Apache IoTDB: A Time Series Database for IoT Applications

CHEN WANG, Tsinghua University, China
JIALIN QIAO and XIANGDONG HUANG, Timecho Ltd, China
SHAOXU SONG, Tsinghua University, China
HAONAN HOU, Timecho Ltd, China
TIAN JIANG, LEI RUI, JIANMIN WANG, and JIAGUANG SUN, Tsinghua University, China

A typical industrial scenario encounters thousands of devices with millions of sensors, consistently generating billions of data points. It poses new requirements of time series data management, not well addressed in existing solutions, including (1) device-defined ever-evolving schema, (2) mostly periodical data collection, (3) strongly correlated series, (4) variously delayed data arrival, and (5) highly concurrent data ingestion. In this paper, we present a time series database management system, Apache IoTDB. It consists of (i) a time series native file format, TsFile, with specially designed data encoding, and (ii) an IoTDB engine for efficiently handling delayed data arrivals and processing queries. The system achieves a throughput of 10 million inserted values per second. Queries such as 1-day data selection of 0.1 million points and 3-year data aggregation over 10 million points can be processed in 100 ms. Comparisons with InfluxDB, TimescaleDB, KairosDB, Parquet and ORC over real world data loads demonstrate the superiority of IoTDB and TsFile.

CCS Concepts: • **Information systems → Data management systems**.

Additional Key Words and Phrases: time series, data model, database engine, distributed

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

1 INTRODUCTION

In the Internet of Things (IoT), a huge amount of time series is are generated by various devices with many sensors attached. The data need to be managed not only in the cloud for intelligent analysis but also at the edge for real-time control. For example, more than 20,000 excavators are managed by one of our industrial partners, a maintenance service provider of heavy industry machines, each of which has hundreds of sensors, e.g., monitoring engine rotation speed. As illustrated in Figure 1, the data are first packed in devices, and then sent to the server via 5G mobile network. In the server, the data are written to a time series database for OLTP queries. Finally, data scientists may load data from the database to a big data platform for complex analysis and forecasting, i.e., OLAP tasks.

Shaoxu Song (https://sxsong.github.io/) is the corresponding author.
Authors' addresses: Chen Wang, wang_chen@tsinghua.edu.cn, Tsinghua University, Beijing, China; Jialin Qiao, jialin.qiao@timecho.com; Xiangdong Huang, hxd@timecho.com, Timecho Ltd, Beijing, China; Shaoxu Song, sxsong@tsinghua.edu.cn, Tsinghua University, Beijing, China; Haonan Hou, haonan.hou@timecho.com, Timecho Ltd, Beijing, China; Tian Jiang, jiangtia18@mails.tsinghua.edu.cn; Lei Rui, rl18@mails.tsinghua.edu.cn; Jianmin Wang, jimwang@tsinghua.edu.cn; Jiaguang Sun, sunjg@tsinghua.edu.cn, Tsinghua University, Beijing, China.

2836-6573/2023/6-ART195
https://doi.org/10.1145/3589775

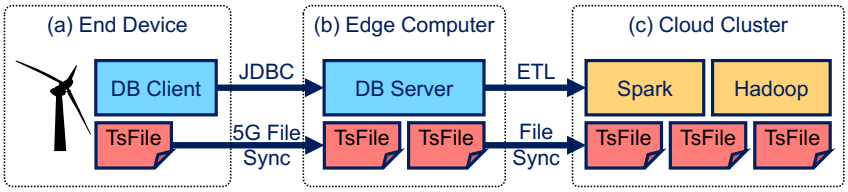


Fig. 1. Data management in IoT scenarios

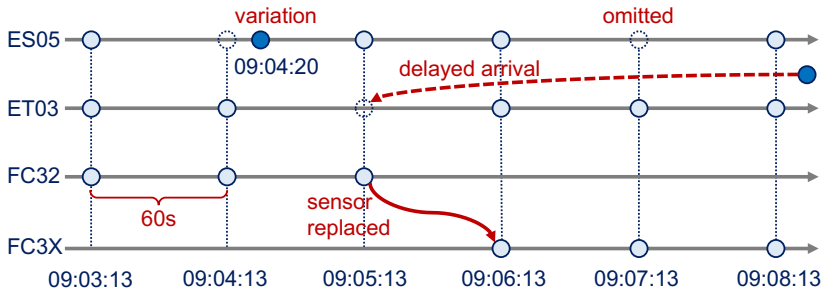


Fig. 2. Time series management issues in IoT scenarios

1.1 Motivation

The process in Figure 1 poses new requirements to time series database management systems. (1) In the end device, such as the aforesaid excavator, a lightweight database or a compact file format is needed to save space and network bandwidth. (2) In the edge server, a full-function database collects, stores and queries the massive data of devices, capable of handling delayed arrivals. (3) In the cloud, a database cluster with complete historical data persistence connects directly to big data analysis systems, such as Spark and Hadoop, and enables OLAP queries. In addition to the large scale issues, millions of series (columns) and billions of points (rows), we highlight below the unique and urgent features in the IoT scenarios.

1.1.1 Device-defined Ever-evolving Schema. Unlike the traditional databases with pre-defined schema, the schema of time series data in the IoT scenario is defined by sensors in the devices. During the device maintenance or upgrade, sensors are frequently removed, replaced or augmented, leading to changed schema. For instance, as illustrated in Figure 2, sensor FC32 for monitoring fuel consumption is replaced by FC3X, at time 09:06:13. We need a data model that is sufficiently flexible to capture such ever-evolving schema.

1.1.2 Mostly Periodical Data Collection. Machine generated sensor data are often collected periodically with a pre-set frequency. While the time series is expected with a regular time interval, there may be small variations due to data bus congestion or network delay. Even worse, those values not changed with the previous may be omitted to save energy. For example, in Figure 2, ES05 is mostly collected in every 60 seconds, but with a small delay from time 09:04:13 to 09:04:20 and an omitted data at time 09:07:13. Data encoding should be able to handle such variations for efficient storage.

1.1.3 Strongly Correlated Series. It is also worth noting that multiple sensors, e.g., in the same module of a device, may collect data at the same time. In addition to the same timestamps, their values may also be correlated. For instance, in Figure 2, the fuel consumption (FC32/FC3X) value

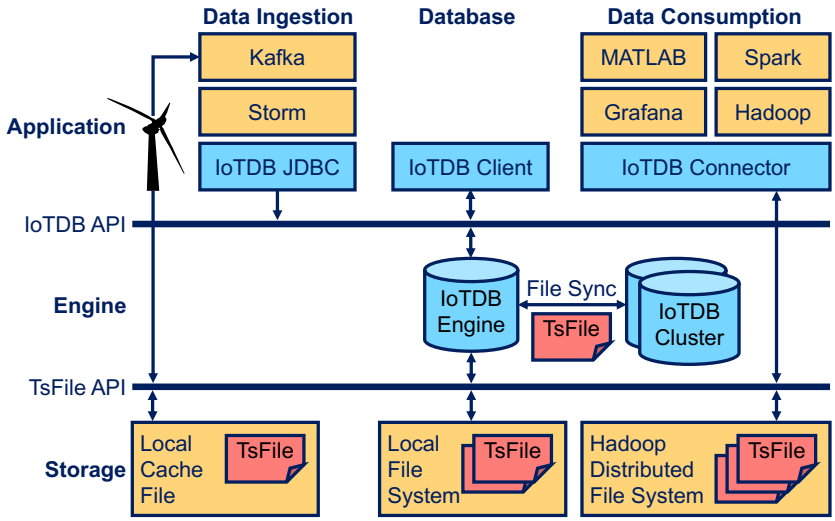


Fig. 3. A data life cycle view of the IoTDB system

should be determined by engine speed (ES05) and torque (ET03) at the same time. Moreover, the wind speeds of close turbines in the same wind farm should be similar with each other. Again, the storage scheme is expected to fully utilize such opportunities in data compression.

1.1.4 Variously Delayed Data Arrival. While most data points arrive in time order, serious delays may occur, e.g., owing to network delay or corruption. For instance, in Figure 2, the data point with timestamp 09:05:13 arrives after it subsequent points. The delay could be various, ranging from seconds to days. This issue is unique in time series data and seriously obstruct the time ordered storage.

1.1.5 Highly Concurrent Data Ingestion. High concurrency is prevalent in the IoT scenarios, from edge devices to the cluster. For instance, each wind turbine has 500+ sensors generating data at the same time. The database needs to ingest the data of 100-200 turbines in a wind farm, while the cloud cluster manages 20,000+ turbines in total. The highly concurrent data make not only database ingestion challenging, but also the replication among nodes in the cluster.

1.2 Solution

In this study, we introduce a novel, IoT-oriented time series database, IoTDB, built upon a new columnar time series file format, TsFile. As illustrated in Figure 3, the solution is deployed ranging from end devices for data collection, edge servers for local control, to the cloud for device monitoring and advanced analysis. It manages millions of time series, stores terabytes of data, inserts 10 million points per second, selects 0.1 million points and aggregates 10 million points in 100 milliseconds. The open architecture enables both real-time query and complex data analysis over the same data without ETL. (Please see the system design in Section 2.)

In general, the features unique in IoT scenarios, aforementioned in Section 1.1, lead to our design choices and novelty below.

1.2.1 TsFile Format. To avoid the costly ETL process, the same TsFile could be deployed and synchronized among end devices, edge servers, and cloud nodes, without packing and unpacking.

Table 1. File comparison

Dimension	TsFile	Parquet	ORC
Data model	Series tree	Nested	Tabular
Index on seriesID	Yes	No	No
Time encoding	Regular interval	Long-typed	Long-typed
Value encoding	Multi-series	Nested-column	Single-column
Frequency domain	Yes	No	No

Design Choices. The major features of TsFile, introduced in Section 4, are highlighted as follows. (1) An IoT native data model organizes time series in a tree structure of devices and sensors. The tree structure could naturally represent the evolving schema, e.g., FC32 and FC3X as siblings in Figure 4 both measuring the fuel consumption before and after sensor replacement. It also serves as an index on series for fast access. (2) Columnar storage of time series with time and value columns enables efficient data encoding specially designed for regular time intervals with variations. Moreover, by packing multiple devices and sensors together in one file, the same time column could be share among multiple series with aligned timestamps. It also enables the effective value encoding by leveraging the correlations among series.

Novel Techniques. (1) While the series tree structure is capable to represent the ever-evolving schema, we also propose methods [52] to automatically identify the changed sensors monitoring the same thing, such as the aforesaid FC32 and FC3X both measuring the fuel consumption before and after sensor replacement. Please see Section 3.3 for an introduction. (2) Moreover, our comparative analysis of encoding [59] shows that most existing data encoding methods do not consider the unique features of time series in the IoT scenario. For timestamp encoding, we propose to imply the regular time interval and thus only need to store the small variation [22], e.g., 09:04:20-09:04:13=7 in Figure 2. Note that different sensor modules or devices may collect data at the same time with identical frequency, sharing the same timestamps. Thereby, we further propose to align the time series for efficient storage by eliminating their similar timestamps [21]. For value encoding, we obtain a regression model to capture the correlations among series and store only the difference between the regression and the observation values [23]. Please see Section 4.3 for more details. (3) Finally, frequency domain analysis on time series is widely conducted. While it is costly to online transform from time domain to frequency domain, e.g., by Fast Fourier Transform (FFT), we store the frequency domain data for reuse, via efficient data encoding in TsFile [57]. Please see Section 4.4 for a discussion.

Comparison. As illustrated in Table 1, the aforesaid features unique in IoT scenarios are not considered in Parquet [42] or ORC [27] for nested or tabular data. Given the time series specialized index and encoding, extensive experiments in Section 7.2 demonstrate the superiority of TsFile against Parquet and ORC.

1.2.2 IoTDB Engine. Over the same TsFile, the IoTDB engine could handle extremely intensive writes as well as various read loads including real-time queries (OLTP) and complex data analysis (OLAP).

Design Choices. The key design choices are discussed in Section 5 and summarized below. (1) An IoT adapted LSM-tree, Log-Structured Merge-Tree [39], efficiently handles delayed data arrivals in a write-intensive workload. For the points with a short delay, the data are first cached in MemTables

Table 2. DBMS comparison

Dimension	Apache IoTDB	InfluxDB	TimescaleDB	KairosDB
Storage	TsFile	TSM	PostgreSQL	Cassandra
Statistics	Yes	No	Yes	No
Organization	LSM (pipelined)	LSM	B+tree	LSM
Data delay	Separation	Merge	Merge	Merge
Replication	NB-Raft	HH	Streaming	HH

and then flushed to disk as TsFile sorted on time. A pipeline of sorting, encoding and compressing is designed to enable parallel computing. (2) A query engine fully utilizes the statistics maintained in different levels of TsFile. Note that the data flushed in TsFile are immutable in the LSM-tree data organization. It motivates us to pre-compute the statistics in different levels of TsFile and utilize them in query optimization, in particular in answering aggregation queries.

Novel Techniques. (1) For the points with a long delay, i.e., their subsequent points have already been flushed to TsFile, they will be written to other TsFiles. It leads to the overlapping time ranges of different TsFiles, incurring great query overhead in the LSM-tree like data organization. According to our study in many real applications [35], the long delay does not occur frequently, e.g., only 0.0375% in the aforesaid heavy machine service provider. In this sense, we propose to automatically separate the delayed data points to reduce the overlaps of TsFiles for efficient query processing [35]. Please see Section 5.2 for more details. (2) To replicate data in a cluster, consensus protocol such as Raft [40] is employed. However, the serialization of appending entries blocks the subsequent requests and thus limits the parallelism and throughput of the system. We propose NB-Raft [33], non-blocking replication for highly concurrent IoT data. In this way, more requests can be processed in parallel, and thus the throughput increases, essential for IoT applications often with vast sensors and fast data ingestion. Please see Section 6.2 for an introduction. (3) For inconsistent replicas, rather than waiting for all the replicas to response, we propose a DYNAMIC read consistency level [53], returning results based on a part of the replicas responded thus far, as introduced in Section 6.3.

Comparison. NoSQL systems such as HBase [5], Cassandra [36] and Druid [60] as well as their time series adaptions OpenTSDB [48] built on HBase and KairosDB [34] on Cassandra, do not have special consideration of variously delayed data arrivals, unique in time series data. It is not addressed either in TimescaleDB [54] built on RDBMS PostgreSQL or the time series specialized InfluxDB [30]. Again, the experiments in Section 7.3 show that IoTDB outperforms the aforesaid InfluxDB, TimescaleDB, KairosDB and so on.

1.2.3 Open Source. IoTDB is now an Apache Top-Level Project [6]. The system is well integrated with the open-source ecosystem, such as JDBC driver, Kafka [9] for data ingestion, Grafana [14] for data visualization, Hadoop [4] and Spark [61] for data analysis, etc. A preliminary version of Apache IoTDB is demoed in [56].

2 SYSTEM DESIGN

Figure 3 presents an overview of the IoTDB system. In contrast to OpenTSDB [48] built on HBase and KairosDB [34] on Cassandra, IoTDB is constructed from scratch. Its core components (in blue and red) successfully fulfill the requirements in the whole life cycle of time series data in IoT applications as follows.

Data Ingestion and Synchronization. To ingest data, as illustrated in Figure 3, the most direct way is to generate the TsFile, and periodically synchronize the written TsFile to the edge server. Alternatively, one may also use IoTDB JDBC to write the data in real time through the IoTDB engine. For extremely heavy writes, big data processing systems such as Kafka [9] and Storm [10] could be connected to absorb burst traffics.

In some industrial scenarios, there may be many IoTDB instances. For example, a company with many wind farms deploys an IoTDB instance in each wind farm for easier local controls of its managed wind turbines. To exchange data between wind farms and converge to the data center, one IoTDB instance synchronizes its local TsFiles to another IoTDB instance. The remote IoTDB receives TsFiles and loads them automatically. In this way, we bridge the gap between end devices and edge servers as well as cloud nodes.

Database Engine. Time Series File (TsFile) is a new file format designed to efficiently manage the time series data, presented in Section 4. They can be resident in either the local file system of end devices and edge servers, or the Hadoop Distributed File System (HDFS) [4] in a cluster. TsFiles can be directly accessed by the IoTDB engine for query and by Hadoop/Spark through the TsFile Connector for analysis. By this means, we support both OLTP and OLAP without reloading data to different stores.

The IoTDB Engine is the core part of IoTDB, introduced in Section 5. It contains a storage engine that can manage millions of time series and write millions of data points per second (with potentially delayed arrivals). It also contains a query engine optimized to get results in a few milliseconds over trillions of data points. Advanced queries on time series supported in IoTDB include but are not limited to downsampling, aggregation, pattern matching, and most importantly are extensible through the User-Defined Functions (UDF). The IoTDB Client is a command-line interface, together with features like continuous query and trigger, enables the local monitoring and control in the edge server, e.g., in a wind farm. A distributed solution of IoTDB is also introduced in Section 6.

Data Consumption. With various built-in (and future extensible) Connectors, the data in IoTDB could be easily consumed by downstream applications. For instance, with the Grafana-IoTDB connector, users can design their dashboard in Grafana [14] to visualize the time series. The IoTDB Client also enables users to write SQL queries and visualize the data interactively in Apache Zeppelin [11]. Using MATLAB, one can read time series data from IoTDB and execute mathematical calculations for analysis. Through the Hadoop-IoTDB and Spark-IoTDB connectors, long historical data can be processed and analyzed in parallel by MapReduce.

3 DATA MODEL

While InfluxDB [30] uses tags to organize and identify time series, we notice after a long survey in our many industrial partners that tags only are too weak to manage devices and sensors rigidly in enterprises, and difficult to optimize the physical storage.

3.1 Design: Logical Schema

IoTDB manages all time series in the form of a tree structure, where each leaf node corresponds to a time series. The hierarchical structure naturally fits the management levels in industry [44] and thus helps users organize time series methodically. An example of the tree in the wind turbine scenario is shown in Figure 4. In particular, the following two levels appear almost in all IoT applications.

Sensor. A sensor measures a physical quantity such as power, voltage, current, bracket displacement, wind speed, vehicle speed, longitude, latitude, and so on. As shown in Figure 4, each leaf

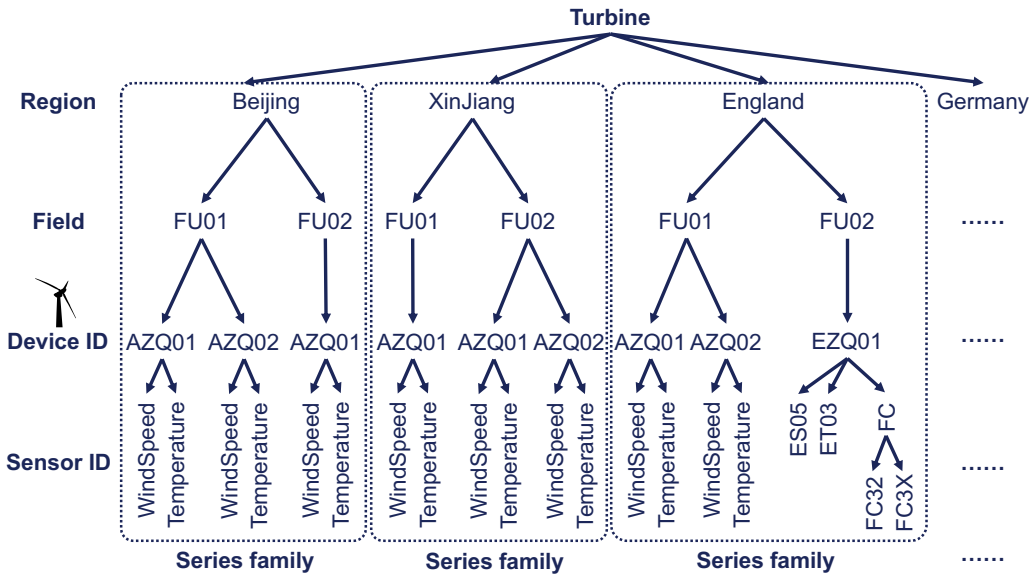


Fig. 4. Data model in IoTDB

node of the tree corresponds to a sensor. Due to space limitations, only a part of the sensors is plotted in this figure.

Device. A device is a piece of equipment that has sensors. In IoTDB, each sensor has its corresponding owning device. As shown in Figure 4, a device is located by the path from the root node to the penultimate level node.

3.2 Design: Physical Schema

While the hierarchy describes how the time series are organized in the logical level, the physical schema hidden beneath further specifies the organization in the physical storage.

3.2.1 Time Series. A time series is a series of data points recorded by a sensor over time. Each series consists of two attributes, time and value, as presented in Figure 5(a). A time series is located by the full path from the root node to the leaf node. For example, in Figure 4, Turbine.Beijing.FU01.AZQ01.WindSpeed represents the time series recorded by the wind speed sensor of the AZQ01 device in the FU01 field of Beijing.

3.2.2 Series Family. Time series from the same device are stored together in TsFiles and a TsFile can store time series from multiple devices. We further control over which devices are stored together based on application requirements, which is the motivation behind the concept of the series family. A series family contains multiple devices whose data will be stored together in TsFiles. Each series family has an independent storage engine, and all TsFiles in it will be stored in the same directory. Thereby, series families are physically isolated like the "database" concept in traditional databases, physically separating the data of different users and meanwhile facilitating data migration and analysis. For example, in Figure 4, all the time series in one region form a series family.

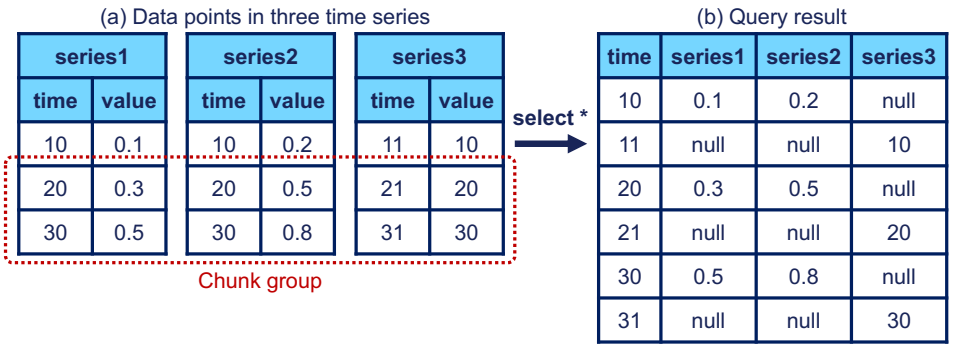


Fig. 5. Time series data and aligned query result

3.3 Technique: Automatic Schema Identification

Different from the traditional schema-first databases, the data may come before the corresponding schema defined in the IoT scenarios, i.e., data-first. For instance, a wind turbine immediately starts to transfer data when installed before registering it in the data center. Moreover, workers may occasionally mis-plug the cables of sensors during equipment maintenance, misplacement among different sensors occurs frequently. According to our preliminary study [52], 200 out of 5.2k tuples are observed with scheme issues in the real wind turbine data. In this sense, we offer automatic schema identification of time series [52]. It enables users to insert data without defining the schema in advance, highly demanded in the write-intensive IoT scenarios to avoid data loss.

4 TSFILE FORMAT

To satisfy the IoT requirements and offer fast data ingestion, efficient query processing, and compact storage space at the same time, we present the *Time Series File* (TsFile), the first file format optimized for time series data management.

4.1 Design: File Structure

When designing the structure of TsFile, we recognize that (1) the data should be compressed as much as possible for saving space; (2) multiple devices need to be packed together to reduce the number of files; (3) some time series are often queried together and expected to be close in physical locations; (4) the data ought to be packed with sizes coincident with file systems to avoid disk fragment; and (5) millions of time series need to be efficiently accessed. Figure 6 shows the structure of a TsFile.

Page. A page is a basic unit to store time series data on disks. Each time series in a page is sorted by time in ascending order, with two columns, one for timestamps and the other for values. The timestamps and values are stored and compressed separately. Such a columnar store compresses time series data greatly, given the increasing time and steady value in many industrial applications (see a detailed discussion in Section 4.2). While pages share the same size for easier access, the amount of data on a page is configurable, affecting the performances of compression and query processing.

Chunk. A chunk consists of a metadata header and several pages, all belonging to one time series. Unlike pages, the chunk size is variable. The chunk metadata includes the data type of this series,

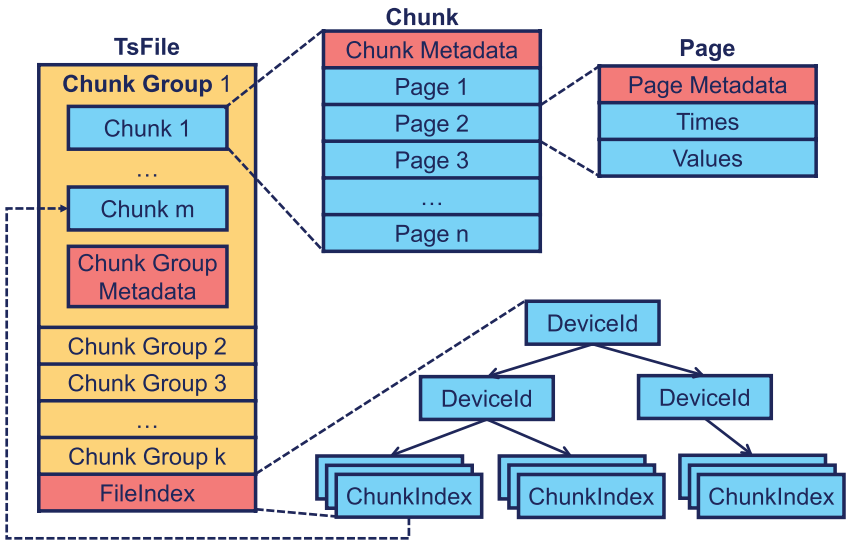


Fig. 6. TsFile structure

as well as the encoding and compression information, which allows us to use different algorithms to compress different time series.

Chunk Group. A chunk group contains multiple chunks, which belongs to one or multiple series of a device written in the same period of time. The reason for grouping them together in the continuous disk space is that these chunks of series of a device are often queried together. For example, in Figure 5(a), a device has 3 sensors corresponding to 3 series. The chunk group contains 3 chunks of series, respectively, written in the same period.

Block. When writing data to a TsFile, all chunk groups being written are buffered in memory first. We bound the memory consumption by setting a threshold on the total memory usage of all active chunk groups. When the memory usage reaches the threshold, we flush all chunk groups to the TsFile. Therefore, these chunk groups form a block, which is a logical concept. The block size can be adjusted for achieving data locality in HDFS.

Index. The file metadata at the end of TsFile contains a chunk-level index and the file-level statistics for efficient data access. File-level statistics record the statistics of each time series, such as start time, end time, count, and the corresponding positions of the chunk-level index. The chunk-level index records the position of each chunk and is grouped by device and sensor. That is, we can directly get all chunk metadata belonging to a time series. Each index also contains the time information, such as the maximum and minimum timestamps, which can be used in query optimization.

4.2 Design: Columnar Storage

Time and Value Columns. As illustrated in Figures 5(a) and 6, we store the time column for each time series instead of storing only one time column in a device or a data file, which is a tradeoff between space and time. The design choice is made for the following benefits. (1) Null value elimination. There is no need to store null values for aligning multiple series, which saves

the disk occupation and simplifies the implementation. (2) Data access locality. Each time series can be directly processed inside pages, without the need to join the time column elsewhere.

In most cases, sensors collect data at a regular time interval. However, the timestamps usually have small deviations in the real world, which result in many null values when aligning multiple time series, wasting space. Indeed, the time column can be compressed concisely using a proper compression method as introduced below. Therefore, though the data access may introduce the extra overhead of aligning multiple series, this tradeoff is worthy.

Encoding and Compression. By separating the timestamp and value into individual columns, we can adopt different encoding and compression methods to time and value columns. In TsFile, we only compress data at the page level so that we do not need to uncompress the whole file to read a part of the data.

We first translate timestamps/values into bytes by data encoding, then use a compression method to compress the encoded bytes. Many existing encoding methods are implemented, including run-length encoding (RLE) [24], Bit-packing [17], delta-delta encoding (2DIFF) [3], GORILLA [43] and so on. The implemented compression methods are Snappy [25], GZIP [19], LZ4 [16], etc.

4.3 Technique: Times Series Encoding

While there exists many encoding algorithms, our comparative analysis of encoding [59] shows that most of them do not consider the unique features of time series in the IoT scenario.

For timestamp encoding, as introduced in Section 1.1.2, most sensor data are collected with regular time intervals but may have some variation and omission. Therefore, we propose to imply the regular time interval [22], e.g., in every 60 seconds in Figure 2. Rather than storing the long timestamp (or its delta), we only need to store the count of regular time intervals and the small variation, e.g., 2 regular time intervals and 0 variation for 09:08:13 in ES05.

For value encoding, we utilize the correlations among series as mentioned in Section 1.1.3. Specifically, we use a model to capture the similarity or regression relationships among series [23]. Consequently, it only needs to record the difference between the regression and the observation values. For instance, we store one series of large wind speed values v. Some other close wind turbines in the same wind farm record only the small delta Δ_v to v instead.

It is worth noting that different sensor modules or devices may collect data at the same time with identical frequency, such as ES05 and ET03 in Figure 2 sharing the same timestamps. Since their timestamps have no need to repeat for each time series, we propose to find those time series and align them for efficient storage by eliminating their similar timestamps [21]. Of course, if the timestamps do not exactly match, it may incur exact cost of storing null values, as shown in Figure 5(b). It is automatically determined whether storing individual time series as Figure 5(a) or aligning some of them as Figure 5(b) is more efficient in storage.

4.4 Technique: Frequency Domain Encoding

Finally, frequency domain analysis on time series is widely conducted in signal processing. It is costly to online transform from time domain to frequency domain, e.g., by Fast Fourier Transform (FFT), due to its quasilinear time complexity. Therefore, we propose to store the frequency domain data for reuse, via efficient data encoding in TsFile [57]. Quantization is first conducted on the transformed frequency domain data, with unnecessarily high precision. Moreover, we devise a bit-width decreasing scheme to further reduce the space, referring to the extremely skewed distribution of frequency domain data.

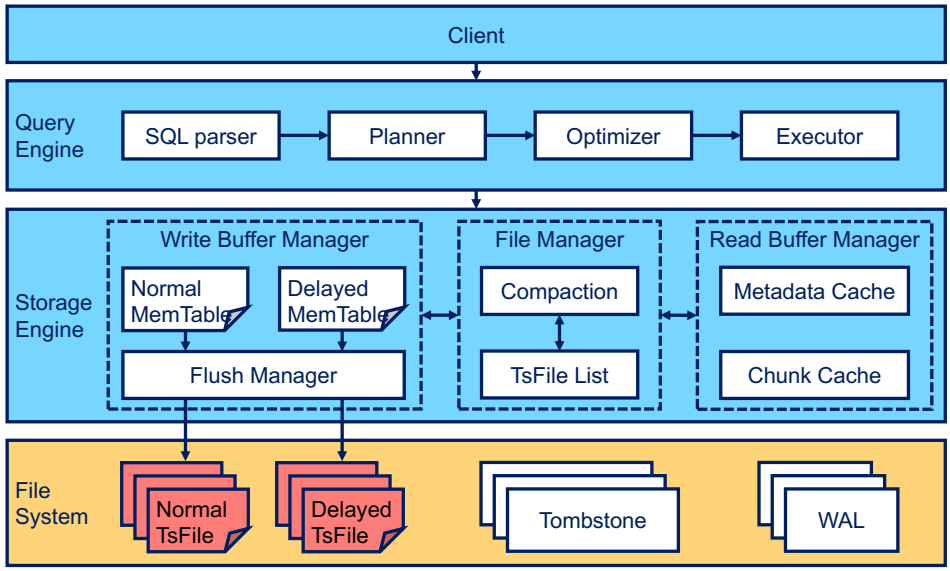


Fig. 7. Architecture of IoTDB engine

5 IOTDB ENGINE

The IoTDB engine design fully considers the following IoT scenarios, (1) delayed arrivals, such that data generated first arrive later, (2) efficient query processing, over various time ranges of miscellaneous series, (3) SQL like queries, extensive for OLTP and OLAP purposes. Figure 7 presents the architecture of IoTDB engine.

5.1 Design: Storage Engine

As shown in Figure 7, the storage engine mainly handles the write, read and management of TsFiles. While the read buffer for caching chunks and metadata are standard, we mainly introduce the special design for IoT data writes below. As introduced in Section 1.1.4, the data arrivals could be delayed in various degree. For example, wind turbines could be occasionally offline for several days, owing to bad weather or network maintenance. When a turbine reconnects, it first reports its current work status to the server and then re-sends data collected in the past several days. The storage engine should be specially designed to handle these delayed arrivals in IoT.

5.1.1 MemTables. To handle intensive writes and delayed arrivals, IoTDB leverages a strategy similar to LSM-Tree [39]. We maintain MemTables in memory for newly inserted data. Different from a MemTable in KV stores that represents one table, a MemTable in IoTDB contains multiple time series for a period, which need to be sorted separately. Before written to MemTable, each insertion will be added to the Write Ahead Log (WAL) to avoid data loss.

For each series family, introduced in Section 3.2.2, we have two active MemTables for data insertion with and without delay, i.e., *delayed* MemTable and *normal* MemTable. To distinguish the delayed data from normal data, we introduce a time detector. It maintains a timestamp for each device, called *stable time*, to record the largest timestamp for each device that has been persistent on disk. For instance, in Figure 8(b), suppose that the point with data time 6 has been written to disk. When IoTDB receives a new data point whose timestamp is greater than the stable time, e.g., the point with data time 7, it is added into the normal MemTable. Otherwise, it is inserted into the

delayed MemTable, such as the point with data time 5, arrived at time 7 as well. In this way, it is guaranteed that all the data points in the normal MemTable have time greater than those persistent on disk, i.e., in-order.

5.1.2 Flushing. MemTables are flushed one by one to a TsFile for data persistence. When we flush a normal MemTable on disks, the MemTable is appended into an unclosed normal TsFile as a block of chunk groups in Figure 6. Similarly, the delayed MemTable will be written into the unclosed delayed TsFile. The flush tasks are asynchronous and managed by the flush manager.

When the size of an unclosed TsFile exceeds a predefined threshold, or periodically, the file will be closed. Its metadata will be flushed to the end of the file. Then a new unclosed TsFile is created for flushing MemTables.

5.1.3 TsFiles. Referring to the aforesaid stable time, since each *normal* MemTable has data time greater than all the previously flushed normal TsFiles, the data time ranges of all normal TsFiles are non-overlapping, denoted by red rectangles in Figure 8(b) and (d). In contrast, *delayed* MemTable contains data points with time smaller than the stable time, i.e., the flushed delay TsFiles, denoted by the blue rectangles in Figure 8(b) and (d), may have time ranges overlapping with the other normal/delayed TsFiles.

Normal TsFiles of a series family form a list ordered by the data timestamp, while delayed TsFiles are in the other list sorted by the ingestion time, a.k.a. version. Two lists in a series family are maintained by file manager for read-write concurrent safety. Later written data always has a larger version, which is indicated by the TsFile generation time and the chunk offset inside TsFile. If a TsFile generation time is larger, then its data is newer. Inside a TsFile, the data is compared by the offset, i.e., a larger offset means newer.

When updating a single data point, we consider it as an insertion operation. The updated value in the insertion operation will be saved into a delayed TsFile. In the reading process, if there are two versions of values for a data point, the version in the delayed TsFile will be adopted. In this way, the updated data takes effect. If more than one version exists in delayed TsFiles, the latest-inserted one will be applied.

5.1.4 Tombstones. Deletion is also supported, e.g., for releasing space in the edge server [37]. The delete operation is often in a form of

```
delete ts where time <= t,
```

which will delete the data of time series `ts` whose time is less than or equal to `t`.

Deletions are recorded as tombstones outside the TsFile, which is append-only by flushing. Notably, deletions in IoT applications are rare. To store these operations, a tombstone file is attached to each TsFile. When receiving a deletion, it is applied to the MemTables first, and then stored into a tombstone file of each TsFile that contains the data to delete. The function of the tombstone file is similar to the differential file [47]. These tombstone files are used to re-apply deletions when querying.

5.1.5 Compaction. According to the aforesaid writing process, time ranges of delayed TsFile and normal TsFile may overlap, which leads to merging the data in multiple files in a query. To accelerate query processing, there is a compaction process to reorganize files in the background. All the three kinds of files, normal TsFiles, delayed TsFiles, and tombstone files, are involved in the compaction process, and will be compacted into new normal TsFiles, which have non-overlapping time ranges. After a compaction process, the corresponding delayed TsFile and tombstone files are removed. In this way, only normal TsFiles with non-overlapping time ranges are left eventually, which accelerates the query problem. The compaction can be triggered periodically in practice.

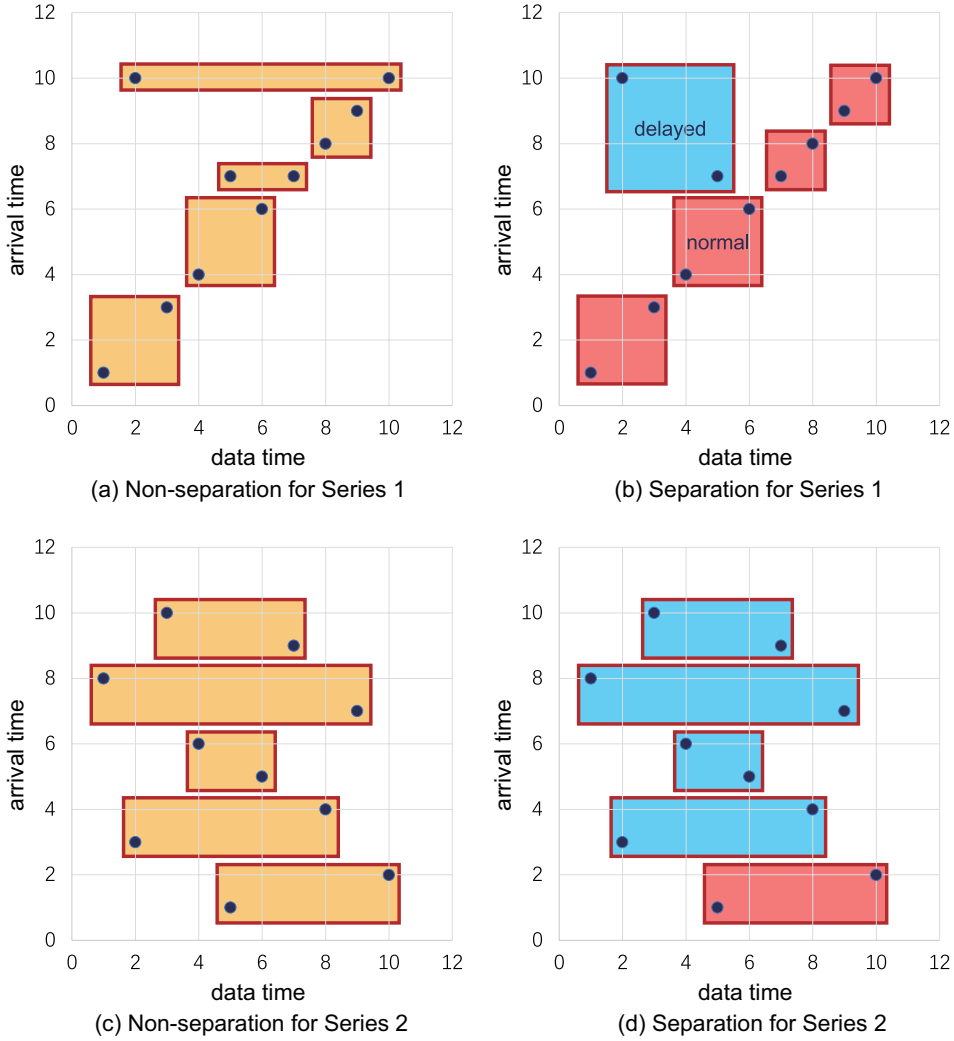


Fig. 8. Separating delays (blue rectangles) works better in (b) than non-separation in (a), but not in (d) compared to (c)

5.2 Technique: Automatic Delay Separation

When there are only a small proportion of delayed data arrivals, i.e., most data are in normal TsFile without time range overlapping, the separation of delayed data helps in query processing and write amplification. To merge the delayed TsFile, we only need to consider those with time ranges covering some points in the delayed one. For instance, for the delayed TsFile denoted by blue rectangle in Figure 8(b), the two data points only need to merge with two left-bottom normal TsFiles in red. In contrast, for the conventional LSM-tree without separation in Figure 8(a), many files are with overlapping data time and need to be merged.

However, for the cases where most data are disordered, i.e., no clear relationship between data time and arrival time as in Figure 8(c) and (d), it is more like the conventional LSM-tree scenario

where data arrival is not related to the key (time). In this case, the separation of delayed data has no advantage but incur extra cost of maintaining two types of MemTables/TsFiles.

Therefore, we propose to automatically determine whether separating the delayed data or not, by observing the data arrivals [35]. It is not surprising that both the write amplification and query processing are improved, by avoid overlapping TsFiles when possible.

5.3 Design: Query Engine

Like typical DBMS, the query engine translates SQL queries, introduced in Section 5.4, into operators that are executable in the database. As illustrated in the architecture in Figure 7, the SQL parser generates an Abstract Syntax Tree (AST) of the input SQL from the client. The planner analyzes the AST and generates the query execution plan, which consists of many operators. The optimizer transforms the query execution plan into an optimized form, e.g., by expression compression. The executor receives a query execution plan and conducts the query task to return data. In the following, we introduce the four main query operators in IoTDB.

5.3.1 Selection Operation with Time Predicates. It is common to query a time series with predicates in the time dimension, e.g.,

```
select WindSpeed from Turbine.Beijing.FU01.AZQ01 where
time > 2019-01-01 11:00:00 and time < 2019-02-01 11:00:00.
```

It queries the time series of wind speed within a month. In execution, the query engine builds a series reader to fetch the relevant data from memory and disk, and apply the time predicates on each point to return the query result. When fetching relevant data in memory, the query engine directly extracts the data points of the selected time series that exactly meets the time predicates from MemTables.

To speed up fetching data on disk, the query engine uses different levels of statistics as described in Section 4.1 to filter out irrelevant data at different levels of granularity. Firstly, it uses the file-level statistics of the file metadata, to skip TsFiles that do not contain the selected time series or do not intersect with the query time range. Then, it reads the chunk-level index of the remaining TsFiles, and uses the chunk-level statistics to skip chunks that do not meet the time predicates as a whole. The remaining chunks are read from disk using the chunk position information in the chunk-level index. Finally, the query engine deserializes the page-level statistics and skips deserializing the remaining page data which do not meet the time predicates as well.

5.3.2 Selection Operation with Value Predicates. Predicates on the value dimensions are also widely specified, e.g.,

```
select WindSpeed from Turbine.Beijing.FU01.AZQ01
where Temperature < 0 and WindSpeed > 0.
```

It queries the time series of wind speed when the temperature is below zero and wind speed is above zero. During query execution, a time generator component is first constructed based on the predicates. The time generator is a binary expression tree, whose leaf nodes are series readers of the relevant time series. The time generator will generate timestamps that satisfy the predicates. The series reader with value predicates works in a similar way as that with time predicates introduced in Section 5.3.1. The only difference is that when value predicates are involved, only statistics of data that do not overlap with any other data on time ranges can be used to help filter out irrelevant data.

Secondly, a series reader is built on the projected time series to retrieve points under those generated timestamps. The query process is optimized in the following two ways. (1) In a series reader, we mark the position of the data point that is currently being read. Then we can get the

next data point quickly because the reading process always marches forwards in time increasing direction. (2) When a time series in the predicates is also selected, we cache its values under the generated timestamps when the time generator works. The cached data can be used later as the retrieved data for the selected time series, avoiding repeated data retrievals.

5.3.3 Alignment Operation of Multiple Series. When multiple time series are queried, for example,

```
select Temperature, WindSpeed, Voltage
from Turbine.Beijing.FU01.AZQ01
```

a multi-way merging algorithm is applied to join these time series by time. In this example, there will be 3 series readers built for these time series. Then a merge join is applied upon those series readers. Analogous to outer join in relational databases, if a series does not have a value at a timestamp, a `null` value will be added in the result, as shown in Figure 5.

5.3.4 Aggregation Operation Using Statistics. To accelerate the execution of aggregation operators, statistics at different granularities are used, including page, chunk and file statistics, as introduced in Figure 6. Statistics can be directly used to update the aggregation result when the corresponding granularity of data meets the following requirements: (1) all data points satisfy the predicates and (2) no data point is updated or deleted. During execution, the query engine checks the usability of statistics in descending order of granularity, i.e., the file-level statistics is checked first, then the chunk-level, and finally the page-level. Benefiting from these statistics, it is usually not necessary to retrieve all raw data.

5.4 Design: TSQL

TSQL is a SQL-like interface for users to manage time series data in IoTDB. While the `insert` statement is the same as in RDBMS, we mainly introduce the rich queries on time series data, often encountered in OLTP and OLAP tasks in industrial IoT scenarios.

5.4.1 Range Query. Query by time is the most prevalent need in the IoT scenarios, e.g., to retrieve the wind speeds of the device `Turbine.Beijing.FU01.AZQ01` during the time range `(t1, t2)`.

```
select WindSpeed from Turbine.Beijing.FU01.AZQ01
where time > t1 and time < t2
```

At the same time, users can add value predicates like `WindSpeed < 20` to further filter out the result that they do not care about.

5.4.2 Alignment Query. When multiple time series are selected, they are aligned by timestamp, as exemplified in Figure 5.

```
select Temperature, WindSpeed, Voltage
from Turbine.Beijing.FU01.AZQ01
```

It is indeed a natural outer join on time. Again, predicates on time and value could be specified in the `where` clause to filter the data.

5.4.3 Aggregation Query. The aggregation query gives a summary of the selected time series by applying an aggregation function on it. For example, the query below gets the number of points in the time series of wind speed from the device `Turbine.Beijing.FU01.AZQ01`

```
select COUNT(WindSpeed) from Turbine.Beijing.FU01.AZQ01
where time > t1 and time < t2
```

IoTDB supports multiple aggregation functions such as `max_value`, `min_value`, `last_time`, `first_time`, `count`, `sum`, `avg`, etc.

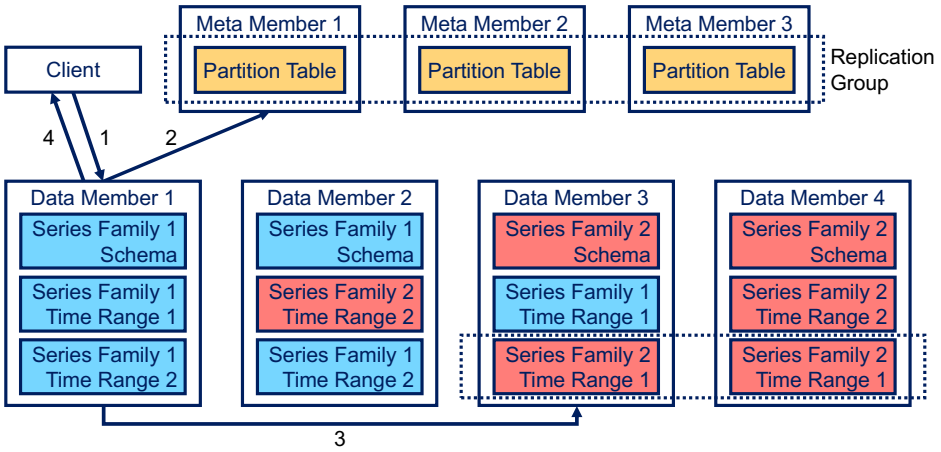


Fig. 9. Architecture of IoTDB Cluster

5.4.4 Group By Time Query. It is also called the *sliding window query*, processing a time series in a period of time with a user-defined resolution. For example, the following query gets the average temperature from 00:00 to 08:00 per day in a month:

```
select mean(Temperature) from Turbine.Beijing.FU01.AZQ01
group by ([2022-01-01, 2022-02-01), 8h, 1d)
```

The process consists of two steps, segmentation and aggregation. To start with, it calculates the first window according to the start time `2022-01-01` and time interval `8h`, i.e., `[2022-01-01 00:00, 2022-01-01 08:00)`. Then, the aggregation of the window is applied. The window will slide by step `1d` and finally return 31 aggregation values to the client.

5.5 Technique: Advanced Extensible Queries

More advanced queries on the time series data for IoT scenarios have been introduced in our preliminary studies and implemented in IoTDB. For instance, the pattern matching query [58] returns the occurrences that are similar to the input sequence, by a `PATTERN` function. The event pattern query [49] finds events satisfying certain occurrence conditions specified by `ESEQ` and `EAND`. The anomaly detection function [63] identifies outliers as well as the corresponding explanations of why being outlying. The data imputation function [50, 62] fills the missing or null values, as a `FILL` function.

To meet customized computing needs for domain specific queries, IoTDB allows users to create User-Defined Functions (UDF) [7]. It takes one or multiple time series as input, and output one time series, which has any number of data points. With UDF, IoTDB provides a variety of built-in functions, such as `SIN`, `LOG10`, and `TIME_DIFFERENCE`. The UDF library [8] now includes more than 60 functions on data profiling [15, 46], data quality [51], frequency domain analysis [57], etc.

6 DISTRIBUTED SOLUTION

While TsFiles can be distributed in HDFS and manipulated by Spark as illustrated in Figure 3, we provide a native solution for better data distribution and query processing. Let us first illustrate the overview of the IoTDB cluster architecture, in Section 6.1. We then present two optimization strategies in writing and reading replicas, in Sections 6.2 and 6.3, respectively.

6.1 Design: Partition and Replication

Both schema and data are partitioned and replicated, e.g., by consistent hashing, for easier load balance. As shown in Figure 9, the partition table is replicated on all nodes of meta members, which store the hash code of all physical nodes and all series family names.

For better distribution, the schema is partitioned by series family, a group of series often processed together as introduced in Section 3.2.2. The data is also first partitioned by series family vertically, then by time range horizontally. As illustrated in Figure 9, there are 2 schema partitions and 4 data partitions, each series family corresponds to one schema partition and has 2 data partitions.

Each schema or data partition is allocated to N nodes by consistent hashing, where N is the replication factor ($N = 2$ in Figure 9). For each partition, the same color rectangles in the dashed box form a replication group. In each replication group, we use consensus protocol to maintain consistency across multiple replicas. All the partition tables in meta members form a replication group as well.

When writing data into IoTDB, (1) the client sends the request to any data member. (2) The data member gets the partition information from the meta member. (3) The request is then routed to all replicas for data consistency. (4) Finally, the origin data member returns the response to the client. Query is processed similarly.

6.2 Technique: NB-Raft for Replication

Consensus protocols are essential in efficiently keeping replicas consistent in distributed systems. Raft consensus protocol [40] models data as a continuous log without holes, making the state space smaller than other protocols, and thus is easier to understand and maintain. Unfortunately, the log serialization scheme of Raft limits its parallelism and consequently the throughput. A typical IoT application manages thousands of devices, samples millions of time series, and generates tens of GBs data every second. Directly applying Raft to IoT scenarios pushes the protocol throughput to its limits, and thus potentially shut out data from the system.

To increase parallelism and throughput, we propose a novel adaption, *Non-Blocking Raft* (NB-Raft) [33], for high throughput IoT data. Intuitively, once an entry arrives at enough followers (a quorum), rather than waiting for other entries, we may first notify the client so that it can issue the next request. By introducing a new intermediate state WEAK_ACCEPT for entries, the "Early Return" is enabled. Instead of blocking the subsequent entries, multiple entries from the same client can thus be processed in parallel.

6.3 Technique: DYNAMIC Read Consistency

While the aforesaid NB-Raft increases parallelism, the "Early Return" may lead to consistency issues among replicas. With write consistency level QUORUM, i.e., persistence in quorum replicas will be regarded as a successful write for efficiency, we may need to retrieve the quorum replicas for the correct results, a.k.a. read consistency level QUORUM.

Note that update of a sensor reading rarely occurs in practice. That is, replicas are mostly consistent with each other, without two versions of the same data. Intuitively, rather than waiting for the remaining replicas to response, we propose a DYNAMIC read consistency level [53], returning results based on a part of the replicas responded thus far. Remarkably, the number of replicas to retrieve could be dynamically determined referring to the confidence of obtaining the correct results.

Table 3. Dataset summary

Dataset	# time series	# data points	total size
TY-vehicle	28,781	1,511,347,655	56G
ZC-train	93,284	46,708,284,999	248G
ZY-machine	111,027	17,580,821,079	735G
TSBS	300,000	92,897,280,000	70.8G

7 EXPERIMENTS

We compare TsFile and IoTDB separately to the state-of-the-art file formats and time series databases that are widely used in industry.

7.1 Experimental Settings

7.1.1 Hardware. For the stand-alone version, we conduct the experiments on an 8-core Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz machine with 32GB memory and 10T HDD, running 64-bit Ubuntu Server 16.04.7 LTS. We also use another PC as the client in the database experiment. The machines are connected with 10GB network.

For the distributed version, we use machines from Alibaba Cloud with Intel Xeon (Cascade Lake) Platinum 8269 2.5 GHz/3.2 GHz CPU, 256GB memory and PL1 ESSD.

7.1.2 Baselines. We compare TsFile v0.13.0 with two columnar file formats: Parquet v2.9.0 [42] and ORC v1.5.4 [27, 41].

For time series database competitors, we compare IoTDB v0.13.0 in three typical categories: time series native InfluxDB v1.8.10 [30], KairosDB v1.2.2 [34] on NoSQL database Cassandra v3.11.2 [36], and TimescaleDB v1.7.5 [54] on RDBMS PostgreSQL v12.7 [45].

7.1.3 Datasets. We employ three real datasets collected by our industrial partners and a time series benchmark, as listed in Table 3.

TY-vehicle. The TY-vehicle data are collected from excavators and delivery vehicles of Tianyuan Technology, an engineering machinery company. The sensors monitor vehicle speed, fuel gauge readings, acceleration, and 1033 other measurements. The data set is very sparse as the time series has varying frequencies. By aligning time series as in Figure 5, only 4.79% data are not null.

ZC-train. The ZC-train data are collected by monitoring trains in Shanghai subway, manufactured by the CRRC Corporation Limited. The 4705 measurements include temperature, speed, load, etc. Because data are collected and uploaded by trains using a batched manner, the dataset is relatively dense and 72.8% of it is not null.

ZY-machine. The ZY-machine dataset consists of the production logs of machines in China Tobacco. 117 indices like water content, batch number, and rod number are recorded to monitor the production. Since there are different types of machines working concurrently, the dataset is also very sparse, with 88.5% null values.

TSBS. The TSBS dataset is generated by the commonly used Time Series Benchmark Suite [55], consisting of 400 hosts, each managing 9 devices (3600 devices in total). The devices take samples every 15 seconds without any null values. The time range of the dataset is 40 days, making the total size of the data set 70.8GB.

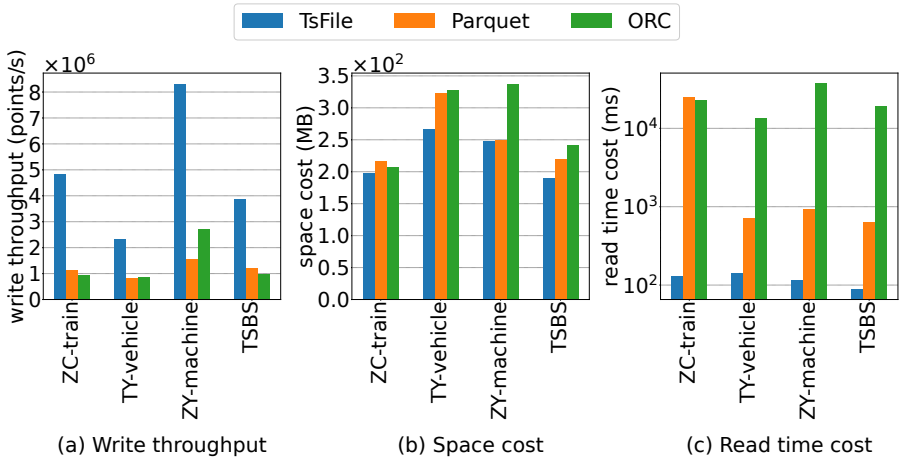


Fig. 10. File evaluation

7.1.4 Queries. We consider four main query types.

Q1 – Time range query

```
select series from device where time > ? and time < ?
```

Q2 – Alignment query

```
select series1, series_2, ... from device
```

Q3 – Aggregation query

```
select count(series) from device
```

Q4 – Downsampling query

```
select count(series) from device group by ([start_time,
end_time), sampling_interval)
```

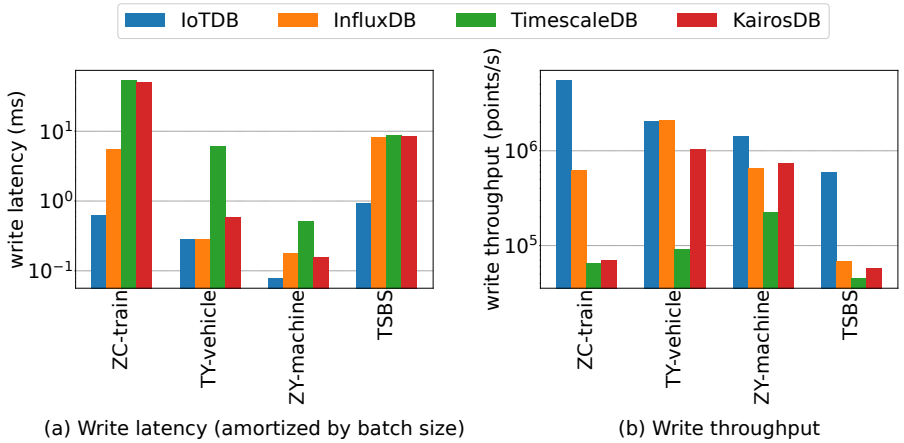
7.2 File Comparison

We evaluate the performance of TsFile, Parquet, and ORC on write throughput, disk occupation, and read time cost. Figure 10 reports the results over four data sets.

We use a non-aligned schema, {time, deviceId, measurementId, value}. This schema avoids storing nulls in the file. It is storage efficient for IoT scenarios where null values are generated due to inconsistent sampling frequencies or various start points of different sensors and devices. Since Parquet and ORC are not specially designed for IoT scenarios, they store deviceId and measurementId as normal columns. In contrast, TsFile stores them in chunk headers and thus duplicated information are significantly reduced.

TsFile uses insertTablet as the writing interface, and Parquet and ORC use their default writing interfaces respectively. All three file formats perform lossless compression and encoding of data. Parquet and ORC have auto-encoding turned on, which can choose the most appropriate encoding method for each column based on the data type and compression efficiency.

In terms of queries, both TsFile and Parquet support filtering and only data that meets conditions are returned. ORC will return the entire block of data that may contain query results, which requires further filtering by the upper layer.



(a) Write latency (amortized by batch size) (b) Write throughput

Fig. 11. Write latency and throughput of various databases

7.2.1 Write Performance. The results in Figure 10(a) show that TsFile has a superior write performance over all other files, as its IoT-aware structure design avoids storing redundant information like deviceId and measurementId. TsFile and ORC provide a vectorized write interface suitable for IoT data ingestions which usually stream in as small batches. Such a vectorized interface improves memory locality and results in a better performance.

The performances are also affected by data type and encoding. Parquet uses dictionary encoding as the first choice, but if the size of dictionary and encoded data exceeds the raw data size, it falls back to another encoding like RLE. Consequently, Parquet writes the same data twice from time to time, and thus has a low throughput. ORC uses a dictionary encoding backed by red-black tree, which is computationally expensive and slows down its performance.

7.2.2 Space Cost. The space cost is reported in Figure 10(b). TsFile has compatible space-efficiency compared with other formats. It avoids storing duplicated deviceId and measurementId as data columns, but builds finer indices which occupy more space. As shown in the following experiments, this sacrifice brings extraordinary improvement over query time, i.e., worthwhile. The duplicated deviceId and measurementId can be removed by dictionary encoding to some extent, but when the number of distinguish ids are large (especially in the TSBS data set), they will still occupy a large space, making Parquet and ORC less feasible.

7.2.3 Read Performance. Remarkably, the read performance of TsFile is clearly better than all the competitors in all tests, as illustrated in Figure 10(c). The results are not surprising, given the indices and statistics stored in TsFile. Such a significantly lower read time cost verifies again the necessity of developing a time series native file format. ORC is slower due to its absence of supporting filters, and the application-level filtering is less efficient.

7.3 Database Comparison

We compare IoTDB with InfluxDB, TimescaleDB and KairosDB on write and query performances. When writing data, IoTDB uses the Java native interface insertTablet(), and the JVM memory is set to 20G. For InfluxDB, time series are stored as one measurement with one tag. The retention policy is default with an infinite duration, which results in one week shard duration [28]. The cache-max-memory-size is set to 0 to avoid writing failure. Influxdb-java client library [29] is used. For TimescaleDB, we create one regular table which includes a time, a series id and the metric

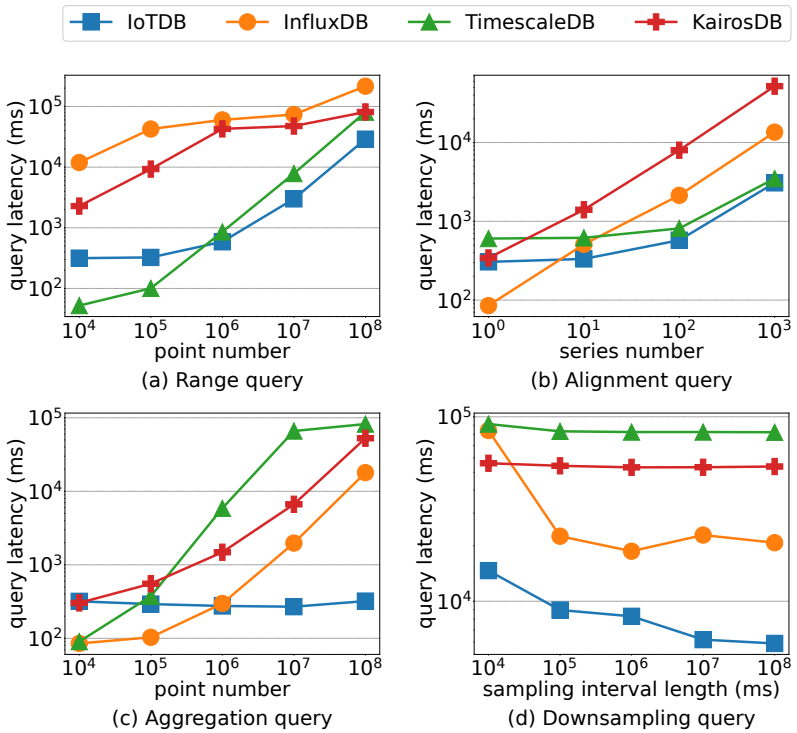


Fig. 12. Query processing over various databases

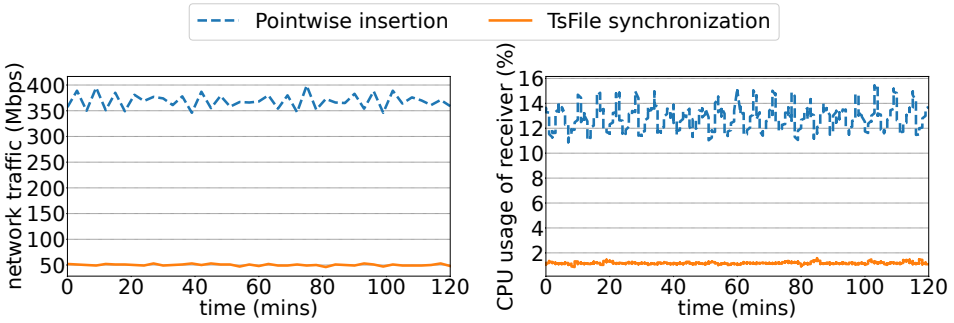
columns. Then we create a hypertable which is partitioned by time. The time interval used for time partitioning in TimescaleDB is one week. We use the default configuration of KairosDB and its backend store Cassandra.

7.3.1 Write Performance. Figures 11(a) and 11(b) present the write latency and throughput, respectively. In general, IoTDB shows better performance in almost all tests, higher write throughput as well as lower write latency. The improvement by IoTDB is limited compared to InfluxDB, in the TY-vehicle data in Figure 11. The reason is that this dataset contains a large proportion of strings, which is not specialized by either system, and thus the systems do not show much difference. As both IoTDB and InfluxDB use time-series-oriented storage implementation, they are better tuned for time-series ingestion than PostgreSQL of TimescaleDB or Cassandra of KairosDB.

7.3.2 Query Processing. The query performance is affected by the number of data points and time series. Therefore, we use benchmark to generate 100 time series, including 10 devices, each device with 10 sensors. In particular, each time series contains about 3 years of data, for a total of 10 billion data points in the entire dataset. The data type is Double, and the time interval between adjacent data points is 1 second.

Figure 12 reports the results of range query, alignment query, aggregation query, and downsampling query. The experiments vary query time ranges, the number of time series (sensors), and sampling intervals.

For time range query in Figure 12(a), the query latency of IoTDB is higher when the queried time range is small. The reason is that the IO unit of IoTDB is a chunk. The default size of chunk is equal or larger than 64K. Therefore, although the queried time range is small, IoTDB still needs to



(a) Network traffic (b) CPU usage of receiver

Fig. 13. Performance of two synchronization strategies

read a chunk and unpack a data page, thus reading more data than the query wanted. Although more data are read, the query latency is still acceptable. For the same reason, the aggregation query of IoTDB shows higher time cost for time range 1h in Figure 12(c).

The improvement of IoTDB is not significant when the number of queried sensors is small in Figure 12(b). The reason is that the alignment query needs to start new threads for each series, which is more meaningful when queried series reaches a certain amount.

Thanks to the multiple level statistics of IoTDB, including page-level, chunk-level, and file-level, down-sampling queries can use higher level statistics when the sampling interval becomes larger. Consequently, the latency is reduced as it does not need to read disk chunks or decode pages, which is shown in Figure 12(d).

The results demonstrate that IoTDB shows better performance when the queried data are large in scale. Remarkably, the superiority of IoTDB is especially significant in aggregation over large-scale data, where the pre-computed statistics in TsFile (introduced in Section 4.1) perform.

7.4 Distributed Solution Evaluation

The evaluation of the distributed solution is in two aspects, (1) how the compacted TsFile improves data synchronization between nodes in a cluster, and (2) how the IoTDB cluster preforms compared to the existing solution.

7.4.1 Sync between IoTDB nodes. There are two ways of synchronizing data between IoTDB nodes, (1) Pointwise insertion, inserting points one by one; (2) TsFile synchronization, loading TsFiles without data packing and unpacking. To compare these two methods, we deploy IoTDB in two nodes N_s and N_r. First, 50 clients write 100 billion data points to N_s. Then, N_s synchronizes data to N_r either in Pointwise or TsFile. The data set consists of 20 series families and 100 thousand devices, with 100 sensors per device. The batch size is 100 points. Considering each raw data point has 16 bytes (8 bytes for the timestamp and 8 bytes for the double value), there are 1600 GB raw data in total. All the TsFiles occupy 389GB, which means the compression ratio is around 4.1:1.

Figure 13 illustrates the network traffic and CPU usage of the receiver. It is not surprising that TsFile synchronization shows significantly lower network and CPU costs. The results verify the motivation of ETL-free data synchronization in the Introduction.

The network traffic of TsFile synchronization is about 50Mbps while the pointwise is 370Mbps. It is because TsFile is a compressed columnar format, whose compression ratio is high compared to the raw data. However, the network cost between the two synchronization methods is around 7.4:1,

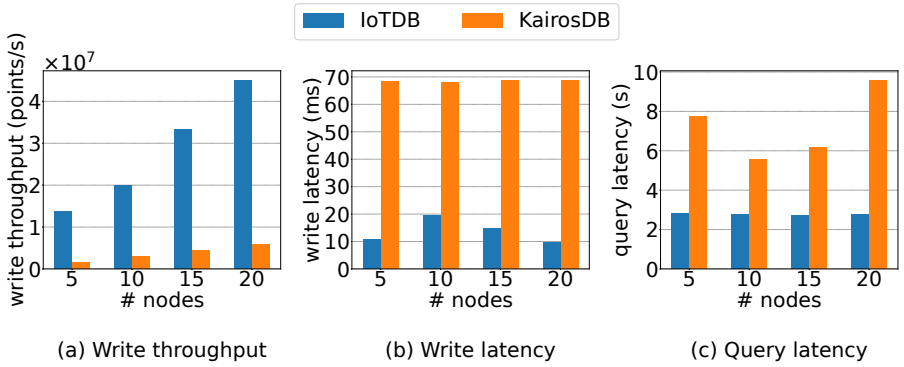


Fig. 14. Distributed solution evaluation

which is higher than 4.1:1. It is because for each synchronization operation in the pointwise way, not only 16 bytes are sent but also extra contents like the time series id (i.e., the device name, and the sensor name).

7.4.2 IoTDB Clusters. For the IoTDB cluster, we compare it with the KairosDB cluster, which is open source. We again use benchmark to write 10 million time series with 0.1 million devices and 100 sensors in each device. The data type is still Double, and the time interval between adjacent data points is 1 second. The number of benchmark nodes is the same as database nodes. Each benchmark node is associated with 10 clients, i.e., the total number of clients varying from 50 to 200.

Figure 14 reports the write and read performance on different nodes in the cluster, varying from 5 to 20. The replication factor is 3 for both databases. IoTDB shows about an order of magnitude improvement on write throughput compared to KairosDB, and much lower write latency. Meanwhile, the query latency of IoTDB is only about 36% of KairosDB.

The write latency of KairosDB is stable in Figure 14(b), while IoTDB first increases and then drops with the increase of the number of nodes. The reason is that data partition of series family by consistent hashing is not evenly distributed, leading to some hot nodes with heavy loads, expected to be optimized in future study.

8 RELATED WORK

8.1 File Storage

The data placement structure affects the data management system performance in a fundamental way [26]. For the requirements of big data analysis, there are many file formats presented, such as Parquet [42], Avro [13], ORCFile [27], and RCFile [26]. They integrate well with the data analysis systems, such as Impala [20], SparkSQL [12], MapReduce [18], Spark [61], etc. Unfortunately, these big data processing oriented file formats do not fit well the IoT applications. They do not consider the features special in the IoT scenarios as summarized in Section 1.1 for both data organization and encoding.

8.2 Time Series Database

Jensen et al. [31] provide a comprehensive survey of time series management systems. In addition to the time series native InfluxDB [30], RDBMS adapted TimescaleDB [54] and NoSQL adapted KairosDB [34] (OpenTSDB [48] also in this category), compared in Section 7.3, there are also

Table 4. Data scale that Apache IoTDB manages in industry

Company	Industry	# time series	# data points	time range
GW	Turbine	4 million	36 trillion	2 years
CRRC	Ship	2 million	63 trillion	1 year
XTXY	Energy	15 k	962 billion	2 years
CTHIC	Tobacco	0.2 million	6 trillion	1 year
CCAG	Auto	10 million	78 trillion	3 months
AUTOAI	Auto	24 million	187 trillion	3 months
TY	Machinery	8 million	92 billion	8 months
BMW	Auto	9 k	56 trillion	1 year

other alternatives. The in-memory solutions, such as Gorilla [43] and Beringei [1] by Facebook, or Monarch [2] by Google, are obviously not economic enough for the traditional manufacturing industry to persistent the whole historical data. On the other hand, model-based approaches, like ModelarDB [32] and Plato [38], are often lossy (though efficient), and again do not meet the requirements of precise monitoring and control in industry.

9 CONCLUSIONS

In this paper, we present a new time series management system, Apache IoTDB, with an open architecture specially designed to support both real-time query and big data analysis for IoT applications. The system includes a new time series file format, TsFile, with columnar storage of time and value to avoid null values and enable effective compression. Remarkably, without costly ETL, the same TsFile could be written in end devices, queried in databases in the edge server, and analyzed in big data systems in the cloud. Built upon TsFile, the IoTDB engine uses an LSM-tree like strategy to process extremely intensive writes and handle delayed data arrivals, very prevalent in IoT scenarios. A rich set of extensible queries, together with pre-computed statistics in TsFiles, enable both efficient OLTP and OLAP tasks in IoTDB. The results show that IoTDB can manage millions of time series, terabytes (TBs) of data, 10 million points insertion per second, 1-day data selection of 0.1 million points and 3-year data aggregation over 10 million points in 100 milliseconds.

To the best of our knowledge, IoTDB has been successfully deployed in more than 200 major companies. Table 4 illustrates some statistics on typical deployments in industrial applications. In particular, Alibaba Cloud as a leading cloud service provider hosts many intelligent manufacturing applications. Previously, InfluxDB was provided as the time series database product in the service, which unfortunately cannot meet the query latency requirement of customers. Therefore, Redis was provided additionally for the query on the latest data. Recognizing the superiority of IoTDB, the InfluxDB+Redis solution has now been replaced. Built upon Apache IoTDB, it becomes the time series database product Lemming provided in Alibaba Cloud.

ACKNOWLEDGMENTS

This work is supported in part by National Key Research and Development Plan (2021YFB3300500), National Natural Science Foundation of China (62021002, 62072265, 62232005), 31511130201, Beijing National Research Center for Information Science and Technology (BNR2022RC01011), and Alibaba Group through Alibaba Innovative Research (AIR) Program. Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

REFERENCES

[1] Beringei: A high-performance time series storage engine | engineering blog | facebook code.

[2] C. Adams, L. Alonso, B. Atkin, J. Banning, S. Bhola, R. Buskens, M. Chen, X. Chen, Y. Chung, Q. Jia, N. Sakharov, G. Talbot, N. Taylor, and A. Tart. Monarch: Google's planet-scale in-memory time series database. *Proc. VLDB Endow.*, 13(12):3181–3194, 2020.

[3] M. P. Andersen and D. E. Culler. Btrdb: Optimizing storage system design for timeseries processing. In A. D. Brown and F. I. Popovici, editors, *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*, pages 39–52. USENIX Association, 2016.

[4] Apache Hadoop. https://hadoop.apache.org/.

[5] Apache HBase. Apache hbase home page. http://hbase.apache.org/.

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

[7] Apache IoTDB. https://iotdb.apache.org/UserGuide/Master/Process-Data/UDF-User-Defined-Function.html.

[8] Apache IoTDB. https://iotdb.apache.org/UserGuide/Master/UDF-Library/Quick-Start.html.

[9] Apache Kafka. https://kafka.apache.org/.

[10] Apache Storm. http://storm.apache.org/.

[11] Apache Zeppelin. https://zeppelin.apache.org/.

[12] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: relational data processing in spark. In T. K. Sellis, S. B. Davidson, and Z. G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1383–1394. ACM, 2015.

[13] AVRO. https://avro.apache.org/docs/current/.

[14] F. Beneventi, A. Bartolini, C. Cavazzoni, and L. Benini. Continuous learning of HPC infrastructure models using big data analytics and in-memory processing tools. In D. Atienza and G. D. Natale, editors, *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*, pages 1038–1043. IEEE, 2017.

[15] Z. Chen, S. Song, Z. Wei, J. Fang, and J. Long. Approximating median absolute deviation with bounded error. *Proc. VLDB Endow.*, 14(11):2114–2126, 2021.

[16] Y. Collet et al. Lz4: Extremely fast compression algorithm. *code. google. com*, 2013.

[17] I. J. Cox. Increasing the bit packing densities of optical disk systems. *Applied optics*, 23 19:3260–1, 1984.

[18] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[19] P. Deutsch. GZIP file format specification version 4.3. *RFC*, 1952:1–12, 1996.

[20] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, S. Legg, and K. Kavukcuoglu. IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures. In J. G. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 1406–1415. PMLR, 2018.

[21] C. Fang, S. Song, H. Guan, X. Huang, C. Wang, and J. Wang. Grouping time series for efficient columnar storage. In *ACM SIGMOD International Conference on Management of Data, SIGMOD*, 2023.

[22] C. Fang, S. Song, and Y. Mei. On repairing timestamps for regular interval time series. *Proc. VLDB Endow.*, 15(9):1848–1860, 2022.

[23] C. Fang, S. Song, Y. Mei, Y. Yuan, and J. Wang. On aligning tuples for regression. In *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 14 - 18, 2022*, pages 336–346. ACM, 2022.

[24] S. W. Golomb. Run-length encodings (corresp.). *IEEE Trans. Inf. Theory*, 12(3):399–401, 1966.

[25] S. H. Gunderson. Snappy: A fast compressor/decompressor. *code. google. com/p/snappy*.

[26] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 1199–1208. IEEE Computer Society, 2011.

[27] Y. Huai, A. Chauhan, A. Gates, G. Hagleitner, E. N. Hanson, O. O'Malley, J. Pandey, Y. Yuan, R. Lee, and X. Zhang. Major technical advancements in apache hive. In C. E. Dyreson, F. Li, and M. T. Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1235–1246. ACM, 2014.

[28] InfluxData. https://docs.influxdata.com/influxdb/v1.7/concepts/glossary/#retention-policy-rp/.

[29] InfluxData. https://github.com/influxdata/influxdb-java/.

[30] InfluxData. Influxdb home page. https://www.influxdata.com/time-series-platform/influxdb/.

[31] S. K. Jensen, T. B. Pedersen, and C. Thomsen. Time series management systems: A survey. *IEEE Trans. Knowl. Data Eng.*, 29(11):2581–2600, 2017.

[32] S. K. Jensen, T. B. Pedersen, and C. Thomsen. Modelardb: Modular model-based time series management with spark and cassandra. *Proc. VLDB Endow.*, 11(11):1688–1701, 2018.
[33] T. Jiang, X. Huang, S. Song, C. Wang, J. Wang, R. Li, and J. Sun. Non-blocking raft for high throughput iot data. In *IEEE International Conference on Data Engineering, ICDE*, 2023.
[34] KAIROSDB. https://kairosdb.github.io/.
[35] Y. Kang, X. Huang, S. Song, L. Zhang, J. Qiao, C. Wang, J. Wang, and J. Feinauer. Separation or not: On handing out-of-order time-series data in leveled lsm-tree. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*, pages 3340–3352. IEEE, 2022.
[36] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Oper. Syst. Rev.*, 44(2):35–40, 2010.
[37] L. Li, J. Liu, J. Fan, X. Qian, K. Hwang, Y. Chung, and Z. Yu. SOCA-DOM: A mobile system-on-chip array system for analyzing big data on the move. *J. Comput. Sci. Technol.*, 37(6):1271–1289, 2022.
[38] C. Lin, E. Boursier, and Y. Papakonstantinou. Approximate analytics system over compressed time series with tight deterministic error guarantees. *Proc. VLDB Endow.*, 13(7):1105–1118, 2020.
[39] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.
[40] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (Usenix ATC 14)*, pages 305–319, 2014.
[41] ORCFile. https://orc.apache.org/.
[42] PARQUET. http://parquet.apache.org/documentation/latest/.
[43] T. Pelkonen, S. Franklin, P. Cavallaro, Q. Huang, J. Meza, J. Teller, and K. Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proc. VLDB Endow.*, 8(12):1816–1827, 2015.
[44] PI. https://docs.osisoft.com/bundle/pi-server/page/the-structure-of-pi-af-asset-models_2.html.
[45] PostgreSQL. https://www.postgresql.org/.
[46] D. Samariya and J. Ma. A new dimensionality-unbiased score for efficient and effective outlying aspect mining. *Data Sci. Eng.*, 7(2):120–135, 2022.
[47] D. G. Severance and G. M. Lohman. Differential files: Their application to the maintenance of large databases. *ACM Trans. Database Syst.*, 1(3):256–267, 1976.
[48] B. Sigoure. Opentsdb: The distributed, scalable time series database. *Proc. OSCON*, 11, 2010.
[49] S. Song, R. Huang, Y. Gao, and J. Wang. Why not match: On explanations of event pattern queries. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 1705–1717. ACM, 2021.
[50] S. Song, A. Zhang, J. Wang, and P. S. Yu. SCREEN: stream data cleaning under speed constraints. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 827–841. ACM, 2015.
[51] Y. Su, G. Yikun, and S. Song. Time series data validity. In *ACM SIGMOD International Conference on Management of Data, SIGMOD*, 2023.
[52] Y. Sun, S. Song, C. Wang, and J. Wang. Swapping repair for misplaced attribute values. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pages 721–732. IEEE, 2020.
[53] Y. Sun, Z. Zheng, S. Song, and F. Chiang. Confidence bounded replica currency estimation. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 730–743. ACM, 2022.
[54] TimescaleDB. https://www.timescale.com/.
[55] TimescaleDB. https://github.com/timescale/tsbs.
[56] C. Wang, X. Huang, J. Qiao, T. Jiang, L. Rui, J. Zhang, R. Kang, J. Feinauer, K. Mcgrail, P. Wang, D. Luo, J. Yuan, J. Wang, and J. Sun. Apache iotdb: Time-series database for internet of things. *Proc. VLDB Endow.*, 13(12):2901–2904, 2020.
[57] H. Wang and S. Song. Frequency domain data encoding in apache iotdb. In *Proceedings of the VLDB Endowment, PVLDB*, 2022.
[58] J. Wu, P. Wang, N. Pan, C. Wang, W. Wang, and J. Wang. Kv-match: A subsequence matching approach supporting normalization and time warping. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 866–877. IEEE, 2019.
[59] J. Xiao, Y. Huang, C. Hu, S. Song, X. Huang, and J. Wang. Time series data encoding for efficient storage: A comparative analysis in apache iotdb. *Proc. VLDB Endow.*, 15(10):2148–2160, 2022.
[60] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli. Druid: a real-time analytical data store. In C. E. Dyreson, F. Li, and M. T. Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 157–168. ACM, 2014.
[61] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In E. M. Nahum and D. Xu, editors, *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*. USENIX Association, 2010.

[62] A. Zhang, S. Song, and J. Wang. Sequential data cleaning: A statistical approach. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 909–924. ACM, 2016.

[63] A. Zhang, S. Song, J. Wang, and P. S. Yu. Time series data cleaning: From anomaly detection to anomaly repairing. *Proc. VLDB Endow.*, 10(10):1046–1057, 2017.

Received 2 February 2023