflows, big data, and knowledge engineering.