


An Efficient NoSQL-Based Storage Schema for Large-Scale Time Series Data

Ruizhe Ma, University of Massachusetts, Lowell, USA

Weiwei Zhou, Nanjing University of Aeronautics and Astronautics, China

Zongmin Ma, Nanjing University of Aeronautics and Astronautics, China*

 <https://orcid.org/0000-0001-7780-6473>

ABSTRACT

In IoT (internet of things), most data from the connected devices change with time and have sampling intervals, which are called time-series data. It is challenging to design a time series storage model that can write massive time-series data in a short time and can query and analyze the persistent time-series data for a long time. This paper constructs the RHTSDB (Redis-HBase Time Series Database) storage model based on Redis and HBase. RHTSDB uses the memory database Redis (Remote Dictionary Server) to cache massive time-series data, providing efficient data storage and query functions. HBase is used in RHTSDB for long-term storage of time-series data to realize their persistence. The paper designs a cold and hot separation mechanism for time-series data, where the infrequently accessed cold data are stored in HBase, and the frequently accessed and latest data are stored in Redis. Experiments verify that RHTSDB has apparent advantages over Apache IoTDB and HBase in data intake and query efficiency.

KEYWORDS

HBase, Query, Redis, Storage, Time-Series Data

INTRODUCTION

With the development of the Internet of Things (IoT) (Eom & Lee, 2017), the amount of time-series data has shown explosive growth. Time-series data refers to a sequence of data points collected at fixed time intervals (Lee & Chung, 2014). Each data point is associated with a timestamp that indicates the generation time of the corresponding data. Typically, the data collected by a sensor in a particular period can be expressed as a time series $[(t_p, v_1), (t_2, v_2), \dots, (t_n, v_n)]$, where v_i refers to the value collected at t_i time (Di Martino *et al.*, 2019). Of course, complete time-series data can include the collection time and collection value as well as the source description information of the current collection value. For example, we need to include some measurement data information, such as the names of collection subject and collection index. Comprehensive use cases in the real world have generated a large amount of measurement data from millions or billions of different sources. Slack

DOI: 10.4018/JDM.339915

*Corresponding Author

This article published as an Open Access article distributed under the terms of the Creative Commons Attribution License (<http://creativecommons.org/licenses/by/4.0/>) which permits unrestricted use, distribution, and production in any medium, provided the author of the original work and original publication source are properly credited.

collects measurement data from 4 billion unique sources at 12 million samples per second daily, for example, generating up to 12 TB of compressed data daily. It is essential to manage and process a large amount of time-series data efficiently. Unfortunately, many off-the-shelf systems cannot scale to support these workloads, which leads to the random Patchwork and vulnerability of customized solutions (Solleza, Crotty, Karumuri, Tatbul & Zdonik, 2022). For this reason, diverse time series databases are proposed to ensure efficient ingestion performance and save storage space as much as possible. Given that time-series data in applications are generally massive and redundant data containing source description information in time-series data are enormous, efficient storage and query of massive time-series data is challenging.

We identify two major categories of time series databases: which are respectively called *native time series databases* and *common time series databases* in this paper. The native time series databases are the storage systems that are developed especially for time-series data management according to their structural and usage characteristics, such as InfluxDB¹, FluteDB (Li *et al.*, 2018), and Apache IoTDB (Wang *et al.*, 2020 & 2023). This category of time series databases can efficiently reduce the overhead of storage space and the query delay. However, for time-series data management and processing, many other functions and operations are essential in time series databases, such as flexible aggregation, data retention, multidimensional range query, among others. While the native time series databases cannot provide full support to time-series data analysis well, mature database systems are good at dealing with relationships between data and support many unnecessary operations and guarantees for time series, increasing inefficiency and unnecessary complexity (Shafer, Sambasivan, Rowe, & Ganger, 2013). The common time series databases are the storage systems that directly apply the common databases for storing and processing time-series data. Depending on what types of databases are applied, we further identify two categories of common time series databases. The first one uses relational databases as the back end of common time series databases (e.g., (Rhea *et al.*, 2017)). In recent years, NoSQL (Not only SQL) databases have attracted increasing attention from both academia and industry (Hu & Dessloch, 2015), which offer flexible data representation models and horizontal hardware scalability so that Big Data can be processed in real time (Bajaj & Bick, 2020). The second category of common time series databases uses NoSQL databases for processing time-series data (Di Martino *et al.*, 2019).

NoSQL databases contain four major types of database models: *key-value stores*, *column-family stores*, *document stores*, and *graph stores* (Grolinger *et al.*, 2013; Van Erven *et al.*, 2019). Different types of NoSQL databases, say Redis² (a key-value store), HBase³ (a column-family store), Cassandra⁴ (a column-family store), MongoDB⁵ (a document store), Couchbase⁶ (a document store), OrientDB⁷ (a graph store), have very different performances (Matallah, Belalem & Bouamrane, 2020). Among these NoSQL databases, for example, Redis (remote dictionary server) is a high-performance memory-based NoSQL database, which has excellent data writing performance and supports data persistent storage and replication of master-slave nodes (Zhou, Lu, Zhang & Qi, 2020); HBase, an open-source, distributed and versioned NoSQL database, uses Hadoop distributed file system (HDFS) to provide distributed file storage services, which has the characteristics of high availability, robust scalability and the ability to store massive data. With one of NoSQL databases as back end, some storage models for massive time-series data have been developed, for example, ModelarDB+ (Jensen, Pedersen & Thomsen, 2021) based on Cassandra, NagareDB (Calatrava, Fontal, Cucchiatti & Diví-Cuesta, 2021) built on top of MongoDB, OpenTSDB⁸ based on HBase, and KairosDB⁹ based on Cassandra.

Selecting a specific NoSQL database for massive time-series data management is more flexible and cost-saving, where the performance of the storage model is determined by the performance of the used database models (Rinaldi *et al.*, 2019). It has been demonstrated that different NoSQL databases have very different performances (Matallah, Belalem & Bouamrane, 2020). For ingestion of large-scale time-series data into the target time series database, for example, the storage consumption of HBase slowly increases along with a significant increase in data size (e.g., a 50% increase in data size requires about a 20% increase in storage consumption). However, the storage consumption of

Redis rapidly increases along with a significant increase in data size (e.g., a 50% increase in data size requires about a 200% increase in storage consumption). In addition, the time of data ingestion in HBase dramatically increases along with a significant increase in data size (a 50% increase in data size requires a more than 200% increase in data ingestion time), and the time of data ingestion in Redis does not significantly increase along with a significant increase of data size (e.g., a 50% increase in data size requires a less than 200% increase in storage consumption). To take full advantage of the superior performance of different NoSQL databases simultaneously, joint use of multiple NoSQL databases has emerged for efficient time-series data storage. Cassandra and MongoDB, for example, are used for discrete time-series data modeling (Ramesh, Sinha & Singh, 2016), where the former has a sequential data storage mechanism and the latter has a flexible schema and rich query language. In addition, several in-memory time series databases such as Gorilla (Pelkonen *et al.*, 2015) and Monarch (Adams *et al.*, 2020) are applied to significantly improve the reading performance of time-series data by saving all the latest data in memory. In practice, the in-memory time series databases are generally built atop the common databases (e.g., Gorilla on HBase and Monarch on relational databases). The idea of jointly using in-memory databases and HBase has been applied in, for example, image data management (Zhou, Lu, Zhang & Qi, 2020) and financial data (Li, Guo & Guo, 2019), where Redis acts as data cache model.

Based on Redis and HBase, this paper proposes a storage schema RHTSDB (Redis HBase Time Series Database) for large-scale time-series data storage. In RHTSDB, to improve the data intake rate, Redis is used as an in-memory time series database, which can eliminate the cache through data expiration time and elimination strategies like FIFO (First in First Out), LRU (Least Recently Used), LFU (least frequently used), etc. Note that although Redis also supports disk persistence of time-series data, it does not provide a complete storage scheme, and its data intake and query performance decline significantly when the size of time-series data increases. For this reason, RHTSDB adopts HBase rather than Redis to store long-term time-series data. HBase, an open-source project, can customize requirements to a certain extent. In addition, HBase is a distributed storage system that has good expansion characteristics and low requirements for server performance. Overall, in RHTSDB, HBase stores infrequently accessed data (called cold data) for solid data scalability, while Redis stores frequently accessed and the latest time-series data (called hot data) for excellent data intake performance. To implement the storage schema of RHTSDB, we need to solve two crucial problems: how to separate hot and cold time-series data and how to transform time-series data between Redis and HBase. We design a cold and hot time-series data separation mechanism for the first issue and propose a cache elimination strategy. As to the second issue, we consider both Redis keys and HBase row keys, designing the HBase and Redis middle key. The row key design of HBase is critical and is related to data query performance (Li, Guo & Guo, 2019). Continuous row keys and heavy data operations may lead to a single regional hotspot and even regional server unavailability. In addition, we supply RHTSDB with some standard functions, supporting time-series data insertion, single value query, range query, multidimensional query, and time-series data update. We verify our time series storage schema RHTSDB with experiments and show its advantages over Apache IoTDB and HBase in data intake and query efficiency.

The rest of this paper is organized as follows. The section on related work mainly discusses the existing work of time-series data storage models. The Redis-HBase time-series database schema section proposes our storage schema RHTSDB for large-scale time-series data. The experiment section evaluates and analyzes the performance of RHTSDB. Conclusion summarizes our work in this paper and presents the work to be carried out in the future.

RELATED WORK

For massive time-series data analysis and processing, it is essential but challenging to store and query time-series data efficiently. Time-series data are a special type of data, and an efficient way is to use

native time series databases, which are specially developed for time-series data management according to the structural and usage characteristics of time-series data. Several native time series databases have been developed and used. InfluxDB written in the Go language without external dependence, for example, implements the TSM (Time-Structured Merge) Tree and optimizes the LSM (Log-Structured Merge) Tree for time-series data. In InfluxDB, the data points stored in the memory data structure are collectively written to disk. By absorbing the delta coding scheme proposed by Gorilla (Pelkonen *et al.*, 2015), InfluxDB can save storage space by obtaining efficient data compression performance. FluteDB (Li *et al.*, 2018) designs a TTSM (Triggered Time Series Merge) Tree to optimize key-related operations and physical storage in memory at the expense of some acceptable data accuracy and consistency. Its storage scheme has strong pertinence and low external dependence. Byteseries (Shi *et al.*, 2020) effectively solves the problem of data redundancy in massive time-series data. It divides the memory into active buffer and static buffer, where the active buffer is mainly responsible for efficient data intake with a dynamic data structure, and the static buffer is mainly used for data storage. In addition, Byteseries proposes the compressed inverted index algorithm, which can ensure efficient ingestion and multidimensional query performance. Aiming at high throughput, low latency, and advanced timing analysis performance indicators of time-series data, Apache IoTDB (Wang *et al.*, 2020) adopts the LSM Tree mechanism and provides high-performance data reading/writing and rich query capabilities in the cloud. Apache IoTDB customizes an efficient directory organization structure for IoT scenarios. In particular, Apache IoTDB can be seamlessly connected with Apache Hadoop, Spark, Flink, and other extensive data systems. At the edge, Apache IoTDB provides the capability of lightweight TsFile management. The data on end is written to the local TsFile, and Apache IoTDB provides specific basic query capabilities. At the same time, Apache IoTDB supports the synchronization of TsFile data to the cloud. The native time series databases usually adopt some storage optimization algorithms themselves and make the storage of some time-series data with superior performance. However, in addition to complex design, long development cycle, and high cost, they may be problematic in their adaptability aspect: for example, they fail to provide full support to time-series data management (e.g., flexible aggregation and multidimensional range query); a native time series database with superior performance in storing some time-series data may not do well for other time-series data.

Database systems have been widely used in data management of diverse applications, where various types of data are stored in databases and then processed. Of course, using common databases in time-series data is beneficial to time-series data analysis (Shafer, Sambasivan, Rowe, & Ganger, 2013). Currently, relational databases are the most widely used mainstream databases and have been used as a back end to store time-series data (Rhea *et al.*, 2017). Note that traditional relational databases are incapable of dealing with massive time-series data. Nowadays, NoSQL databases have been extensively applied for Big Data management and processing in diverse applications, such as GIS (Guo & Onstein, 2020), cultural heritage (Abdelmoumni & Chenfour, 2022), social network (Lee, Jeon, & Song, 2020), and healthcare (Sen & Mukherjee, 2023). In particular, some efforts have been devoted to the use of NoSQL databases in sensor networks and the Internet of Things. In the Internet of Things scenario for manufacturing, Gamero *et al.* (2022) developed a decoupled architecture of SQL and NoSQL database management systems. They demonstrate that MySQL is favored for higher-order insights, and NoSQL can reduce system latency for known access patterns at the expense of integrated query flexibility. The Internet of Things uses many sensors and produces a large amount of sensing data. Mehmood, Culmone & Mostarda (2017) model temporal aspects of sensor data, develop a prototype for the MongoDB real-time platform, and discuss the temporal data modeling challenges and decisions.

Sensor data is typically a kind of stream data which can be stored in a NoSQL datastore for stream analytics (Mahmood, Orsborn, & Risch, 2020) and represented as time-series data. Some NoSQL-based storage models for massive time-series data have been developed (Di Martino *et al.*, 2019). Calatrava, Fontal, Cucchiatti, & Diví-Cuesta (2021) build a time-series database on top of MongoDB,

called NagareDB. The goal of NagareDB is to easily access three of the essential resources: hardware, software, and expert personnel. Based on the column-family database, Cassandra, Jensen, Pedersen & Thomsen (2021) design a multi-model online algorithm with a user-defined value error range and propose a general modular distributed architecture ModelarDB+. As a portable library, ModelarDB+ uses spark for query processing, which can significantly reduce development costs. Tsubouchi *et al.* (2019) introduce a time series database architecture HeteroTSDB, using Amazon Web Services, which automatically tiers on heterogeneous key-value stores. Bollen *et al.* (2023) propose the use of temporal graph databases to represent and query time series data in transportation networks. In (Ochiai, Ikegami, Teranishi & Esaki, 2014), based on HBase, the *RowKey* is redesigned for time-series data to provide Get and Set operations. However, their proposal does not support the range query of time-series data. In addition, OpenTSDB, a distributed, scalable time series database, is the product of redesign based on HBase; KairosDB stores time series in Cassandra. Different types of NoSQL databases generally have different performances (Matallah, Belalem & Bouamrane, 2020). Focusing on experimental analysis for time series data storage structures in databases, Li, Pu, Li & Xu (2023) compare and analyze the space and time consumed in processing bulk loading/insertion, range query, and aggregate calculation of time series data under row and column storage. They conclude that the choice of time-series storage should be based on the specific application scenarios. To fully utilize the respective advantages of different NoSQL databases and further improve the storage efficiency of massive time-series data, Ramesh, Sinha & Singh (2016) present several data modeling schemes in Cassandra and MongoDB to store the discrete time-series data, where Cassandra has a sequential data storage mechanism. MongoDB has a flexible schema and a rich query language.

To further improve the storage efficiency of massive time-series data, in-memory time series databases have been proposed, where the time-series data used more recently are saved in memory, and their reading performance can be significantly improved. Gorilla (Pelkonen *et al.*, 2015), Facebook's in-memory time series database, applies a simple 3-tuple to present a time-series data: a string key, a 64-bit time stamp integer, and a double precision floating point value. All time-series data are sharded based on these unique string keys, and each time series dataset is mapped to a single Gorilla host. Also, Gorilla leverages compression techniques like delta-of-delta timestamps and XOR'd floating point values to improve query efficiency by reducing Gorilla's storage footprint. Monarch (Adams *et al.*, 2020), Google's in-memory time series database, stores time-series data in schematized tables, and each table consists of multiple key columns, a value column for a history of points of the time series. Key columns (also referred to as fields) form the time series key, which has targets and metrics two sources. The in-memory time series database Heracles (Wang, Xue, & Shao, 2021) uses a two-level epoch-based memory mechanism, allowing in-memory data to be flushed and reclaimed gradually, where un-reclaimed data can still serve queries. Heracles is implemented based on Prometheus¹⁰, a representative open-source time-series database. In practice, the in-memory time series databases should be combined with disk-based time series databases for full time-series data management. For example, time-series data in Gorilla is written to an HBase data store, and a relational data model underlies Monarch's expressive query language for time series analysis.

The paper proposes an RHTSDB storage schema based on Redis and HBase for massive time-series data after stating time-series data are stored in memory and in NoSQL databases, where Redis functions as an in-memory time series database to improve data reading efficiency and HBase functions as a disk-based time series database to facilitate data storage scalability. The storage schema we use differs from NoSQL-based time-series storage solutions, which only use a single type of NoSQL database (e.g., Cassandra, HBase, or MongoDB) and only one type of in-memory storage solution. Additionally, our storage schema differs from existing works that combine in-memory and disk-based time series databases because we use databases specifically designed for time series rather than general purpose databases, even though they use general purpose databases as disk-based time series databases. In order to fill this gap, we use Redis as a time series in-memory database and HBase as a disk-based time series database. For massive time-series data, we develop a cold and hot separation

mechanism and provide a wide range of support for manipulating time-series data, such as insertion, update, single value query, range query, and multidimensional query.

STORAGE SCHEMA OF TIME SERIES DATABASE WITH REDIS-HBASE

Our storage schema RHTSDB (Redis-HBase Time Series Database) mainly comprises two modules: *Trie Tree Time Series* (TTTS) and *Redis-HBase Time to Live* (RHTTL). The TTTS module is responsible for data writing and query in Redis and provides the most basic data writing, single value query, range query, multidimensional query, and data update functions. The RHTTL is responsible for the data persistence function in HBase. In detail, RHTTL realizes the hot and cold separation mechanism of time-series data, where the infrequently accessed cold data are stored in HBase, and the frequently accessed data are stored in Redis, continuously maintaining the time-series data in Redis.

We formally describe our storage framework RHTSDB as a quintuple: $RHTSDB = \{TsS, ReS, HbS, MaS, OpS\}$. Among them, *TsS* is a set of original time-series data to be stored; *ReS* is the Redis database that stores the hot time-series data; *HbS* is the *HBase* database that stores the cold time-series data; *MaS*, is a set of mappings between the hot and cold time-series data; *OpS* is a set of operations (e.g., writing and querying operations) over the stored time-series data.

With the TTTS and RHTTL modules, RHTSDB supports time-series data mapping between Redis and HBase: refreshing some time-series data in Redis to HBase and loading some time-series data in HBase to Redis. For this purpose, we first design consistent keys for time-series data stored both in Redis and HBase to prevent the additional conversion of key values in data mapping and querying, and then propose a cache elimination strategy to ensure rational allocation of hot and cold time-series data both in Redis and HBase. In this case, the design of keys and cache elimination strategies must fully consider the structure and interaction of time-series data.

Trie Tree Time Series

As a module of the RHTSDB schema, the TTTS module is mainly responsible for the acquisition, query, and update of time-series data. To deal with time-series data, TTTS divides time-series data into four parts: *timestamp*, *metric*, *tags*, and *value*. The tags include *the dimension tag name* and *the corresponding dimension tag value* of time-series data. The value includes *the index name* and *the specific measurement value*. An example of time-series data division in TTTS is shown in Fig. 1.

The process of ingestion of TTTS includes:

- (a) All fields in time-series data are divided into four parts, including *metric*, *timestamp*, *dimension label*, and *measured values*.

In Fig. 2, for example, the *metric* of time series entity is device, the *timestamp* is 1652198940, and the *dimension label* is represented by a set of key value pairs: $\{\{\text{battery_status: discharging}\}, \{\text{device_id: demo0000}\}, \{\text{BSSID: A0:B1:C5:D2:E0:F3}\}, \{\text{SSID: wealth net}\}\}$. In addition, the *measured values* are represented by a set of key value pairs: $\{\{\text{battery_temperature: 96}\}, \{\text{BSSID:}$

Figure 1. Time-series data format

<i>timestamp</i>	<i>metric</i>	<i>tags</i>		<i>value</i>	
2022-05-09 15:10:00	device	<i>tag_{k1}</i>	<i>tag_{v1}</i>	<i>field_{k1}</i>	<i>field_{v1}</i>
2022-05-09 15:35:00	stock	<i>tag_{k2}</i>	<i>tag_{v2}</i>	<i>field_{k2}</i>	<i>field_{v2}</i>
2022-05-09 15:40:00	device	<i>tag_{k3}</i>	<i>tag_{v3}</i>	<i>field_{k3}</i>	<i>field_{v3}</i>
...

91.7}, {cpu_avg_1min: 5.26}, {cpu_avg_5min: 6.172}, {cpu_avg_15min: 6.510666667}, {mem_free: 650609585}, {mem_used: 34930415}, {rss: -42}}.

- (b) For the four parts above (except $field_v$), the corresponding dictionary tree needs to be built. We assign the corresponding number to each node and finally get a number combination represented by $metric$, $tags$, and $field_k$, with PRE_KEY means. At the same time, we also need to build an inverted index list for the dimension query.
- (c) $PRE_KEY+timestamp$ is regarded as the final stored key, and its value is the index value that corresponds to the $field_k$.
- (d) Redis stores each pair of key and value combinations in the List data type.

The query process of TTTS includes:

- (a) Query the PRE_KEY corresponding to $metric$, $tags$, and $field_k$ from the dictionary tree constructed by the four parts in memory. If there is no number, the query will end. Otherwise, the key will be returned, represented by $n_0n_1n_2$. The corresponding PRE_KEY is queried from the Inverted Index List if it is a multidimensional query.
- (b) According to time information, the key ($n_0n_1n_2t$) that stores the final single value is obtained for the time of a single value. If it is a range query, the keys within the time range are obtained, represented by $n_0n_1n_2t^*$.
- (c) Querying the corresponding data in the List.

To improve the searching performance in NoSQL databases, it is crucial to build and utilize a tree-based index (Karras *et al.*, 2022). In the TTTS module of RHTSDB, time-series data divided into four parts may contain too many tags and values, which have different timestamps. To speed up their searching, the dictionary tree is built, where each node is assigned with a corresponding number.

Redis-HBase Time to Live

Redis supports two storage mechanisms, AOF (Append Only File) and RDB (Redis DataBase). For the RDB snapshot mode, Redis forks a sub-process whenever data is saved, where the sub-process carries out the persistence work. When the dataset is relatively large, *fork* may be very time-consuming, causing the server to stop client processing within a certain millisecond. If the dataset is extensive and the CPU time is very tight, the stop time may be as long as a whole second. For the AOF mode, when the amount of data is large, the AOF file may not be able to restore the dataset to how it was saved when reloaded. It is necessary to modify the data in real-time or AOF simultaneously, which inevitably leads to both methods' inefficiency. Therefore, these two storage methods of Redis cannot solve the problem of massive time-series data persistence perfectly. To deal with the persistence of massive time-series data stored in Redis, RHTTL is designed and applied for scalability. With RHTTL, the infrequently accessed cold data are stored in HBase, and the frequently accessed data are stored in Redis. To implement the hot and cold separation mechanism of time-series data, RHTTL needs to continuously maintain the time-series data in Redis.

Figure 2. Time-series data sample

<i>metric</i>	<i>timestamp</i>	<i>battery_status</i>	<i>device_id</i>	<i>bssid</i>	<i>ssid</i>	<i>battery_temperature</i>
device	1652198940	discharging	demo000000	A0:B1:C5:D2:E0:F3	stealth-net	96
<i>bssid</i>	<i>cpu_avg_1min</i>	<i>cpu_avg_5min</i>	<i>cpu_avg_15min</i>	<i>mem_free</i>	<i>mem_used</i>	<i>rss</i>
91.7	5.26	6.172	6.5106666666667	650609585	34930415	-42

With the hot and cold separation mechanism of time-series data, realizing the mutual mapping of time-series data between Redis and HBase is essential. To this end, we need to comprehensively consider the design of key between Redis and HBase. In HBase, the structural design of *RowKey* is open to users on the one hand, but on the other hand, the *RowKey* design directly decides if the storage and query of data in HBase can finally get good performance (Ochiai, Ikegami, Teranishi & Esaki, 2014). In addition, the data model also directly affects the reading and writing performance of the time series database (Rinaldi *et al.*, 2019). So, it is not trivial to design the *RowKey* of HBase, and this is especially true for time-series data stored both in Redis and HBase. Combined with the key design in Redis, the final setting of *RowKey* in HBase is the same as that in Redis. The consistent key of time-series data stored in Redis and HBase can both avoid unnecessary conversion of keys when querying many data points are queries. In case the *RowKey* in HBase conflicts with the key in Redis or they need to go through a complex conversion, a bottleneck in reading performance will occur in the storage schema based on Redis and HBase. Then, unlike the key design of common data in HBase, the *RowKey* design of time-series data in HBase should consider timestamps. Based on our discussion in the TTTS module, we use “*PRE_KEY*+timestamp” as the *RowKey* of HBase. The table structure design of HBase with *RowKey* is shown in Fig. 3, where each value of *RowKey* (say, *PRE_KEY*+*day1*) corresponds to a column family that consists of a set of value pairs about fine-grained time points and the corresponding values (say, a value pair *time12: value12*).

Redis is an in-memory database, and its storage space is limited by memory size. So, it is essential to refresh some data from Redis to HBase so that only the latest data and the data with high query frequency, are stored in Redis. Moreover, we need a reasonable strategy for Redis’s cache replacement. An improper design of the data expiration rule may lead to a cache avalanche. An extreme case is that all data are set to be expired simultaneously, and all data in the cache are cleared at the same time. When this occurs, all query requests have to be sent to HBase for answer return, and this significantly degrades the system performance.

The original cache elimination strategy used in Redis only considers a single factor, such as the query frequency or write time (Li, Guo & Guo, 2019). In this paper, we present a cache elimination strategy based on the query frequency and update frequency of time-series data. The expiration time of time-series data can be calculated as follows.

$$RHTTL = \alpha \times \frac{PRE_KEY_{query}}{PRE_KEY_{update}} \times \beta + T$$

We use $\alpha \in [0,1]$ mainly to adjust the weight of query and update ratio. PRE_KEY_{query} and PRE_KEY_{update} represent query and update frequency of time-series data respectively. In addition,

Figure 3. RowKey and table structure design

RowKey		Column Family				
1	<i>PRE_KEY</i> ₁ + <i>day</i> ₁	<i>time</i> ₁₁	<i>time</i> ₁₂	<i>time</i> ₁₃	<i>time</i> _{1n}
		<i>value</i> ₁₁	<i>value</i> ₁₂	<i>value</i> ₁₃	<i>value</i> _{1n}
2	<i>PRE_KEY</i> ₂ + <i>day</i> ₃	<i>time</i> ₂₁	<i>time</i> ₂₂	<i>time</i> ₂₃	<i>time</i> _{2n}
		<i>value</i> ₂₁	<i>value</i> ₂₂	<i>value</i> ₂₃	<i>value</i> _{2n}
...					
t	<i>PRE_KEY</i> _m + <i>day</i> _n	<i>time</i> _{t1}	<i>time</i> _{t2}	<i>time</i> _{t3}	<i>time</i> _{tn}
		<i>value</i> _{t1}	<i>value</i> _{t2}	<i>value</i> _{t3}	<i>value</i> _{tn}

β and T are the initial values given by the model. By jointly setting and adjusting α , β and T , we can finally obtain a rational allocation of hot and cold time-series data in both Redis and HBase. To implement RHTTL, we create an RHTTL helper in Redis to record the auxiliary information of RHTTL. The RHTTL helper design is shown in Fig. 4.

The expired data in Redis needs to be refreshed to HBase, and at the same time, the cold data in HBase needs to be loaded to Redis. Given that Redis and HBase have different storage models, we need to map time-series data between Redis and HBase. First, for the mapping of time-series data from Redis to HBase, as shown in Fig. 5, we identify three scenarios to map the Redis data that will be persisted to HBase: (1) the keys maintained in the memory are periodically checked, and the expired data will be refreshed to HBase; (2) when the number of keys maintained in the memory exceeds the threshold set by the system, the corresponding data refresh mechanism is triggered; (3) the data to be refreshed are encapsulated according to their keys, which will be refreshed to HBase by calling myHBaseClient. After the data is refreshed from Redis to HBase, their corresponding RHTTL file needs a real-time update.

Then, for the mapping of time-series data from HBase to Redis, as shown in Fig. 6, we follow three major steps to map the HBase data that will be cached to Redis: (1) we traverse the rowkeys of all data to be cached; (2) for each item of rowkey, we calculate and obtain its column clusters; (3) we traverse each column cluster then cache the specific data to Redis. After the data is cached, the number of operations on the corresponding rowkey needs to be calculated, and the RHTTL file needs to be updated accordingly.

Finally, in our storage schema RHTSDB for time-series data, data ingestion and query processes are based on the TTTS module; after TTTS ingests and queries Redis data, the RHTTL module updates the RHTTL files and cold data in Redis and also loads hot data from HBase to Redis. In the following, we present data ingestion and query processes of RHTSDB.

Figure 4. RHTTL helper design

$PRE_KEY_1+day_1+\"ce\"$	$status_1$	$search_count_1$	$update_count_1$
$PRE_KEY_1+day_2+\"ce\"$	$status_2$	$search_count_2$	$update_count_2$
$PRE_KEY_1+day_1+\"ce\"$	$status_3$	$search_count_3$	$update_count_3$
.....
$PRE_KEY_m+day_n+\"ce\"$	$status_i$	$search_count_i$	$update_count_i$

Figure 5. Redis to HBase processing

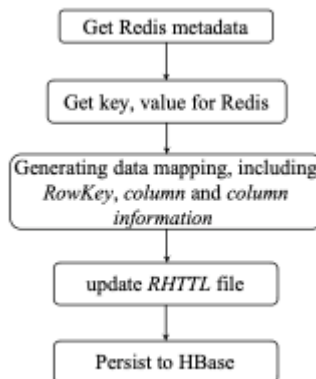
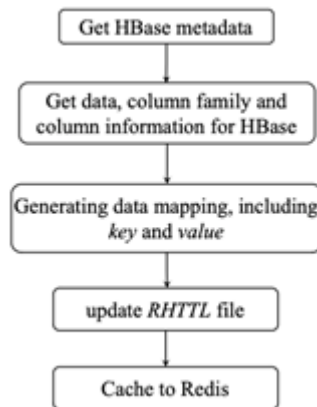


Figure 6. HBase to Redis processing



Ingestion and Update of RHTSDB

Based on the data ingestion function of TTTS for Redis, data ingestion of RHTSDB is implemented by mapping time-series data between Redis and HBase. Before ingesting data, it is needed to judge if the number of keys maintained by RHTTL files exceeds the given threshold. If so, RHTSDB refreshes the data that are maintained by the keys and have the lowest RHTTL value (i.e., colder keys) into HBase. Of course, this is a rare case because RHTSDB always maintains the RHTTL files after each operation so that smooth data insertion can be carried out next time.

For the data intake process, RHTSDB uses a specific data insertion command format: *RHTSDB.INSERT timestamp | metric tags_k_i:tags_v_i | field_k_i:field_v_i*. When an insertion operation is allowed in Redis, RHTSDB generates the corresponding *PRE_KEY* according to the *metric* and *tags*. According to the corresponding *timestamp*, *field_k* generates the keys for time-series data storage in Redis. Finally, RHTSDB performs the TTTS's insertion operation and then updates the RHTTL file again.

For the data update operation, RHTSDB uses a specific update command format: *RHTSDB.UPDATE metric tags_k_i:tags_v_ifield_k_i timestamp value*. The value in the command represents the latest collected value. First, RHTSDB uses *metric*, *tags*, and *field_k_i* to generate the *key* used for storage. If we need to update the values of multiple indicators, we will get a set of keys. Then, we judge if a key to be updated exists in the RHTTL file and if it exists in Redis or HBase. When the key exists in Redis, we call the update command of TTTS to update the data in Redis. If the key does not exist in Redis, we further query if there is a target key in HBase BloomFilter. If so, we will update the target data in HBase. Finally, RHTSDB updates all the fields that need to be updated and then updates the RHTTL file.

Query of RHTSDB

RHTSDB supports diverse queries of time-series data, including *single-value query*, *range query*, and *multidimensional query*. A single-value query queries the collection value of a specific collection index at a particular time point. The specific command format of a single-value query is:

```
RHTSDB.SELECT_ONE metric tags_ki:tags_vi field_ki timestamp.
```

In the single-value query, *timestamp* defines a time point. A range query is to query the collection value of a particular collection index in a given range. The time range may be tens of minutes or hours. The specific command format of a range query is:

```
RHTSDB.SELECT_RANGE metric tags_ki:tags_vi field_ki timestamp1  
timestamp2.
```

In the range query, *timestamp1* and *timestamp2* represent the start time and the end time, respectively. A multidimensional query enables users to query data within the corresponding time range according to the dimension label of time-series data. The specific command format of a multidimensional query is:

```
RHTSDB.SELECT_DIMENSION tags_ki: tags_vi timestamp1 timestamp2,
```

In the multidimensional query, *timestamp1* represents the start time, and *timestamp2* represents the end time. When *timestamp1* is equal to *timestamp2*, it means to query the collection value at a fixed time point. If a dimension is not limited in the multidimensional query, the corresponding dimension information can be replaced by * in the command.

With the hot and cold separation mechanism of time-series data, RHTSDB stores newly acquired data and frequently queried data in Redis and stores the data with low query frequency in HBase. For a given request of querying time-series data, RHTSDB first makes a search in Redis and then returns the answer satisfying the user request. In case the answers do not exist in Redis (meaning Redis query failed), RHTSDB then judges if there is a corresponding key in the bloom filter of HBase. If the key does not exist, the query finally ends, and no answer is returned. If the key exists, RHTSDB searches in HBase and returns the answer from HBase. The single-value, range, and multidimensional queries in RHTSDB follow a similar process: generating the tags, judging the query, and updating the RHTTL file. Note that for answers from HBase, the answers will be loaded to Redis as hot data. In the following, we present a generic algorithm for querying time-series data with RHTSDB.

Algorithm 1. The Query of Redis HBase Time Series Database

```

Data:
    Metric m ;
    Tags key-value pair tk1 : tv1, tk2 : tv2 ... tkn : tvn ;
    Field key field1, field2, ..., fieldk ;
    Start timestamp tstart, End timestamp tend ;

Result:
    The Set of Time Series value VTS ;
1  Setprekey ← validate(m, tk1 : tv1, ... tkn : tvn, field1, ..., fieldk);
2  Setkeys ← compute(Setprekey, tstart, tend);
3  if Setkeys = ∅ then
4  | VTS ← None;
5  else
6  | for key in set keys do
7  |   if Redis has key then
8  |   | VTS ← searchRedis(key);
9  |   | if VTS = ∅ then
10 |   | | excute HBase search Module;
11 |   | end
12 |   | update RHTTL files;
13 |   else
14 |   | if Bloomfilter has key then
15 |   | | VTS ← searchHBase(key);
16 |   | | cache data to Redis;
17 |   | | update RHTTL file;
18 |   | else
19 |   | | VTS ← None;
20 |   | end
21 |   | update RHTTL file;
22 |   end
23 | end
24 end

```

For a query given by the user, RHTSDB first generates the corresponding keys according to the query conditions. If the query conditions are illegal, Set_{keys} must be \emptyset . After obtaining the set of keys of query criteria, RHTSDB queries each key to determine if each key exists in Redis or HBase and then obtains the answer stored for each key. Note that RHTSDB does not judge if all keys are valid at one time. In addition, to keep the whole RHTSDB consistent concerning the existence of time-series data in Redis or HBase, RHTSDB will update the RHTTL file by using a hot and cold separation mechanism in real-time before the final result is returned and finally returns *RedisModuleReply*. RHTSDB completes the time-series data acquisition and query function through the mutual cooperation of TTTS module and RHTTL module, where *searchRedis()* relies on the TTTS module to perform specific query operations.

The time complexity of RHTSDB is mainly determined by querying RHTSDB. The worst situation is that for a given query, the possible answers do not exist in Redis (that is, answers are cold data rather than hot data), and HBase has to be searched for possible answers returned. The time complexity of RHTSDB is roughly comparable to that of HBase.

EXPERIMENT

We used the RHTSDB model proposed in this paper to perform massive time-series data ingestion and query. We report the experimental results in this section. The time-series data storage models used in the comparative experiment include HBase, MongoDB, InfluxDB, and Apache IoTDB, which were compared with RHTSDB over several metrics. As an essential part of Hadoop ecology, HBase has good expansion performance. Its underlying LSM tree structure can satisfactorily complete the production environment of large-scale modeling and a small amount of reading. MongoDB is the most widely used document-oriented database, which excels at read operations due to memory-loaded MongoDB register maps. We compared RHTSDB with HBase and MongoDB to show the effect of query speed optimization used in this paper. InfluxDB is a native time series database, which has been demonstrated to have high ingestion and query throughput of time-series data, compared with several databases, TimeScaleDB, Durid, and Cassandra (Shah, Jat & Sashidhar, 2022). Apache IoTDB is an emerging storage system focusing on large-scale IoT time-series data management. It provides many query and analysis functions, including single value query, range query, multidimensional query, and aggregation analysis. It has been demonstrated (Wang *et al.*, 2020) that Apache IoTDB has good performance in benchmark tests. Apache IoTDB is selected for comparison to better evaluate the comprehensive performance of RHTSDB. In the experiment, we mainly measure the uptake rate and storage consumption of time-series data in RHTSDB, HBase, InfluxDB, and Apache IoTDB as well as storage consumption of time-series data in RHTSDB, HBase, MongoDB, InfluxDB, and Apache IoTDB.

Experimental Environment

Our RHTSDB is developed in C++ language. As a third-party module, RHTSDB can assist Redis in work. We can run RHTSDB as a configuration file by adding *RHTSDB.so* files to the *redis.conf* configuration file. Alternatively, we can also use the module load command to load RHTSDB after Redis is started with *redis.so* file. The server used in the experiment is equipped with the Hadoop service corresponding to HBase. Considering that HBase is written in Java language and the external module of Redis uses C/C++ language, we used Thrift¹¹ programming framework to complete the communication between Redis and HBase. This experiment uses the pseudo-distributed method to test the performance of each model. The server is configured with 2 Intel (R) Xeon (R) CPU, 2.40GHz, 8GB of memory, 100GB of SSD, and the operating system is Ubuntu 20.04 LTS. The configuration information of other software versions installed in the experiment is shown in Table 1.

Table 1. Experimental Environment Configuration

Component	Version
JDK	Jdk-8u271-linux-x64
Hadoop	Hadoop-3.2.2
Thrift	Thrift-0.11.0
Tcl	Tcl-8.6.11
HBase	Hbase-2.3.7
Redis	Redis-6.2.5
Apache IoTDB	Apache IoTDB-0.13.0

Datasets

The datasets used in the experiment are the sensor information of IoT (Internet of Things) equipment provided by the TimescaleDB¹² official website. There are three datasets, including 1,000 devices recorded over 1000 time intervals, 5000 devices recorded over 2000 time intervals, and 3000 devices recorded over 10000 time intervals. From small to large, their sizes are 144.8MB, 1.45GB, and 4.34GB, respectively, corresponding to 1 million, 10 million, and 30 million of data. These datasets include indicators such as CPU, sensor time, device ID, memory, and network collected from mobile devices. According to the construction rules of RHTSDB for time-series data, we classify the collection indicators in the dataset. The data type of metric is a string, the time type is datetime, and the fields maintained by tag and field are shown in Table 2 and Table 3.

Table 2. Tag Information

Tag name	datatype
device_id	string
battery_status	string
bssid	string
Ssid	string

Table 3. Field Information

Field name	datatype
battery_level	int
battery_temperature	double
cpu_avg_1min	double
cpu_avg_5min	double
cpu_avg_15min	double
mem_free	int
mem_used	int
rssi	int

Data Ingestion Experiment

Data ingestion is the first step in processing time-series data. The time of data ingestion generally refers to the time of loading time-series data into the target time series database. It is widely adopted in the literature that the intake data points per second are used to evaluate the intake unit's performance. However, this way contains many manufactured factors. Sometimes, the specific definitions of data points are different, and this leads to the measurement being just a plan and having reference significance for itself. In addition, different databases have different formats for batch inserts of time-series data. For example, MongoDB usually uses the *mongoimport* command to import data directly; Redis usually adopts the batch command when inserting data. As a result, the direct data intake of different time series databases is not entirely unified. Therefore, in this paper, we use total time as an indicator to evaluate data uptake performance, namely, the time taken to absorb all time-series data in the unit of seconds.

In addition to time, storage resource consumption is an important metric often used for evaluating storage models. Storage resource consumption means the hardware (e.g., SSD) cost of storing time-series data. To save storage space, some time series databases often use compression methods to compress time-series data effectively. Gorilla (Pelkonen *et al.*, 2015), for example, uses Delta and XOR compression algorithms based on encoding compression and TRISTAN (Marascu *et al.*, 2014) algorithm based on dictionary compression, aiming to reduce the storage space of time-series data and save resources. Since time-series data compression may destroy the original time-series metadata and reduce the data uptake performance, our storage schema RHTSDB model does not compress the time-series metadata but retains the original structure of time-series data.

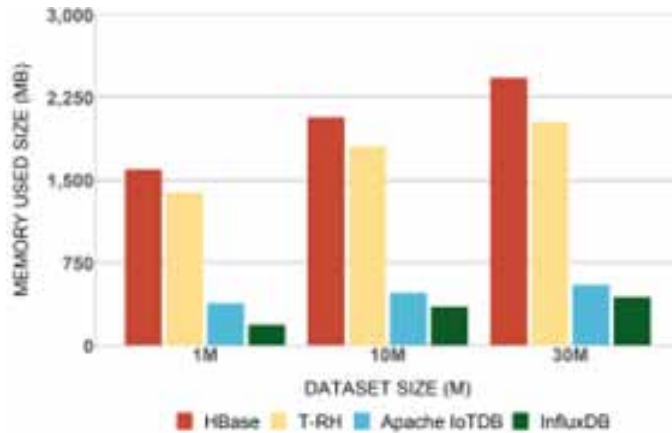
We compared the uptake rates and storage resource consumption of four storage models, HBase, RHTSDB (T-RH in short), Apache IoTDB, and InfluxDB, over three datasets of 1M, 10M, and 30M. The experimental results are shown in Fig. 7 and Fig. 8, respectively, where the unit of data ingestion is second (s) and the unit of storage consumption is MB.

Shown in Fig. 7, our storage schema RHTSDB takes the shortest execution time in data uptake efficiency. The reason is that RHTSDB uses Redis for the initial processing of time-series data, and Redis is an in-memory database. Apache IoTDB needs the longest time in data uptake efficiency mainly because it needs to spend time creating TsFile file and providing corresponding compression algorithm for time-series data, which must reduce its speed of time-series data ingestion to some extent. It is also shown in Fig. 8 that InfluxDB has a significant advantage in storage consumption because of the usage of a compression algorithm. In addition, Apache IoTDB has the second-best

Figure 7. Time of data ingestion



Figure 8. Size of storage consumption



performance in storage consumption because of its internal TsFile design and usage of a compression algorithm. The storage consumption of RHTSDB is slightly lower than that of HBase in each dataset. This is because RHTSDB internally manages the label information of time-series data, significantly reducing the label information redundancy in time-series data.

Data Query Experiment

To test the single value query function, we randomly selected 1000 time-series data from 30M point data using the method of random numbers. We tried to select data from different time points under different time series sources as much as possible:

- (1) we selected 500 data from these 1000 time-series data as successful query objects and generated the corresponding query commands,
- (2) we took 250 data from the remaining 500 data, modifying the labeled information of relevant dimensions in the time series source into non-existent dimensions. For the last 500 data, we modified the measurement metric into non-existent indicators to generate corresponding query commands.

Finally, we randomly ran the generated 1000 query commands and took the average execution time of all query commands as the final query results. Similarly, we generated corresponding test commands for range and dimension queries according to the above rules. However, we further stipulated the first rule above: we set the query time ranges as one day and two days for two halves of the datasets, respectively; for dimension query, we set its time range as two days.

With five storage models, HBase, RHTSDB (T-RH in short), Apache IoTDB, MongoDB, and InfluxDB, we evaluated their functions of *single value query*, *range query*, and *multidimensional query* over three datasets of 1M, 10M, and 30M. These three types of queries are commonly used in time series databases. Their experimental results are shown in Fig. 9, Fig. 10, and Fig. 11, respectively, where the unit of data query are in milliseconds (ms).

It can be observed from Fig. 9, Fig. 10, and Fig. 11 that our storage schema RHTSDB has the best performance in single-value, range, and multidimensional queries, with RHTSDB, MongoDB, and InfluxDB having similar performance in multidimensional queries. This is mainly due to the unique design of the RHTSDB model for the query scenario of time-series data and the usage of hot and cold separation mechanism. It is shown in Fig. 9 that the single value query delay of our RHTSDB is about 0.35 times that of HBase, 0.36 times that of Apache IoTDB, and 0.42 times that of MongoDB,

Figure 9. Response time of single value query

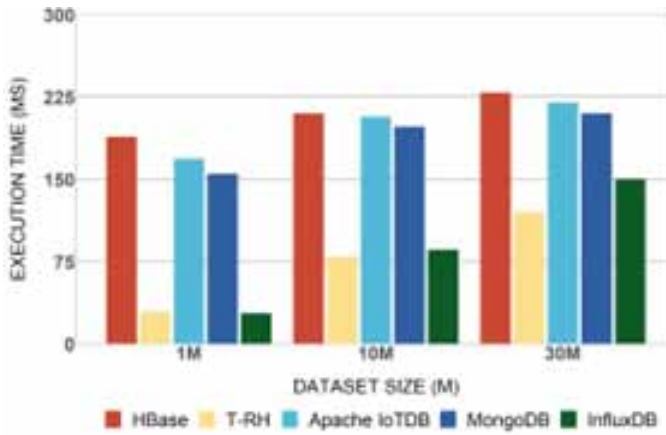


Figure 10. Response time of range query

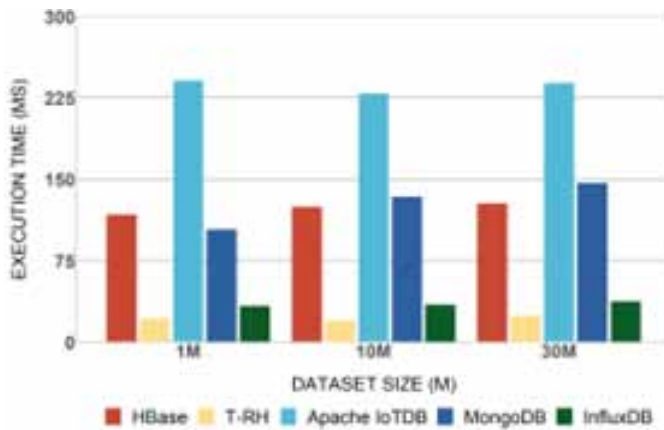
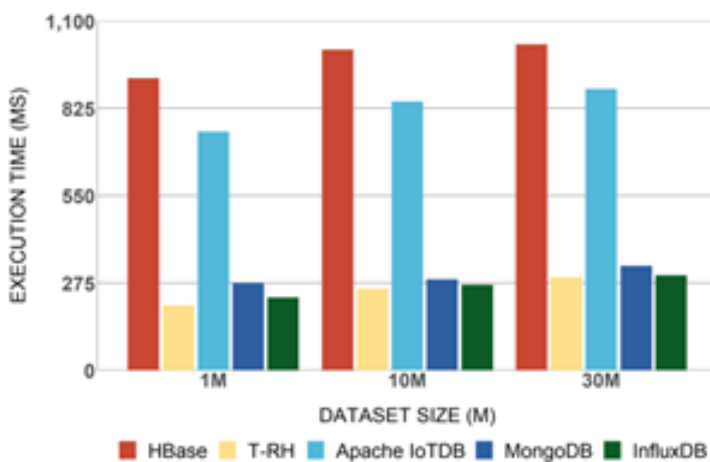


Figure 11. Response time of multidimensional query



respectively. It is shown in Fig. 10 that, for the range query function, the query performance of each storage model is very stable, and the range query delay of RHTSDB is only about 0.18 times that of HBase, 0.1 times that of Apache IoTDB, 0.18 times that of MongoDB, and 0.66 times that of InfluxDB, respectively. It is shown in Fig. 11 that, for the multidimensional query function, the performance of each storage model is relatively stable also, and the multidimensional query delay of RHTSDB is about 0.25 times that of HBase and 0.3 times that of Apache IoTDB.

Discussions

Time-series data ingestion and query are two important functions for massive time-series data storage. It is essential to evaluate time-series databases from these two aspects. Compared with four comparators, HBase, MongoDB, InfluxDB, and Apache IoTDB, our storage schema RHTSDB has the shortest response time in single value query, range query, and multidimensional query over three datasets of 1M, 10M, and 30M. This is mainly due to the fact that RHTSDB is specially designed for the query scenario of time-series data, and the corresponding separation mechanism of hot and cold data is used. For data ingestion, InfluxDB has the minimum storage consumption but a longer time in data uptake; RHTSDB has the shortest execution time in data uptake, but its storage consumption is higher than that of Apache IoTDB and InfluxDB and still lower than that of HBase. So, in summary, our proposed RHTSDB has the advantage of overall performance, but there is room for improvement in terms of storage consumption and data writing.

RHTSDB uses Redis as the cache and HBase as the backend storage, so the scalability of RHTSDB is mainly determined by HBase, a distributed and versioned NoSQL database with robust scalability. With RHTSDB, querying massive time-series data is first conducted in hot data stored in Redis and then in cold data stored in HBase if searching Redis fails. Ideally, the final answers can be returned only by searching Redis. Under this situation, there is no data ingestion and update, and data are considered durable. However, in practice, the absolute data duration under this completely ideal situation is almost impossible, and it is very common that some queries have to be evaluated against HBase. As long as such query evaluations happen infrequently, data can be considered durable. A worse case is that many queries against Redis fail, and they are finally evaluated against HBase. Under this situation, data are considered durable volatile. An extreme case is that no query can return an answer from Redis. Our storage schema RHTSDB proposes and applies a cold and hot separation mechanism for time-series data, which fully considers if data are recently/frequently accessed as well as dynamic maintenance of cold and hot data. This can largely guarantee the RHTSDB's fault tolerance in querying hot data. Of course, the usage of our cold and hot separation mechanism in actual applications must also consider the scale of time-series data to be processed and the memory capacity to be used simultaneously.

THEORETICAL AND PRACTICAL CONTRIBUTIONS

With the wide use and rapid growth of time series data, it has become essential to manage time series data, and diverse storage schemas for time series data have been invented accordingly, including the native time series databases and the time series databases based on NoSQL databases. NoSQL databases have been extensively adopted for data management of many applications in the era of big data. Compared with native time series databases, NoSQL-based time series databases can provide more complete support to time-series data management and an easier use of databases due to their mature techniques and wide applications. Existing NoSQL-based time series databases proposed in several works mainly use single types of NoSQL databases. Few works jointly use in-memory time series databases and disk-based time series databases, but the in-memory databases used are native and designed just for time series.

In this paper, we advocate the use of two different categories of NoSQL databases, Redis and HBase, for large-scale time-series data storage. We reveal that being an in-memory time series

database, Redis can improve time-series data intake speed and reduce redundant dimension labels of time-series data as well as disk space consumption of time-series data. Our proposed storage schema RHTSDB supports a hot and cold separation mechanism and can improve the cache hit rate. Based on two widely used NoSQL databases, RHTSDB is easily deployed and employed, meanwhile, RHTSDB preserves its good performance in the intake speed and query speed of large-scale time-series data.

CONCLUSION

Time-series data produced in the IoT scenario have salient characteristics, such as large volume, continuous generation, high-frequency writing, low-frequency reading, and fast query response. Aiming at their efficient management, we initiate an efficient storage schema RHTSDB for large-scale time-series data based on Redis and HBase in this paper. Among them, Redis can improve time-series data intake speed, reduce the redundancy of dimension labels in time-series data, and reduce disk space consumption by time-series data. Redis is highly dependent on memory, and its persistence scheme is not conducive to managing time-series data. For this reason, HBase is also used in RHTSDB for long-term time-series data persistence. For this hybrid storage schema, we propose an optimized hot and cold separation mechanism for RHTSDB, which can improve the cache hit rate. Instead of the LRU replacement strategy, the flexible way adopted in the paper can determine the specific RHTTL (Redis-HBase time to live) according to query update times. In the HBase layer, BloomFilter is added to save space cost and improve data query speed. The experimental results show that RHTSDB can improve the data intake speed to a certain extent. Compared with the storage schemas HBase and Apache IoTDB, RHTSDB has a significant advantage in data query speed, and its performance in executing range queries is close to Redis.

The hybrid storage schema RHTSDB proposed in the paper has shown some excellent performances, but it still has some space to further improve its deficiencies in memory space and data intake. First, RHTSDB only supports pseudo distribution at present, but a mature time series database should also have the function of distributed storage. Given that Redis and HBase have a distributed deployment function, we will extend RHTSDB to realize distributed deployment based on the distribution functions of Redis and HBase. Second, compared with HBase, the uptake rate of RHTSDB fails to increase significantly. A major reason is that RHTSDB is developed in Redis single thread mode, where a single thread is completed from network IO processing to actual read-write commands. In practice, we can introduce multiple IO threads for RHTSDB to process network requests and further improve the data intake rate. In addition, we plan to add a data compression module into RHTSDB so that the of memory and disk space consumption of massive time-series data can be reduced significantly. We will investigate efficient algorithms for time-series data compression. Finally, it is interesting to investigate the effect of some key parameters (e.g., query rate) on the performance of our approach by varying these parameters. In our future work, we will also evaluate RHTSDB against more time-series data storage models over large-scale time-series datasets.

REFERENCES

- Abdelmoumni, O., & Chenfour, N. (2022). ICHC framework: NoSQL data model and a microservices-based solution for a cultural heritage platform. *International Journal of Software Innovation*, 10(1), 1–16. doi:10.4018/IJSI.293272
- Adams, C., Alonso, L., Atkin, B., Banning, J., Bhola, S., Buskens, R., Chen, M., Chen, X., Chung, Y., Jia, Q., Sakharov, N., Talbot, G., Tart, A., & Taylor, N. (2020). Monarch: Google's planet-scale in-memory time series database. *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, 13(12), 3181–3194. doi:10.14778/3181-3194
- Bajaj, A., & Bick, W. (2020). The rise of NoSQL systems: Research and pedagogy. *Journal of Database Management*, 31(3), 67–82. doi:10.4018/JDM.2020070104
- Bollen, E., Hendrix, R., Kuijpers, B., Soliani, V., & Vaisman, A. (2023). Analysing river systems with time series data using path queries in graph databases. *ISPRS International Journal of Geo-Information*, 12(3), 94. doi:10.3390/ijgi12030094
- Calatrava, C. G., Fontal, Y. B., Cucchiatti, F. M., & Diví-Cuesta, C. (2021). NagareDB: A resource-efficient document-oriented time-series database. *Data*, 6(8), 91. doi:10.3390/data6080091
- Di Martino, S. (2019). Industrial internet of things: persistence for time series with NoSQL databases. In *Proceedings of the 28th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises* (pp. 340-345). IEEE. doi:10.1109/WETICE.2019.00076
- Eom, S., & Lee, K.-H. (2017). Incorporating spatial queries into semantic sensor streams on the Internet of Things. *Journal of Database Management*, 28(4), 24–39. doi:10.4018/JDM.2017100102
- Gamero, D., Dugenske, A., Saldana, C., Kurfess, T., & Fu, K. (2022). Scalability testing approach for Internet of Things for manufacturing SQL and NoSQL database latency and throughput. *Journal of Computing and Information Science in Engineering*, 22(1), 060901. Advance online publication. doi:10.1115/1.4055733
- Grolinger, K., Higashino, W. A., Tiwari, A., & Capretz, M. A. M. (2013). Data management in cloud environments: NoSQL and NewSQL data stores. *Journal of Cloud Computing: Advances. Journal of Cloud Computing (Heidelberg, Germany)*, 2(22), 22. Advance online publication. doi:10.1186/2192-113X-2-22
- Guo, D., & Onstein, E. (2020). State-of-the-art geospatial information processing in NoSQL databases. *ISPRS International Journal of Geo-Information*, 9(5), 331. doi:10.3390/ijgi9050331
- Hu, Y., & Dessloch, S. (2015). Temporal data management and processing with column oriented NoSQL databases. *Journal of Database Management*, 26(3), 41–70. doi:10.4018/JDM.2015070103
- Jensen, S. K., Pedersen, T. B., & Thomsen, C. (2021). Scalable model-based management of correlated dimensional time series in ModelarDB+. In *Proceedings of the 37th International Conference on Data Engineering* (pp. 1380-1391). IEEE. doi:10.1109/ICDE51399.2021.00123
- Karras, A. (2022). Query optimization in NoSQL databases using an enhanced localized R-tree index. In *Proceedings of the 24th International Conference on Information Integration and Web Intelligence* (pp. 391-398). doi:10.1007/978-3-031-21047-1_33
- Lee, C.-H., & Chung, C.-W. (2014). Compression schemes with data reordering for ordered data. *Journal of Database Management*, 25(1), 1–28. doi:10.4018/jdm.2014010101
- Lee, M., Jeon, S., & Song, M. (2020). Characterizing user interest in NoSQL databases of social question and answer data. *The Journal of Supercomputing*, 76(5), 3866–3881. doi:10.1007/s11227-018-2293-x PMID:32435085
- Li, C., Li, B., Bhuiyan, M. Z. A., Wang, L., Si, J., Wei, G., & Li, J. (2018). FluteDB: An efficient and scalable in-memory time series database for sensor-cloud. *Journal of Parallel and Distributed Computing*, 122, 95–108. doi:10.1016/j.jpdc.2018.07.021

- Li, K., Guo, K., & Guo, H. (2019). Financial big data hot and cold separation scheme based on HBase and Redis. In *2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking* (pp. 1612-1617). IEEE. doi:10.1109/ISPA-BDCIcloud-SustainCom-SocialCom48970.2019.00237
- Li, L., Pu, F., Li, Y., & Xu, J. (2023). A comparative study of row and column storage for time series data. *Proceedings of the 4th International Conference on Spatial Data and Intelligence*, 223-238. doi:10.1007/978-3-031-32910-4_16
- Mahmood, K., Orsborn, K., & Risch, T. (2020). Wrapping a NoSQL datastore for stream analytics. *Proceedings of the 21st International Conference on Information Reuse and Integration for Data Science*, 301-305. doi:10.1109/IRI49571.2020.00050
- Marascu, A. (2014). TRISTAN: real-time analytics on massive time series using sparse dictionary compression. In *Proceedings of the 2014 IEEE International Conference on Big Data* (pp. 291-300). IEEE. doi:10.1109/BigData.2014.7004244
- Matallah, H., Belalem, G., & Bouamrane, K. (2020). Evaluation of NoSQL databases: MongoDB, Cassandra, HBase, Redis, Couchbase, OrientDB. *International Journal of Software Science and Computational Intelligence*, 12(4), 71–91. doi:10.4018/IJSSCI.2020100105
- Mehmood, N. Q., Culmone, R., & Mostarda, L. (2017). Modeling temporal aspects of sensor data for MongoDB NoSQL database. *Journal of Big Data*, 4(1), 8. doi:10.1186/s40537-017-0068-5
- Ochiai, H., Ikegami, H., Teranishi, Y., & Esaki, H. (2014). Facility information management on HBase: large-scale storage for time-series data. In *Proceedings of the 38th International Computer Software and Applications Conference Workshops* (pp. 306-311). IEEE. doi:10.1109/COMPSACW.2014.54
- Ouaknine, K., Agra, O., & Guz, Z. (2017). Optimization of rocksdb for redis on flash. In *Proceedings of the 2017 International Conference on Compute and Data Analysis* (pp. 155-161). doi:10.1145/3093241.3093278
- Pelkonen, T., Franklin, S., Teller, J., Cavallaro, P., Huang, Q., Meza, J., & Veeraraghavan, K. (2015). Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, 8(12), 1816–1827. doi:10.14778/2824032.2824078
- Ramesh, D., Sinha, A., & Singh, S. (2016). Data modelling for discrete time series data using Cassandra and MongoDB. In *Proceedings of the 3rd International Conference on Recent Advances in Information Technology* (pp. 598-601). IEEE. doi:10.1109/RAIT.2016.7507966
- Rhea, S. (2017). LittleTable: a time-series database and its uses. In *Proceedings of the 2017 ACM International Conference on Management of Data* (pp. 125-138). ACM. doi:10.1145/3035918.3056102
- Rinaldi, S. (2019). Impact of data model on performance of time series database for internet of things applications. In *Proceedings of the 2019 IEEE International Instrumentation and Measurement Technology Conference* (pp. 1-6). IEEE. doi:10.1109/I2MTC.2019.8827164
- Sen, P. S., & Mukherjee, N. (2023). Ontology-based data modeling for NoSQL databases: A case study in e-healthcare application. *SN Computer Science*, 4(1), 3. doi:10.1007/s42979-022-01405-5
- Shafer, I., Sambasivan, R. R., Rowe, A., & Ganger, G. R. (2013). Specialized storage for big numeric time series. *Proceedings of the 5th USENIX Workshop on Hot Topics in Storage and File Systems*.
- Shi, X. (2020). ByteSeries: an in-memory time series database for large-scale monitoring systems. *Proceedings of the 11th ACM Symposium on Cloud Computing*, 60-73. doi:10.1145/3419111.3421289
- Solleza, F., Crotty, A., Karumuri, S., Tatbul, N., & Zdonik, S. (2022). Mach: a pluggable metrics storage engine for the age of observability. *Proceedings of the 12th Annual Conference on Innovative Data Systems Research*.
- Tsubouchi, Y. (2019). HeteroTSDB: An extensible time series database for automatically tiering on heterogeneous key-value stores. *Proceedings of the 43rd IEEE Annual Computer Software and Applications Conference*, 264-269. doi:10.1109/COMPSAC.2019.00046
- Van Erven, G. C. G., Carvalho, R. N., Cordeiro da Silva, W. M., Lifschitz, S., Vera-Olivera, H., & Holanda, M. (2019). Designing graph databases with GRAPHED. *Journal of Database Management*, 30(1), 41–60. doi:10.4018/JDM.2019010103

Wang, C. (2023). Apache IoTDB: A time series database for IoT applications. *Proceedings of the ACM on Management of Data*, 1(2), 195:1-195:27. doi:10.1145/3589775

Wang, C., Huang, X., Qiao, J., Jiang, T., Rui, L., Zhang, J., Kang, R., Feinauer, J., McGrail, K. A., Wang, P., Luo, D., Yuan, J., Wang, J., & Sun, J. (2020). Apache IoTDB: Time-series database for internet of things. *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, 13(12), 2901–2904. doi:10.14778/3415478.3415504

Wang, Z., Xue, J., & Shao, Z. (2021). Heracles: An efficient storage model and data flushing for performance monitoring timeseries. *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, 14(6), 1080–1092. doi:10.14778/3447689.3447710

Zhou, L., Lu, B., Zhang, S., & Qi, L. (2020). Data cache optimization model based on HBase and Redis. *Proceedings of the 3rd International Conference on Data Science and Information Technology*, 31-35. doi:10.1145/3414274.3414279

ENDNOTES

- 1 <https://www.influxdata.com/>
- 2 <https://redis.io/>
- 3 <https://hbase.apache.org/>
- 4 https://cassandra.apache.org/_/index.html
- 5 <https://www.mongodb.org/>
- 6 <https://www.couchbase.com/couchbase-server/overview>
- 7 <https://orientdb.org/>
- 8 <http://opentsdb.net>
- 9 <http://kairosdb.github.io>
- 10 <https://prometheus.io/>
- 11 <https://thrift.apache.org>
- 12 <https://docs.timescale.com/timescaledb/latest/tutorials/sample-datasets>

Ruizhe Ma is currently an assistant professor in the Department of Computer Science at the University of Massachusetts Lowell, USA. She received her Ph.D. and M.S. degrees from the Department of Computer Science at Georgia State University and her B.S. degree from Northeastern University, China. Dr. Ma's research focuses on data mining, machine learning, big data analytics, graph mining, and K-12 education.