Apache TsFile: An IoT-native Time Series File Format

Xin Zhao
Tsinghua University
zhao-x19@mails.tsinghua.edu.cn

Jialin Qiao*
Timecho Ltd
jialin.qiao@timecho.com

Xiangdong Huang
Tsinghua University
huangxdong@tsinghua.edu.cn

Chen Wang
Timecho Ltd
wangchen@timecho.com

Shaoxu Song†
Tsinghua University
sxsong@tsinghua.edu.cn

Jianmin Wang
Tsinghua University
jimwang@tsinghua.edu.cn

ABSTRACT

The proliferation of the Internet of Things (IoT) has led to an exponential increase in time series data, distributed and applied in various contexts, demanding a dedicated storage solution. Based on our observations and analysis of IoT production systems, we have characterized 3 requirements for time series data: (1) a close association with devices and sensors, (2) continually synchronizing between cloud-edge, and (3) requiring the ability for high ingestion and low latency access on big volume data. Despite the growing trend, current time series database systems lack a standardized file format, and existing open file formats do not adequately leverage the unique characteristics of IoT time series data. In this paper, we introduce Apache TsFile, a specialized file format tailored for IoT time series data. TsFile organizes data by devices, creating indexes based on device-related information. Our experiments demonstrate the efficiency of TsFile in achieving high data ingestion rates, minimizing latency, and optimizing data compactness.

PVLDB Reference Format:
Xin Zhao, Jialin Qiao, Xiangdong Huang, Chen Wang, Shaoxu Song, and Jianmin Wang. Apache TsFile: An IoT-native Time Series File Format. PVLDB, 17(12): 4064 - 4076, 2024.
doi:10.14778/3685800.3685827

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

1 INTRODUCTION

Time series data are prevalent in the Internet of Things (IoT) scenarios. With the widespread deployment of sensor-equipped devices, a vast amount of time series data are generated to reflect the operational states of these devices. These series serves diverse purposes including simulation design, production manufacturing, and equipment maintenance. For instance, CCS, one of our partners, tracks the time series data throughout whole lifecycle of 30 million of shipbuilding components, storing these data in cluster servers for long-term maintenance and analysis. While another of our industrial partners, ZY, has sensors installed on their rock drilling machines, caching posture and position information on device controller to enable real-time control. Unless otherwise specified, time series and series will be used interchangeably in this paper.

1.1 Motivation

In the aforesaid IoT scenarios, rather than directly storing the time series data in databases such as InfluxDB [18], it is highly desired to first store the time series as files in end devices, and then sync them to edge and cloud servers. The reason is that time series database management systems are often too heavy to be installed in end devices. While SQLite [31] is light enough for end devices, it incurs huge ETL costs to transfer the data from end devices to the cloud, e.g., hosted by InfluxDB.

Some open file formats, such as Apache Parquet [19, 20, 28], have been applied to store time series data. However, they do not recognize and leverage features of time series data in IoT, resulting in performance fallback to some extent. To be more specific, these features include 3 aspects as follows.

1.1.1 *Series Specific Compression.*

As sensors detects physical status like temperature, speed, pressure, or displacement and convert these into digital signals all the time, voluminous time series data has been produced and requires efficient storage. Sensors produce distinct series even when measuring the same type of physical quantity, reflecting variations in the objects being measured. Each time series fluctuates with inherent patterns, adhering to the physical laws underlying its sensor. Selecting a suitable encoding and compression scheme for each series is vital for optimal compactness [39], with each stored separately and contiguously.

However, common file formats, such as Apache Parquet, typically place multiple series of the same physical quantity type into a single column, applying a uniform compression scheme across the entire column. Time series that measure physical quantities of the same type can differ vastly in patterns, leading to additional space overhead due to the uniform compression method. This situation motivates the design in Sect 3.2, which enables each series employ individual encoding and compression scheme.

1.1.2 *Hierarchical Device Identification.*

Once transmitted from sensors to an Industrial PC (IPC) or PLC, time series data are matched with specifications from the point table using a communication address assigned by field engineers during installation, as shown in Figure 1 (a). The identifier of the device, an essential part of the specification, typically possesses a hierarchical structure. For instance, energy and power enterprises employ KKS coding [38] to

*Jialin Qiao is the PMC Chair of Apache TsFile Committee (https://tsfile.apache.org/).
†Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685827

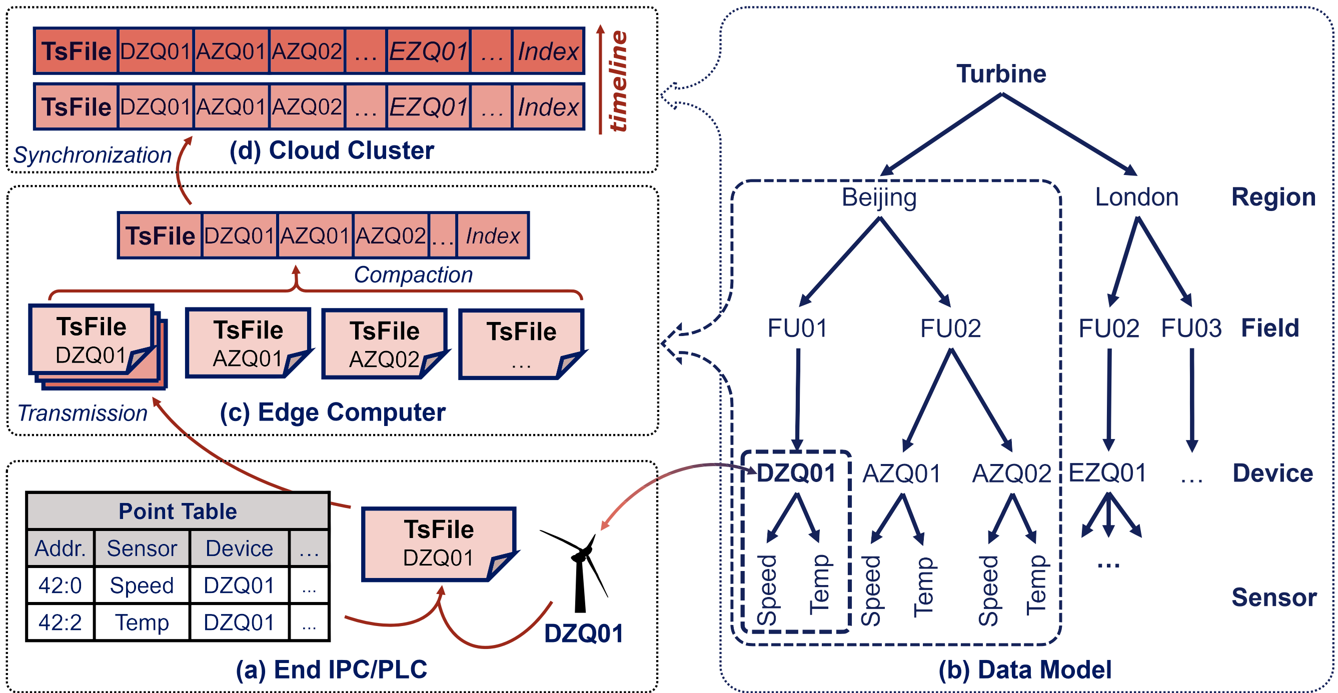

Figure 1: Hierarchy Across Endpoint, Edge and Cloud

categorize and identify devices within a power plant, while the Domain Model in IoT-A [7] presents a self-association of device entity, both exemplifying a hierarchical structure. Figure 1 (b) depicts a company with numerous wind farms across different regions. On this hierarchy, each leaf represents a sensor collecting time series, while the path from the root to the parent of the leaf denotes the identifier of the device, i.e., device ID. The hierarchical structure reveals the relationship between the identification of related time series, and thus leads to the design in Sect 4.2. As the hierarchy naturally represents the entities and relationships in the scenarios, it is also referred as the data model in the following sections.

Device ID remains static throughout the lifecycle of time series while serving as a part of the index for access, thereby ought to be handled distinctively from ordinary time series data. In Parquet and similar open file formats, both the device ID and time series data are stored as ordinary columns without any dedicated indexes. To achieve reasonable latency for series access, these formats resort to sorting rows by device IDs, facilitating binary search upon related columns. However, the repetition of device specifications across numerous rows introduces storage redundancy even with dictionary encoding employed. Moreover, utilizing nested datatype to describe the hierarchy of device IDs increases complexity due to the column-striping and record-assembly algorithms [26].

1.1.3 ETL-free File Compaction. Time series data is typically compacted several times during the synchronization, as shown in Figure 1 (c) and (d). End devices, such as IPCs or PLCs, are commonly resource-constrained and thus store only the latest time series data for real-time control while continuously transmitting this data. Edge computers gather time series from multiple endpoints and compact them into consolidated files for efficiency. Ultimately, cloud servers preserve gross time series data for long-term application, conducting compaction for higher performance.

As Parquet and similar file formats rely on ordering rows by device ID to ensure efficient access, preserving the order throughout compaction is essential but costly. Compacting multiple pages, each belonging to different files and containing interleaved device IDs, into a single consolidated page requires decoding and rewriting, making the compaction rather expensive. This situation motivates a layout where data points from the same time series are stored contiguously, as elaborated in Sections 3.2 and 3.3.

1.2 Contribution

In this paper, we introduce a novel open file format dedicated to time series in IoT scenarios, referred to as *TsFile* (Time Series File). TsFile enhances the entire lifecycle of IoT time series data. On resource-limited endpoint devices, an open file format allows for direct data manipulation, eliminating dependency on additional processes. At the edge level, it reduces the overhead of ETL tasks during data compaction. On cluster servers, directly analyzing extensive time series data from files proves more efficient than executing database system operations [9, 26].

Specifically, the unique IoT features stated in Section 1.1 have shaped the design choices and novelty as below.

(1) TsFile organizes data by series, enabling distinct encoding and compression schemes for each series. This strategy effectively minimizes the space cost for series exhibiting various patterns. Data points within one series are store contiguously, leveraging inherent patterns for enhanced compression. Series originating from the

same device are stored in locality, since they are more likely to be accessed together for joint analysis. As some sensors generate multiple readings at once, a common timestamp sequence is utilized to reduce storage footprint;
(2) TsFile constructs indexes based on device identifiers and sensor names, thoroughly eliminating storage redundancy of identifiers. The index adopts two implementations, based on B-Tree and Trie respectively, leveraging shared prefix among identifiers originated from the hierarchical structure;
(3) As TsFile organizes data by series, and distinct files being compacted, whether at the edges or in the cloud, are disjoint in terms of time range, compaction is simplified to the concatenation of series data and adjustment of index offsets. This approach minimizes deserialization and decoding, which constitute the most expensive part of ETL.

This paper is organized as follows: Section 2 gives a overall perspective of TsFile structure, Section 3 and Section 4 delve into the design principles behind Apache TsFile. Section 5 provides straightforward examples of usage for further comprehension. Section 6 compares TsFile against prevalent open file format and evaluates its design choices. Section 7 explores related research on IoT time series data models and other open file formats. Finally, Section 8 concludes the paper.

2 TSFILE FORMAT OVERVIEW

The overall structure of TsFile is divided into 2 parts: the Data Area and the Index Area, as shown in Figure 2. The Data Area is self-documenting and thus can be independent of the Index Area, in spite of the low efficiency. The Index Area can be implemented in alternative structures to satisfy specific application requirements. This paper only outlines a B-Tree-based implementation.

The Data Area comprises various Chunk Groups, each holding time series data for a device over a specific period. A device may be associated with multiple Chunk Groups, depending on the workload. Within a Chunk Group, each Chunk contains data for a single series. Other than TsFiles resulting from compaction, each Chunk within one Chunk Group is associated with a distinct series.

The Index Area links query conditions, such as identifiers, time, or value ranges, to data offsets in the Data Area. It includes a Bloom Filter to quickly determine the presence of a specific series, thus speeding up searches across multiple TsFiles. The Chunk and Series Indexes are crucial for fast access and will be explored further in subsequent sections.

3 TSFILE DATA AREA

The principle of the data area is to store the data points of each time series in a columnar way to enhance compression efficiency and to provide locality at both the device level and file system block level. This principle differs TsFile from other common open file formats with higher compression ratio and throughput for time series in IoT scenarios.

3.1 Chunk Group

The data area is organized into one or more contiguous chunk groups, with each chunk group corresponding to all time series data from a single device over a period. Devices can be categorized into

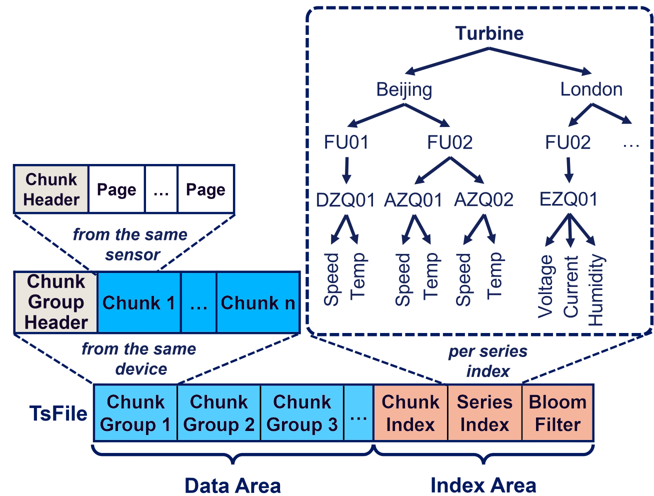

Figure 2: Data Area and Index Area in TsFile

aligned and non-aligned types, and accordingly, chunk groups also fall into these two categories. A chunk group consists of a header and one or more chunks, each chunk stores data from a specific time series. The header of the chunk group stores the identifier of the device, which is the path from the root to the device node in the data model. The concept of chunk groups achieves device-level locality, as different time series from the same device are often queried simultaneously.

Chunk groups are the basic units for flushing TsFile on secondary storage. When data is written to TsFile, it is first buffered in memory. Once the memory usage reaches a threshold, the buffer, which may contain multiple chunk groups, will be flushed to secondary storage. This threshold can be adjusted in line with the file system configuration to deliver block-level locality. For example, adjusting the buffering threshold based on the block size in HDFS can prevent a single chunk group from being stored separately across different blocks.

Common file formats use a tabular structure as the data model, organizing tuples with their ingesting order into row groups as the unit for writing to secondary storage [15, 16, 26]. In contrast, TsFile flushes multiple independent chunk groups once it reaches the memory threshold, with each chunk group corresponding to a distinct device, thereby offering improved locality. Furthermore, different chunks may consist of varying chunks, while different row groups always contain the same set of columns. This feather is beneficial for typical industrial scenarios, as datasets from our partners illustrate in Section 6.1.2. In these scenarios, one file may contain data points from up to thousands of sensors with different names; these sensors are distributed across various devices, with most devices having only a tiny subset of all the sensors. Figure 4 showcases a common scenario where, despite tuples being sorted by device IDs, values from distinct series end up grouped on the same page due to the row-wise grouping strategy, thereby reducing compression efficiency.

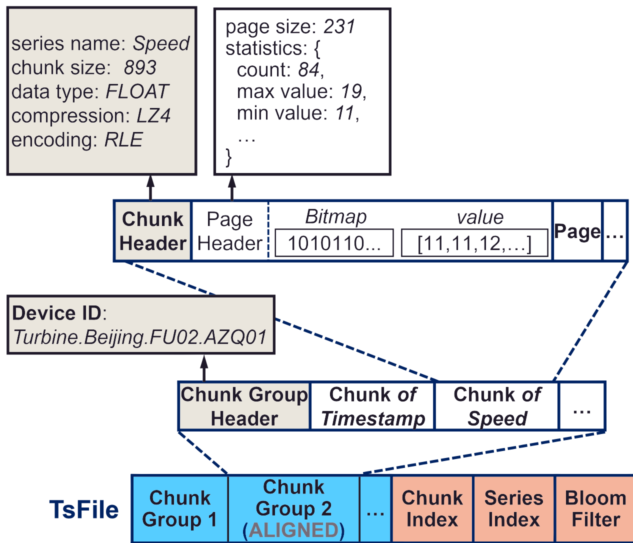

Figure 3: Detail of Data Area

3.2 Chunk

A Chunk consists of a header and one or more pages, storing data of a single time series over a period. The header contains information such as the name of the time series, the data type, the encoding and compression scheme, and the number of pages within the chunk. The time series data in a chunk are sequentially and disjointedly distributed across the pages, ordered by timestamp.

In non-aligned chunk groups, each chunk serializes both timestamps and values into every page. In aligned chunk groups, where all time series of the device share the same sequence of timestamps, there is a specialized chunk stores the sequence of timestamps exclusively, and other chunks store the sequence of value along with a bitmaps indicating null values, as shown in Figure 3.

Each chunk has a corresponding entry in the Chunk Index. When accessing a time series, the system first locates the specific chunk and then evaluates whether to deserialize the pages within the chunk by checking the header of each page. The deserialization, involving decoding and decompression, can introduce significant latency. Deserializing by page rather than by chunk enables a balance between index size and access efficiency.

3.3 Page

A page represents the smallest unit for the serialization or deserialization of time series data, consecutively storing data from a specific series over a given period. IoT time series data, originating from physical states on devices, often exhibit stable values or periodic patterns, thus holding the potential for efficient compression. In contrast, storing time series in the tabular data model of common file formats involves interleaving data values from multiple series in one column, which degrades compression efficiency.

Each page stores its statistical information in the header, enabling fast filtering of irrelevant pages during data access. For non-aligned time series, each page sequentially stores 2 segments: a sequence of timestamps and a sequence of data values. For aligned time series, depending on the type of chunk it belongs to, a page stores only one sequence, either of timestamps or data values.

The ingesting data is first placed in the buffer of the current page. Once the buffer reaches a threshold, the data is encoded, compressed, and written to the buffer of the corresponding chunk. The threshold of a buffer in page is configurable; a higher threshold imposes a higher cost to deserialize a single page even if only a few points are expected, while a lower threshold introduces more fragmented pages thus affecting both the efficiency of locating the target page and data compression efficiency. A reasonable threshold needs to strike a balance between the two.

3.4 Time Series Encoding

Since each data series employs individual encoding and compression schemes, specialized algorithms can more efficiently leverage intervals in timestamps [11], variations in values [12], and patterns in frequency domain [37], compared to ordinary counterparts.

While there are many encoding algorithms available, they haven't fully exploited the characteristics of time series in IoT scenarios [39]. Timestamps from sensor data are typically at fixed intervals, although network delays may introduce variation or loss. To address this, we propose a timestamp encoding method [11] that focuses on these regular intervals, accommodating potential disturbance. Moreover, an effective encoding scheme should exploit not only the patterns within each series but also the intricate relationships that exist among values across various series. Therefore, we have introduced an encoding method based on feature models, which captures similarity and regression relationships among series [12].

When analyzing time series data, frequency domain information plays a crucial role. To avoid performing complex frequency domain transformations, such as Fast Fourier Transform (FFT), with every analysis process, we propose an efficient encoding method that stores the frequency domain of the series directly [37].

4 TSFILE INDEX AREA

The retrieval of time series data typically specify series identifier and target time or value range. Thereby the Index Area mainly includes 2 parts: the Series Index is tasked to locate the entrance of a single series through its identifier, and the Chunk Index is designed for locating the exact chunks containing requested data.

4.1 Chunk Index

Each time series has an entry in the chunk index, which comprises 2 parts as Figure 6 demonstrated. The first part, referred to as the time series metadata (TSM), holds the name and datatype of the corresponding series, along with comprehensive statistical information about the series across the entire file. This information includes maximum and minimum values, as well as the earliest and latest timestamps, among other data. The second part consists of one or more chunk metadata, each stores the statistical information and offset for each chunk. The fields of statistics in both the TSM and chunk metadata are identical.

When flushing chunk groups to a TsFile, the chunk index is maintained in memory. Once all data in the data area has been flushed, the chunk index is then written to the file. When querying data from a TsFile, as Figure 5 shows, the Series Index locates the

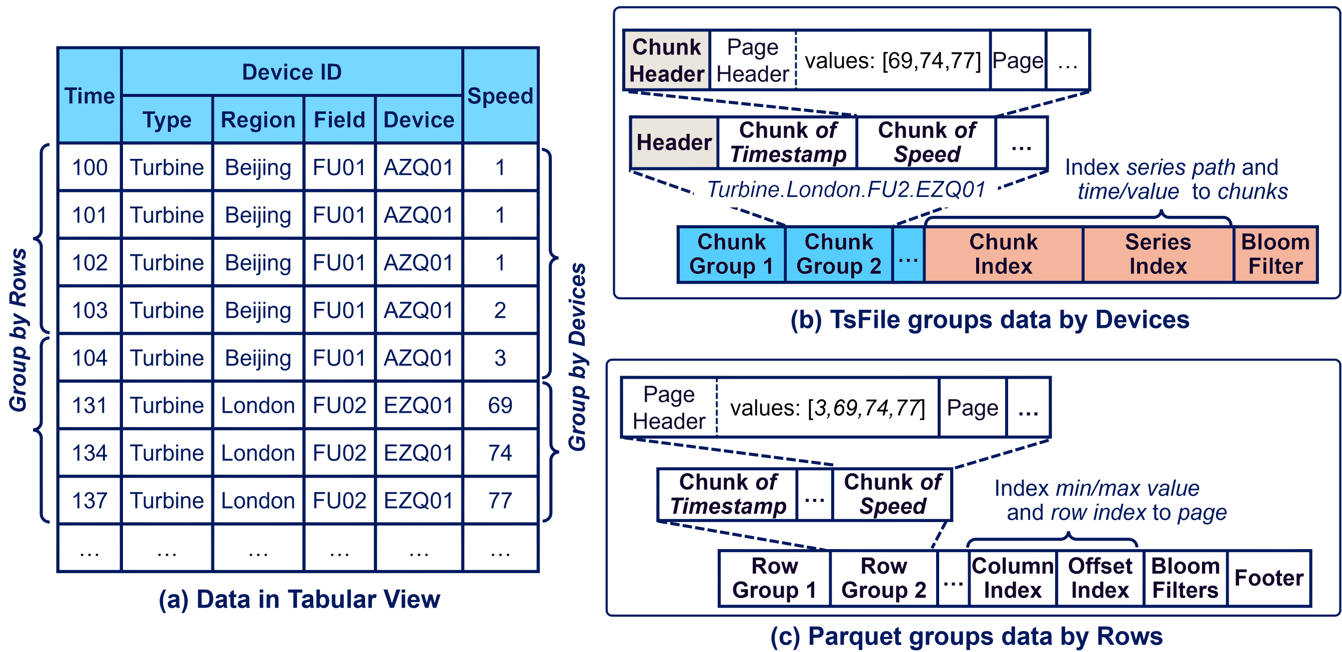

Figure 4: Comparative Example for Data Area

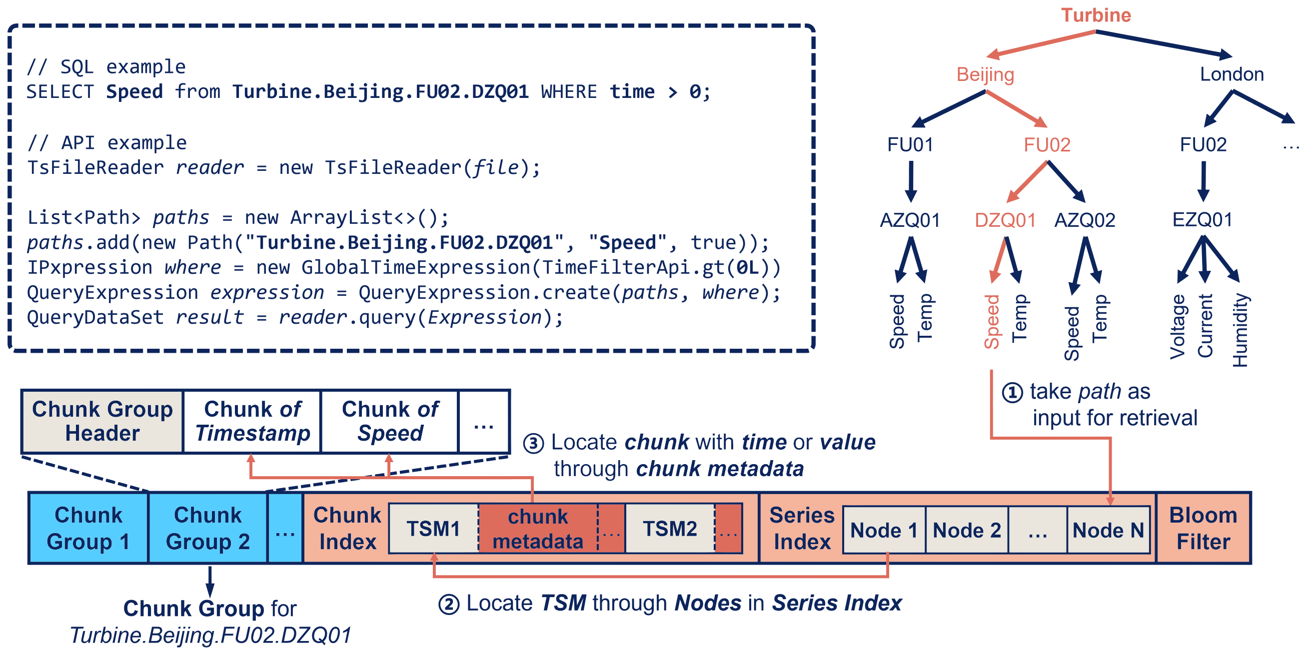

Figure 5: Access from Index Area to Data Area

TSM of the requested series through its identifier. The query process determines the chunks to be accessed by sequentially inspecting the chunk metadata.

Compared to index structures in prevalent open file formats, such as Page Index in Parquet [28], the Chunk Index distinctively indexes only the data within requested series. The count of chunk metadata for a specific series depends solely on its volume in the file, reserving stable access efficiency irrespective of the presence of other series, as Section 6 demonstrate. This approach leverages

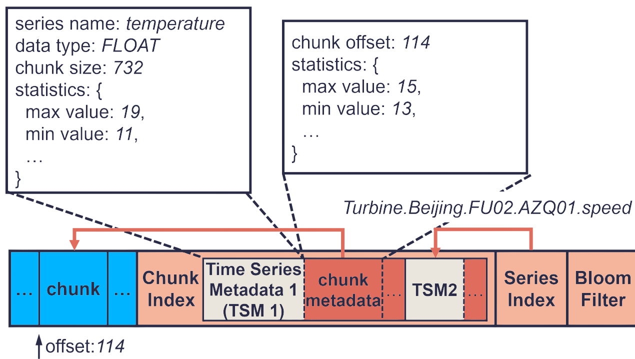

Figure 6: Detail of Chunk Index

the structure of Data Area, where time series data are grouped by devices and sensors.

4.2 Series Index

The Series Index is a composite index structure composed of 2 layers, each being a 256-ary search tree by default. The first layer is a dense index that uses the device ID, which is the path from the root to the device node in the data model, as the index key. The index value is the offset of the root node of a tree in the second layer. The second layer consists of multiple search trees, each indexing the name of a series to the offset of its TSM. Every tree in the second layer indexes only a portion (by default, 1/256) of the series within its corresponding device.

All trees in the Series Index are constructed bottom-up. During the process of writing the chunk index to a file, the offsets and names of the series from the TSMs to be indexed are temporarily buffered in memory. These TSMs serves as the entry of tree nodes, constructing second layer trees from bottom up. Similarly, the offsets of these trees serve as the entries for constructing the tree in the first layer. Ultimately, the offset of the first layer tree is stored in the tail of TsFile, serving as the entry for accessing time series.

The two-layered structure reduces the duplication of device IDs, typically represented by long strings, across its nodes, effectively reducing the footprint of the index. In the second layer of the tree structure, only a subset of the TSMs for each device is indexed. The TSMs that are not indexed are always stored contiguously after the indexed TSMs, and searching for these series requires a linear search starting from the indexed TSMs. In the IoT context, where devices are often equipped with numerous sensors, this sparse indexing approach strikes a balance between space and time.

4.3 Automatic Schema Identification

Consistent patterns exhibited within individual sequence, along with indexes that delineate the relationships between series, facilitate automatic correction of the association between each series and its specification [32], which may be mismatched due to sensor cable misplacement.

In IoT scenarios, field engineers may install new sensors to collect new metrics, introducing additional time series. During device maintenance, engineers might mistakenly connect sensor cables,

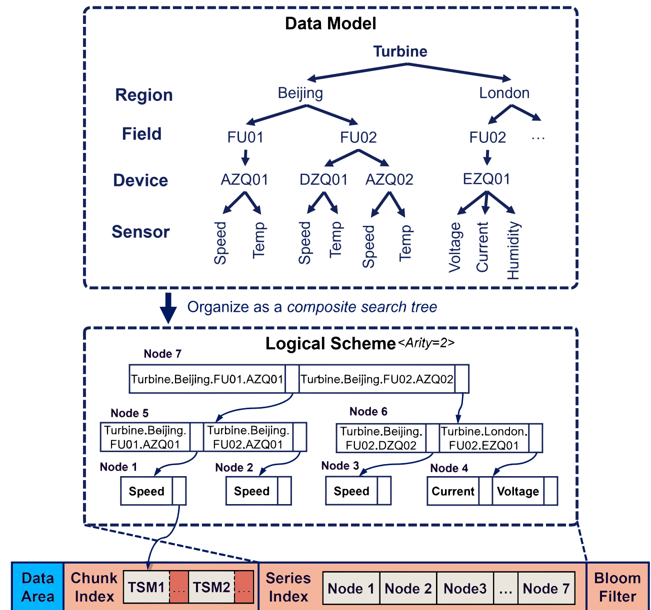

Figure 7: B-Tree Based Series Index

leading to mismatches between time series data and their identifiers, necessitating identification and correction based on data pattern. Previous research has found that such schema issues occur with a probability of about 4% in production environments [32]. Consequently, we propose an automatic method [32] for identifying schema of series to reduce data loss caused by schema issues.

As Section 3 stated, the alignment of time series significantly affects their storage methods. However, due to the aforementioned lack of authority, some series that are actually aligned might be mistakenly stored as non-aligned. Furthermore, given that sensors typically operate at regular sampling rates, certain sensors could have similar sampling timestamps even without being aligned at hardware level, making it efficient to store them as aligned series. Our proposed method [10] evaluates the similarity of series timestamps and automatically groups aligned time series, considering the trade-off with spatial overhead.

5 THE API OF TSFILE

Given its high throughput and efficient storage from columnar organization, combined with the grouping strategy that eases compaction, TsFile is well-suited as SSTable in LSM architecture [27] and thus serves well as storage format in time series database management (TSDBMS), like Apache IoTDB. Moreover, TsFile also provides direct data access through integrated interfaces as follows.

In TsFile, the essential unit of time series data is a quadruple, consisted of a timestamp, a value, and identifications of device and sensor generating the value. Ingest data into TsFile means storing a sequence of these quadruples, while read from TsFile is accessing data through specifying part of them.

5.1 TsFile Writer

TsFile can be created and manipulated on both local and distributed file systems, such as HDFS, integrating seamlessly with the big data ecosystem. The following code example outlines 3 constructors along with its parameter types. Line 1 instantiate a writer with a file descriptor, while line 2 create the writer with a schema describing devices and sensors. `TsFileOutput` in line 3 can be an instance of `HDFSOutput`, which enables TsFile to be store in HDFS.

```
public TsFileWriter(File);
public TsFileWriter(File, Schema);
public TsFileWriter(TsFileOutput, Schema);
```

Constructors below demonstrate details in `Schema`. Among these, `MeasurementSchema` represents the name and physical schema of one series, including datatype, encoding and compression scheme. `MeasurementGroup` describe multiple series within one device and the alignment of the device, while `Schema` holds mappings from device IDs to descriptions of its sensors.

```
public Schema(Map<String, MeasurementGroup>);
public MeasurementGroup(boolean, Map<String,
     MeasurementSchema>);
public MeasurementSchema(
    String, TSDataType, TSEncoding, CompressionType);
```

Each series in TsFile can be assigned with distinct schema, with details are store in the header of each chunks as shown in Figure 3, provided the datatype is compatible with the encoding scheme. This approach offers greater schema flexibility compared to common file formats, which typically stores series with the same name in a single column, applying uniform encoding and compression schemes irrespective of their distinct characteristics.

Before ingesting data from new series, they must be registered in TsFile as follows, which evolvs the schema within the TsFile. Line 7-8 register a time series through specifying its device and schema, after then the series is ready to ingest data.

```
TsFileWriter writer = new TsFileWriter(file);

String device = "Turbine.Beijing.FU01.AZQ01";
MeasurementSchema sensor = new MeasurementSchema(
    "Speed", TSDataType.FLOAT, TSEncoding.RLE);
writer.registerTimeseries(device, sensor);
```

TsFile accepts time series data by `TSRecords` or `Tablets`. The former holds only 1 timestamps, containing multiple values measured at that time from distinct sensors within 1 device. The later submits data points from 1 device in batch, requiring a schema for initiation while providing higher throughput. Line 5-11 shows that a tablet is created with given device and schema list, and the tablet collects data points via arrays of timestamps and values. The evaluations in Section 6 is based on `Tablets` since it represents the ingestion capability of TsFile.

```
TSRecord record = new TSRecord(now(), device);
record.addTuple(new FloatDataPoint("Speed", 1.2);
writer.write(record);

List<MeasurementSchema> schemas = new ArrayList<>();
schemas.add(sensor);
Tablet = new Tablet(device, schemas);
tablet.timestamps[tablet.rowSize] = now();
float[] values = (float[]) tablet.values[0];
values[tablet.rowSize++] = 1.13;
writer.write(tablet);
```

Neither of `Tablet` nor `TSRecord` determines alignment of the device, which is a description on physical level. Aligned and non-aligned devices are logically equivalent, yet they exhibit significant performance differences under specific workloads.

TsFile can not modify or delete data in place, since data from sensor reading is rarely required to update. For those under the TSDBMS, modifications are delivered by tombstones.

5.2 TsFile Reader

There are two ways reading data from TsFile, their constructors are stated below. `TsFileSequenceReader` describes metadata of the TsFile and provides a low level access where data can be located by position and is deserialized from page to page. `TsFileReader` retrieves data with specifications consisted of series identifiers and optional filters. Read with `TsFileReader` requires specifying one or more series. If the process knows nothing about the series inside a TsFile, it is better to access data by `TsFileSequenceReader`. As `TsFileReader` provides more general usage, this paper mainly focus on it and take it for evaluation in Section 6.

```
public TsFileSequenceReader(File);
public TsFileReader(TsFileSequenceReader);
```

`TsFileReader` accepts expressions consisted of specific series paths and filters. Filters can be applied to timestamps or values, and can be composed via logical operators like *and* and *or*.

```
TsFile reader = new TsFileReader(file);
Path series = new Path(device, sensor);
Filter valueFilter = ValueFilterApi.gt(1.1);
Filter timeFilter = TimeFilterApi.gt(now() - 3 * hour);
IExpression filterExpression =
    BinaryExpression.and(
        new SingleSeriesExpression(path, valueFilter),
        new GlobalTimeExpression(timeFilter));
QueryExpression expression =
    QueryExpression
        .create()
        .addSelectedPath(path)
        .setExpression(expression);
QueryDataSet res = reader.query(expression);
RowRecord row = res.next();
```

Code example above demonstrate a naive usage to access data points of a certain series with time and value filters. Line 3-5 exemplify that value filter is applied to a specific series while time filter works on all series specified in line 9-13. Line 15 places the initial data points that meet the filter criteria, each from a selected series, into a `RowRecord`, thereby forming a tabular structure, smoothly integrating with various applications that utilize table formats.

Since TsFile can be stored in distributed file systems like HDFS, it can be split into fixed-size blocks distributed across cluster servers. `TsFileReader` provides an interface for querying data at specific offset range, facilitating data retrieval only on the local server to minimize network overhead in big data analysis.

5.3 TsFile Compaction

`TsFileResource` and `ICompactionPerformer` are the two key components for compaction. The fundamental usage of each is outlined below. `TsFileResource` acts as a summary of `TsFile`, offering statistics on the devices contained within the file, such as timestamps of both the first and the last data point. `ICompactionPerformer`

can be implemented in various approaches but consistently requires both source and target files.

```
TsFileResource rsc1 = new TsFileResource(tsFile1);
TsFileResource rsc2 = new TsFileResource(tsFile2);
TsFileResource rsc3 = new TsFileResource(newFile);

ICompactionPerformer performer =
    new FastCompactionPerformer();
performer.setSourcesFiles(rsc1, rsc2);
performer.setTargetFiles(rsc3);
performer.perform();
```

6 PERFORMANCE EVALUATION

We compare TsFile with other widely-used open file formats, namely Parquet and Arrow. Furthermore, we also compare Apache IoTDB [36], which employs TsFile as its underlying storage format, with InfluxDB [18] and other top performers in time series database track. Among these systems, InfluxDB-IOx [19, 20] utilizes Parquet as its underlying storage. When storing time series data in Parquet, we will discuss the alternatives of schema for fairer comparison.

6.1 Experimental Setup

6.1.1 Hardware. For evaluation in Section 6.2 , we perform the experiments on an 8-core Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz machine with 32GB memory, 1T SSD, and 64-bit Windows 10.

For systematic evaluation in Section 6.3.1, we conduct the experiments on a machine with 20-core Intel(R) Core(TM) i7-12700 CPU, 16 GB memory and 512GB SSD, running 64-bit Ubuntu 22.04.1 SMP. For Section 6.3.2, we conduct the evaluations on a Raspberry Pi 4 Model B with 8GB RAM, which is approximate to industrial end devices in typical IoT scenarios.

6.1.2 DataSets. We employ three public real-life datasets, one time series benchmark, and two datasets from our industrial partners as listed in Table 1.

The Reference Energy Disaggregation Data Set (REDD) [23] contains detailed electricity usage data collected from various households, including both high-frequency appliance-level power usage and low-frequency whole-house power consumption. The dataset used in this paper contains data from 6 buildings, each with approximately 20 meters. Every meter is considered as a device generating only 1 time series and is identified by the combination of building and meter number.

GeoLife [43] and *TDrive* [40, 41] are GPS trajectory datasets consisted of coordinates recorded during a wide array of activities like walking, running, cycling and driving. Every object tracked in these datasets are deemed to be a device equipped with sensors measuring its coordinate, which constitutes time series data.

Time Series Benchmark Suite(TSBS) [34] is a collection of programs widely used to generate tailored dataset for benchmark. This paper employ the IoT case in the suite, where data are pertained to a set of trucks, including their coordinates, velocity and other status. TSBS interleaves data points from different devices, but the data points for each individual device are sequential in terms of the timestamp. As Section 3 illustrates, performance in common file formats like Parquet declines when data points are not sorted by device ID, whereas TsFile maintains unaffected performance. For the sake of fairness, we reorganize all data points by its device ID,

Table 1: Dataset Profile

DataSet	Points	Series	Devices
REDD	56M	115	115
GeoLife	72M	543	181
TDrive	18M	17778	8889
TSBS	496M	16000	4000
ZY	376M	17154	186
CCS	161M	2750	1108

i.e., data points from the same device are stored contiguously and ordered by timestamp, before writing to the file.

The *ZY* dataset, provided by our industrial partner, consists of data points collected by sensors on rock drilling machines. This dataset is more sparse as these data are only available if related machines are working. Furthermore, the quantity of sensors linked to different devices differs significantly. Some devices have fewer than three sensors, while others have over a hundred due to the varying complexity of their tasks.

The *CCS* dataset is provided by our industrial partner as well. The data are collected from shipbuilding components, as mentioned earlier. In comparison to other datasets, some time series in this dataset are collected at high frequency, such as data points from vibration measurements.

6.2 File Evaluation

We evaluate TsFile with Parquet and Arrow, which are representative open file formats in these days, regarding space cost, write speed, and query latency across various datasets. While Arrow was initially designed for in-memory usage, it indeed has an interprocess communication format, which is also known as Feather [25]. When we write data into disks, we actually write Feather files; while we read data from Feather, we actually read Arrow data in memory. In the following experiments, for simplicity, we will refer to both Arrow and Feather collectively as Arrow. Although there are other open file formats such as ORC [16] or RCFile [15], their architecture is similar to that of Parquet and has been thoroughly analyzed in previous research [24, 36, 42].

In contrast to the flexible and IoT-native data model in TsFile, Parquet and Arrow require the data schema to be defined based on data characteristics before writing data to the file. As they employ a tabular schema, if the device ID has multiple fields, there are primarily two alternatives for schema definition. The first approach stores each field from the device ID in an individual field. InfluxDB-IOx, which utilizes Parquet as its underlying storage, adopts this approach [19, 20]. The second approach stores the entire device ID in a single column, resulting in a simpler layout but compromising the atomicity of these fields. For instance, Device ID in TSBS includes three parts: *name*, *fleet*, *driver*. The first definition stores them in different fields while the second stores them in a single one, as the following snippet illustrated. On the other hand, the device ID in TsFile is represented as segmented string such similar to "<name>.<fleet>.<driver>".

```
// schema of Parquet
message TSBS{
```

```
    required binary name;
    required binary fleet;
    required binary driver;
    required int64 timestamp;
    optional double lat;
    optional double lon;
    optional double ele;
    optional double vel;
}
// schema of Parquet-AS
message TSBS{
    required binary deviceID;
    required int64 timestamp;
    optional double lat;
    optional double lon;
    optional double ele;
    optional double vel;
}
```

We implement both strategies on Parquet, referred to as Parquet and Parquet-AS (for Alternated-Schema) respectively in the following sections, while Arrow is only implemented with the second one for simplicity.

TsFile achieves optimal performance by writing `Tablet` in batch. However, as Parquet lacks such batch interface [1] currently, we compare only the write times of internal processes during data ingestion, mitigating effect of interface differences. Furthermore, we have added the construction time of `Tablet` in to the comparison of write time. In terms of space cost, Parquet automatically select the most suitable encoding scheme with its auto-encoding feature, and TsFile consistently uses *GORILLA* [29] encoding for comparability. To mitigate discrepancies in the implementation of compression algorithms, we employ no compression for all file formats.

Through all following evaluations, each datasets are written into one file per format. Parquet takes the default blocks size threshold 128 MB, which equals the flush threshold within TsFile. Arrow takes a default batch size 64K rows. Both Parquet and Arrow enable dictionary encoding for text fields, i.e., component fields of deviceID. unless specifically mentioned, all other configurations are consistent with the default setting.

6.2.1 *Space Cost.*

Space cost is crucial because time series data is either stored on resource-constrained endpoints or edge devices, or is stored with a high volume in cluster servers. Figure 8 (a) reports that with various datasets, TsFile consistently occupies less or approximate space compared to Parquet, and both significantly outperform serialized Arrow format [24] in term of space cost. The figure reveals that the space cost is little affected by the alternative schema definition due to the dictionary encoding. While Parquet utilizes dictionary encoding to mitigate the redundancy of device IDs, this ultimately increases space cost in data area. It should be noted that TsFile consumes slightly more space than Parquet with the TDrive dataset, attributed to the numerous devices each hosting brief series data, leading to an increased number of Chunk Groups in TsFile.

Figure 8 (b) shows TsFile can have larger Index Area than Parquet under specific dataset conditions. However, in these datasets, index area is several orders of magnitude smaller than the data area, thereby its effect on the overall size remains minimal. In Parquet, the size of the index area depends on the number of pages, correlating directly with the overall number of data points. On the other hand, the index area size in TsFile is primarily proportional to the number of devices, as there is no index entry for pages.

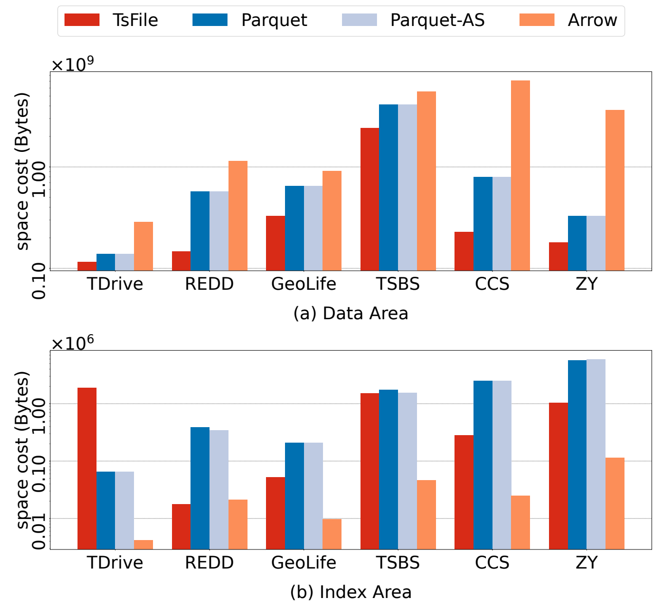

Figure 8: Space Cost

6.2.2 *Write Latency.*

Figure 9 presents the write latency across various datasets for the data area and index area, indicating the rate of data points written per second. TsFile outperforms Parquet in the data area by eliminating the redundant storage of descriptions like device IDs. Parquet introduces extra overhead for its auto-encoding feature and column-striping algorithms, adding complexity even if all datasets employed involve no nested types. Arrow is significantly faster than its competitors, as its in-memory and serialized layouts are similar.

Given that Parquet provides no batch writing interface, whereas TsFile employs a batch style ingestion, we have also included the construction time of `Tablets` in TsFile in the figure. Even with this additional time overhead, TsFile still performs out Parquet. It is worth noting that to imitate the writing method of Parquet, the `tablet` construction approach in the evaluation is far from the optimal, indicating that TsFile could exhibit even better performance in practice.

In the index area, Parquet employs Apache Thrift [4] for serialization, trading off higher latency for reduced space occupation. Like with space cost, latency in the index area is significantly lower than in the data area, minimally affecting the overall performance.

6.2.3 *Raw Query.*

Raw query denotes accessing specific series or multiple series within a device without any filters. To ensure equivalence of queries across different formats for each dataset, we performed uniform random sampling from the dataset prior to querying, using these samples as query conditions. The query commands for each file were defined based on these pre-generated conditions.

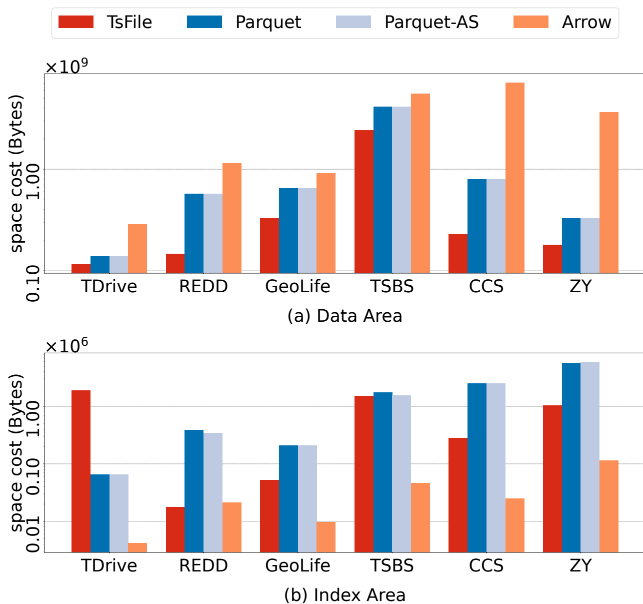

Figure 9: Write Latency

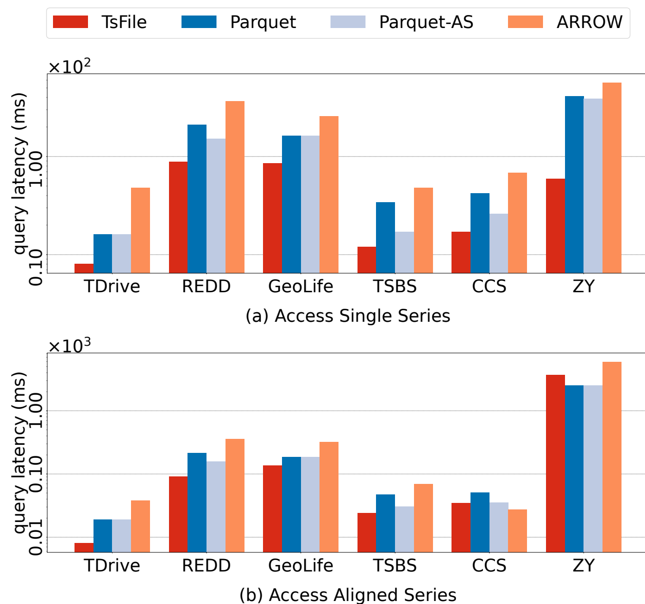

Figure 10: Raw Series Access

We employed a cold query methodology. Only one type of query will be performed once a file is opened. After the file is closed and reopened, queries of other types will then be conducted.

The query latency is calculated between the time when the query is issued and when all target data is received. This process includes the time taken to read the relevant data blocks from disk into memory. All data is sorted first by timestamp and then by device ID, ensuring that data from the same device is stored contiguously and ordered by timestamp.

As shown in Figure 10, TsFile maintains consistently low latency in pinpointing series, whereas Parquet, using page indexes for locating specific series, shows increased latency as Data Area extends, despite data being sorted by device ID. Even under the sorting aforementioned, access latency in Arrow is significantly higher than other formats due to two factors. Firstly, Arrow must deserialize the entire block [24, 42] for each read operation before it can filter the data, thus decoding an excessive amount of unnecessary data; secondly, Arrow does not establish an index at the file level, necessitating sequential scanning of blocks for access. While other modules have been developed for facilitating access to Arrow data structures, they primarily aim to improve expressiveness [2] or universality [3] without modifying the file layout, and therefore, are beyond the scope of this paper.

6.2.4 Filtered query. Filtering time series data based on timestamps or values reflects the comprehensive performance of the index mechanism. For either type of filtering, only the filtered series will be retrieved, regardless of other series under the same device. As shown in Figure 11, TsFile consistently outperforms across all datasets and filters, with timestamp filtering always exhibiting lower latency. This is because timestamps in each series are monotonically increasing, allowing TsFile to leverage this feature to optimize querying process. In contrast, Parquet and Arrow cannot

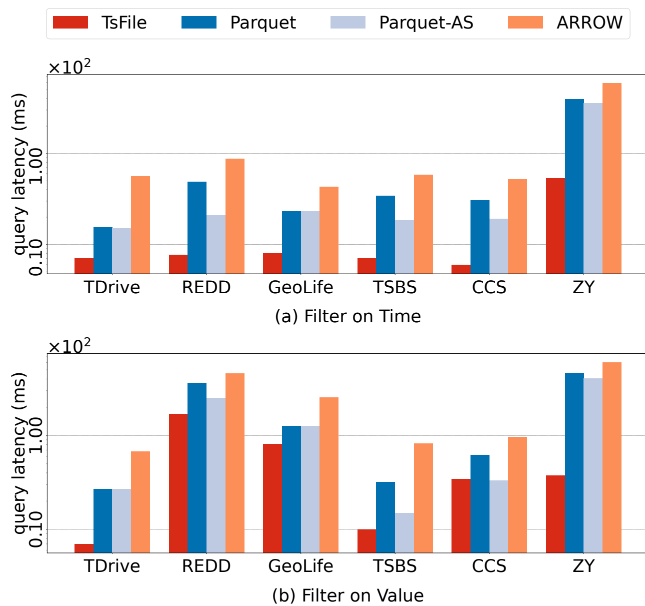

Figure 11: Filtered Series Access

ascertain timestamps are monotonically incrementing, resulting in higher latency.

The effectiveness demonstrated in overall performance comparison derives from a more detailed design. This section evaluates the aforementioned designs using more specific metrics.

6.2.5 Bulk Compaction. Figure 12 illustrates that the compaction method employed for TsFile, termed Fast-Compaction, significantly

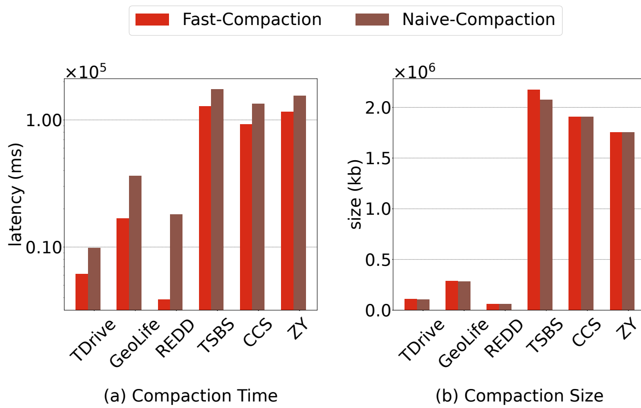

Figure 12: Compact Effect

outperforms Naive-Compaction, which requires reading all chunks to verify the interleaving of devices to maintain their order.

Fast-Compaction sketches all files specified for compaction at begin, executing a multi-way merge for each device appearing in more than one file. When addressing devices from multiple files, it scrutinizes all chunks within the chunk group for any overlap in time ranges. Chunks without time range overlap are directly transferred to the target file by the order of time, without deserialization and decoding. This approach may also benefits from zero-copy technology. In contrast, Naive-Compaction, a method requisite for common open file formats like Parquet that necessitate maintaining device order for efficient access, mandates decoding for each chunk, thus incurring additional latency.

Files resulting from Fast-Compaction are shown to be equally efficient or nearly so, regarding file size and performance across various queries. Details on query latency and other metrics are omitted due to spatial constraints.

6.3 System Evaluation

6.3.1 Overall Performance. Another important role of TsFile is serving as the storage format for TSDBMS. We compared the write throughput and access latency of Apache IoTDB and top performers in the ranking of TSDBMS according to the benchANT [8].

It is worth noting that among these systems, only Apache IoTDB and InfluxDB employs open file format as underlying storage. VictoriaMetrics [35] writes data into 'part' directories on disk, each containing separate time and value files, along with several other metadata and index files. QuestDB [30] stores data of each table into several appending files, each relating to a column. Every column file is accompanied by an extra index file. TimescaleDB [33] employs PostgreSQL for underlying storage so time series data are stored in related files [14]. These file formats do not provide integrated interfaces for direct usage and, thus, typically cannot be utilized independently of their associated database systems. These file formats are challenging to employ for IoT devices with insufficient resources to run a complete database system.

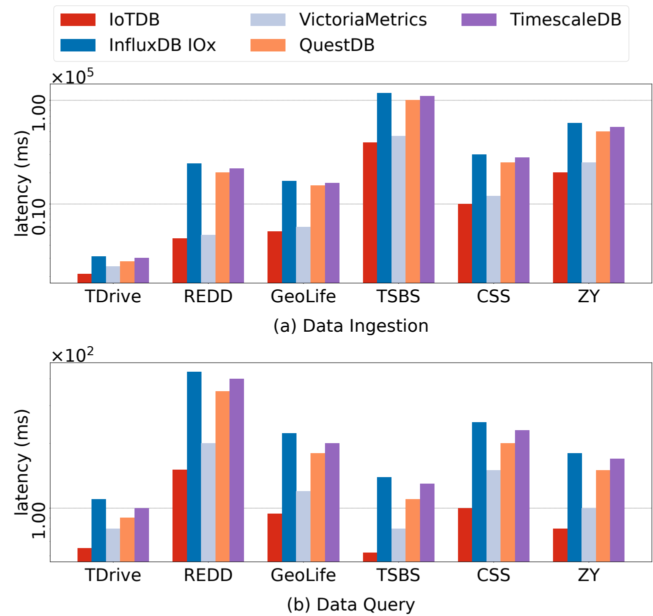

Figure 13: System Comparison

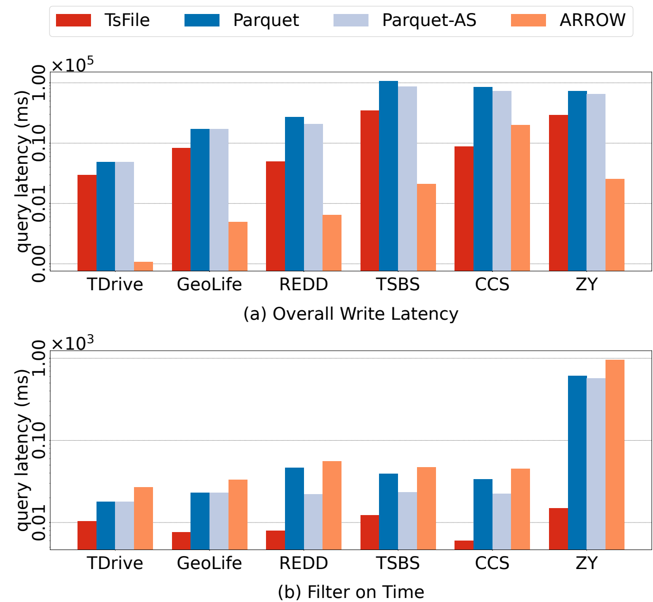

Figure 14: Industrial End Device Setup

Figure 13 reports that Apache IoTDB consistently outperforms other systems across various datasets. Given the earlier comparisons between TsFile and Parquet under various loads and functionalities and the report from benchANT [8], result of this comparison is hardly surprising.

6.3.2 End Device. We conduct the performance on typical industrial devices. Since TsFile provided platform-independent interfaces,

the architectural or system differences between industrial computers and ordinary personal computers have little impact on performance. Figure 14 (a) reports overall write latency. Figure 14 (b) illustrates query latency with filters on time, a typical access pattern in IoT scenario. To cope with the limited memory resource on the end device, we reduced the block size in Parquet and flush threshold in TsFile to 64M. For Arrow, we reduce the batch size to 32K rows. However, these discrepancies barely impact the performance.

7 RELATED WORK

7.1 Data Model for IoT Time Series

The data model of time series represents the associations between series and the semantics of time series data. As the data model determines the physical layout in some extent, therefore designing an efficient file format for IoT time series requires an in-depth study of its data model.

The hierarchical structure is widely applied among conventional time series management. Since the 1980s, the industrial sector has relied on data historians [6] to manage time series data from sensors within industrial systems, making time series modeling a pivotal concern. The International Society of Automation introduced standards like Batch Control ISA-88 [21] and Enterprise-Control System Integration ISA-95 [22], featuring a hierarchical *Physical Model* to delineate the structure among sensors and devices. In 2009, major industry players like IBM and Siemens, alongside other enterprises and institutions, participated in the European Union's Seventh Framework Internet of Things Architecture Project (IoT-A) [7]. This initiative aimed to standardize fundamental IoT concepts, offering a domain model that sketches out entities such as devices and sensors. OSIsoft's renowned PI Asset Framework [5] and General Electric's Predix Asset Model [13] both employ hierarchical structures to organize industrial system entities. The PI Asset Framework utilizes interconnected *Elements* to represent entities, with each element having multiple children but only one parent. Similarly, the Predix *Asset Model* categorizes entities into five hierarchical levels, structuring the complex relationships within industrial systems. For TSDBMS today, a unified data model has yet to be established, the concept of device is implemented in diverse approaches. Such as InfluxDB and Prometheus employ labels for series specification, QuestDB and QuasarDB leverage symbol-type columns in tables for data invariant of time. TsFile, as the underlying storage of IoTDB, organizes device IDs in a hierarchy, thereby adhering to IoT standards and capturing device specifications as proposed by TSDBMS.

7.2 File Format for Big Data

In big data applications, various file formats have been proposed and applied widely for their high performance and concise format design, such as Parquet [28], ORC [16] and RCFile [15]. However, these formats are not designed for time series data in IoT scenario, resulting in performance fall back as show in Section 6.

Specifically, Apache Parquet [28] is featured with column-striping and record assembly algorithms [26], which is designed for the nested data such as XML or JSON documents. However these complex types rarely occurs in time series data as sensor readings are mostly numerical, resulting in unnecessary overhead to store and decode corresponding structures. For instance, although InfluxDB IOx [18, 20] stores time series data into Parquet, it only provides primitive data types to this day. Storing time series data into common file format, like Parquet, ORC or RCFile, incurs another serious drawback, even though descriptive information about time series, like device ID, are static through the life cycle, they are handled as ordinary columns thus introducing unnecessary storage consumption even with dictionary encoding. Let alone that Parquet would withdraw its dictionary encoding once a dictionary grows too large, storing identical time series description repeatedly. In contrast, TsFile stores them in tree-structured index area, reducing space cost while proving fast access.

Apache ORC [16] is a successor of RCFile [15], employing a tabular data model and supporting nested data types as Parquet. Instead of using a specific threshold to determine encoding strategy as Parquet, ORC uses the ratio of distinct values to predetermine whether to apply dictionary encoding [24]. When the number of time series in one file increases, ORC would finally withdraw dictionary encoding the drawback occurs. Contrast to Parquet construct its page-level index after all row groups, ORC maintains index data in the front of each stripes, leading to a higher price to filter with specific condition. ORC investigates the affect of row reordering, table partitioning and data packing on performance in deep dive [17], however these scenario-independent designs are not taking advantage of the features in IoT scenarios. Since the data placement method in ORC is quite similar to that in Parquet, thereby this paper do not take it in evaluations as above.

8 CONCLUSION

This paper presents Apache TsFile, which, to our knowledge, stands as the first open file format specifically designed for time series data in IoT applications. TsFile leverages characteristics inherent to each series, by employing distinct encoding and compression scheme upon individual series. As IoT time series is invariably associated with devices, TsFile groups series from the same device and stores points from same series contiguously, optimizing compactness. Specification of series is crucial for accessing and invariant of time, these elements are utilized for indexes in TsFile. This approach minimizes storage redundancy and ensures rapid data retrieval. Having been recognized as a top-level Apache project, TsFile has seen extensively applied in IoT contexts and demonstrates superior performance as demonstrated by our evaluation. With the rapid emergence of time series data and intelligent devices, we believe that TsFile holds significant promise for enhancing future time series data applications and beyond.

ACKNOWLEDGMENTS

This work is supported in part by the National Natural Science Foundation of China (62021002, 62072265, 62232005, 92267203), the National Key Research and Development Plan (2021YFB3300500), the State Grid Science and Technology Project (5700-202435261A-1-1-ZN), the Chongqing Technical Innovation and Application Development Key Project (CSTB2023TIAD-STX0034), Beijing Key Laboratory of Industrial Big Data System and Application. Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

REFERENCES

[1] Apache. 2024. https://github.com/apache/parquet-java/.

[2] Apache. 2024. https://arrow.apache.org/docs/cpp/gandiva.html.

[3] Apache. 2024. https://arrow.apache.org/docs/java/dataset.html.

[4] Apache Thrift. 2024. https://thrift.apache.org/.

[5] Aveva. 2024. Asset Framework and PI System Explorer. https://docs.aveva.com/bundle/pi-server-af-pse-f/page/1031642.html. Accessed: 2024-02-16.

[6] DC Barr. 1994. The use of a data historian to extend plant life. (1994).

[7] Martin Bauer, Nicola Bui, Jourik De Loof, Carsten Magerkurth, Andreas Nettsträter, Julinda Stefa, and Joachim W Walewski. 2013. IoT reference model. *Enabling Things to Talk: Designing IoT solutions with the IoT architectural reference model* (2013), 113–162.

[8] benchANT. 2024. https://benchant.com/ranking/database-ranking.

[9] Jeffrey Dean and Sanjay Ghemawat. 2010. MapReduce: a flexible data processing tool. *Commun. ACM* 53, 1 (2010), 72–77.

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

[11] Chenguang Fang, Shaoxu Song, and Yinan Mei. 2022. On repairing timestamps for regular interval time series. *Proceedings of the VLDB Endowment* 15, 9 (2022), 1848–1860.

[12] Chenguang Fang, Shaoxu Song, Yinan Mei, Ye Yuan, and Jianmin Wang. 2022. On aligning tuples for regression. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 336–346.

[13] GE Digital. 2022. About Asset Model. https://www.ge.com/digital/documentation/predix-essentials/latest/c_apm_asset_about_asset_model_1.html. Accessed: 2024-02-16.

[14] The PostgreSQL Global Development Group. 2024. https://www.postgresql.org/docs/16/storage-file-layout.html/.

[15] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. 2011. RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 1199–1208.

[16] Yin Huai, Ashutosh Chauhan, Alan Gates, Gunther Hagleitner, Eric N Hanson, Owen O'Malley, Jitendra Pandey, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2014. Major technical advancements in apache hive. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1235–1246.

[17] Yin Huai, Siyuan Ma, Rubao Lee, Owen O'Malley, and Xiaodong Zhang. 2013. Understanding insights into the basic structure and essential issues of table placement methods in clusters. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1750–1761.

[18] InfluxData. 2024. https://www.influxdata.com/time-series-platform/influxdb/.

[19] InfluxData. 2024. https://www.influxdata.com/blog/understanding-influxdb-iox-commitment-open-source/.

[20] InfluxData. 2024. https://github.com/influxdata/influxdb/tree/3c5e5bf241dcc2c0e13554c5286577ad6066bfec/parquet_file/.

[21] International Society of Automation (ISA) 2010. *Batch Control Part 1: Models and Terminology*. International Society of Automation (ISA). Accessed online at https://www.isa.org/products/ansi-isa-88-00-01-2010-batch-control-part-1-models.

[22] International Society of Automation (ISA) 2010. *Enterprise-Control System Integration*. International Society of Automation (ISA). Accessed online at https://www.isa.org/products/ansi-isa-95-00-01-2010-iec-62264-1-mod-enterprise.

[23] J Zico Kolter and Matthew J Johnson. 2011. REDD: A public data set for energy disaggregation research. In *Workshop on Data Mining Applications in Sustainability (SIGKDD), San Diego, CA*, Vol. 25. 59–62.

[24] Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. 2023. A deep dive into common open formats for analytical dbmss. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3044–3056.

[25] Wes McKinney. 2024. https://github.com/wesm/feather/.

[26] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 330–339.

[27] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385. https://doi.org/10.1007/S002360050048

[28] Apache Parquet. 2024. https://parquet.apache.org/.

[29] Tuomas Pelkonen, Scott Franklin, Paul Cavallaro, Qi Huang, Justin Meza, Justin Teller, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, In-Memory Time Series Database. *Proc. VLDB Endow.* 8, 12 (2015), 1816–1827. https://doi.org/10.14778/2824032.2824078

[30] QuestDB. 2024. https://questdb.io/docs/concept/storage-model/.

[31] SQLite. 2024. https://www.sqlite.org/.

[32] Yu Sun, Shaoxu Song, Chen Wang, and Jianmin Wang. 2020. Swapping repair for misplaced attribute values. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 721–732.

[33] TimescaleDB. 2023. https://www.timescale.com/.

[34] TimescaleDB. 2023. Time Series Benchmark Suite (TSBS). https://github.com/timescale/tsbs.

[35] VictoriaMetrics. 2024. https://docs.victoriametrics.com/#storage/.

[36] 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. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[37] Haoyu Wang and Shaoxu Song. 2022. Frequency domain data encoding in apache IoTDB. *Proceedings of the VLDB Endowment* 16, 2 (2022), 282–290.

[38] Zhenhua Wang, Huikun Pei, Xiaomeng Zhang, Chenghao Wang, Xi Chen, and Te Zhou. 2021. Application of KKS Coding and QR Code Technology in Transmission Asset Management. In *2021 IEEE 2nd China International Youth Conference on Electrical Engineering (CIYCEE)*. IEEE, 1–6.

[39] Jinzhao Xiao, Yuxiang Huang, Changyu Hu, Shaoxu Song, Xiangdong Huang, and Jianmin Wang. 2022. Time Series Data Encoding for Efficient Storage: A Comparative Analysis in Apache IoTDB. *Proc. VLDB Endow.* 15, 10 (2022), 2148–2160. https://doi.org/10.14778/3547305.3547319

[40] Jing Yuan, Yu Zheng, Xing Xie, and Guangzhong Sun. 2011. Driving with knowledge from the physical world. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 21-24, 2011*, Chid Apté, Joydeep Ghosh, and Padhraic Smyth (Eds.). ACM, 316–324. https://doi.org/10.1145/2020408.2020462

[41] Jing Yuan, Yu Zheng, Chengyang Zhang, Wenlei Xie, Xing Xie, Guangzhong Sun, and Yan Huang. 2010. T-drive: driving directions based on taxi trajectories. In *18th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems, ACM-GIS 2010, November 3-5, 2010, San Jose, CA, USA, Proceedings*, Divyakant Agrawal, Pusheng Zhang, Amr El Abbadi, and Mohamed F. Mokbel (Eds.). ACM, 99–108. https://doi.org/10.1145/1869790.1869807

[42] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2023. An Empirical Evaluation of Columnar Storage Formats. *Proc. VLDB Endow.* 17, 2 (2023), 148–161. https://www.vldb.org/pvldb/vol17/p148-zeng.pdf

[43] Yu Zheng, Xing Xie, and Wei-Ying Ma. 2010. GeoLife: A Collaborative Social Networking Service among User, Location and Trajectory. *IEEE Data Eng. Bull.* 33, 2 (2010), 32–39. http://sites.computer.org/debull/A10june/geolife.pdf