Hindawi
Scientific Programming
Volume 2021, Article ID 9948533, 13 pages
https://doi.org/10.1155/2021/9948533

Research Article

The Embedded IoT Time Series Database for Hybrid Solid-State Storage System

Tao Cai, Peiyao Liu, Dejiao Niu, Jiancong Shi, and Lei Li

School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang, China

Correspondence should be addressed to Tao Cai; caitao@ujs.edu.cn and Dejiao Niu; djniu@ujs.edu.cn

Received 25 March 2021; Revised 28 August 2021; Accepted 12 October 2021; Published 25 October 2021

Academic Editor: Shah Nazir

IoT time series data is an important form of big data. How to improve the efficiency of storage system is crucial for IoT time series database to store and manage massive IoT time series data from various IoT devices. Mixing NVM and SSD is an effective method to improve the I/O performance of storage systems. However, there are great differences between HDD and NVM or SSD. As a result, NVM and SSD cannot be directly used in the current time series database effectively. We design an IoT time series database with an embedded engine in storage device drivers for the hybrid solid-state storage system consisting of NVM and SSD. The I/O software stack of storing and managing IoT time series data can be shortened to improve the efficiency. Based upon the intrinsic characteristics of IoT time series data and different features of NVM and SSD, a redundancy elimination and compression fusion strategy, a hierarchical management strategy, and a heterogeneous time series data index are designed to improve the efficiency. Finally, a prototype of embedded IoT time series database named TS-NSM is implemented, and YCSB-TS is used to measure the IOPS. The results show that TS-NSM can improve the write IOPS up to 243.6 times and 174.3 times, respectively, compared with InfluxDB and OpenTSDB, and improve the read IOPS up to 10.1 times and 14.4 times, respectively.

1. Background

Time series data is the sequential data with time correlation, commonly generated from social networks, scientific experiments, Internet of Things (IoT), and log systems. This is an important form of big data. The continuous generation, storing, and processing are the main characteristics of time series data. Especially in IoT, a large number of IoT devices concurrently generate a large amount of time series data with a fixed time interval, which brings great challenges to computer system for storing and processing [1]. The storage wall problem is a serious problem for IoT with HDD due to low read and write speed. In recent years, flash-based SSD with large capacity has become more popular, which has the I/O performance advantage compared with HDD. At the same time, nonvolatile memory (NVM) [2], such as Phase Change Memory (PCM) [3], Shared Transistor Technology Random Access Memory (STT-RAM) [4], and the latest technology Intel 3D-Xpoint [5] and Intel Optane DC Persistent Memory [6], provides features such as byte-addressable, long life span, low dynamic energy consumption, and high I/O speed closing to Dynamic Random Access Memory (DRAM). Then, a hybrid solid-state storage system with SSD and NVM can offer a possible way to solve the storage wall of time series data for IoT. However, the SSD and NVM also bring huge challenges to the I/O stack of computer system. The researches have shown that 94% delay in NVM storage systems is caused by file system, general block layer, etc. [7]. Therefore, it is of crucial importance to design a native IoT time series data management engine for hybrid solid-state storage systems.

Current DBMS is devoted to ensure transaction in data processing and is difficult to meet the requirements of time series data storage and management due to the lack of optimization strategies for continuous, high-speed, and concurrent massive data management [8]. Particularly in IoT, there are a large number of time series data generated by many different IoT devices in the meantime, which further increases the difficulty to store and manage IoT time series data with current DBMS. Accordingly, the time series

database is developed based on the current DBMS, which accords with the characteristics of time series data and optimizes some strategies such as data compression and timeliness. Then, it can continuously and concurrently store and manage massive time series data. Regarding IoT time series database, some special features are as follows: (1) continuous high-speed writing. As a large number of IoT devices expose multiple time series data, the IoT time series database needs to continuously store data with high throughput; (2) fast concurrent reading: each IoT device is relatively independent, so the IoT time series database should have high concurrent query capability to meet the different characteristics of each IoT device; (3) data compression: although the amount of IoT time series data is very large, the value often remains unchanged or changes very little over a period of time by using IoT devices. Then, data compression is very useful and critical for the IoT time series database; (4) time range queries: the query within a certain period of time is an important task of the IoT system; (5) self-management over time: IoT time series data has distinct timeliness, and the data access frequency will decrease sharply over time. It is important for the automatically adjustment strategy to store and index. Nevertheless, the present time series database management system is generally implemented by adding a time series management mechanism on the basis of common DBMS, which further increases the complexity of the I/O software stack for IoT time series data storage and management and cannot take advantage of the high I/O speed of SSD and NVM. There are also some differences between SSD and NVM. In spite of lower read and write speed, SSD just uses the block interface, and NVM is byte-addressable. Meanwhile, the current time series databases are designed for block or byte access interface storage devices, but they do not have the optimization strategy for the hybrid solid-state storage system composed of SSD and NVM. NVM's high read and write speed advantage cannot be used to improve the throughput of the current time series database due to the complex I/O software stack by using NVM instead of storage devices directly [9]. In addition, the write times of SSD and NVM are limited. In particular, a single unit in TLC-based SSDs just can write several hundred times. Meanwhile, there are a large number of the same or similar values in IoT time series data, which is very easy and necessary to do some optimization to reduce write amplification. IoT time series data also has different characteristics compared with time series data in social networks. All IoT time series data should be stored, and there is no deletion or update. At the same time, the time interval of series data is relatively fixed, and the data format is regular, which brings some convenience for IoT time series data management. However, IoT is also more time-sensitive for time series data storage and management, which poses a high challenge to the storage system. Thus, an important and urgent issue is how to design a new IoT time series database for the solid-state storage system composed of SSD and NVM.

In this paper, a new IoT time series data management engine is designed and embedded into the device driver of the hybrid solid-state storage system constituted by SSD and NVM. The NVM is used as a fast storage device in our design. The summary of our contributions is as follows:

(1) The IoT time series data storage and management engine is embedded in the storage device driver, which shortens the I/O software stack of IoT time series data management based on the hybrid solid-state storage system and avoids frequent exchange of large amounts of data between the host and the storage system. It can better utilize the high-speed read and write capabilities of NVM and SSD to improve the efficiency of IoT time series data management.

(2) The redundancy elimination and compression fusion strategy is designed according to the characteristics of IoT time series data and the hybrid solid-state storage system, which can reduce the storage consumption and the wear of SSD and NVM.

(3) The hierarchical structure is made by the access and management characteristics of IoT time series data, which can combine the advantages of NVM and SSD to improve the economic and scalability of the storage system.

(4) The two-dimensional hybrid index is designed for NVM, which combines hash and ordered linked list to improve the I/O performance and efficiency of IoT time series database.

(5) Block note index is designed to improve the query efficiency of IoT time series data stored in SSD.

(6) The prototype of an embedded IoT time series database for hybrid solid-state storage systems named TS-NSM is implemented to be tested and compared with the current popular time series databases such as InfluxDB and OpenTSDB. The results showed that TS-NSM can improve the write IOPS up to 243.6 and 201.2 times, respectively, and improve read IOPS up to 10.1 and 14.1 times, respectively.

2. Related Work

NVM has high I/O speed and byte-addressable interface, which can be used to build the hybrid solid-state storage system with SSD. There are many researches of hybrid solid-state storage system and database optimization for NVM.

2.1. Hybrid Storage System Based on NVM. NVM can be used to construct a hybrid memory with DRAM, and it also has a hybrid storage with HDD or SSD. Peiquan Lin constructed a hybrid memory with PCM and DRAM, where DRAM was mainly used as the cache of PCM to improve the lifetime of PCM [10]. A double dynamic bucket linked list was used to manage the space of PCM, and the age-based lazy cache strategy was designed to manage the cache in DRAM. Hibachi proposed a collaborative hybrid cache strategy based on NVM and DRAM [11]. The read and write cache were separate, and different management mechanism was designed for them to improve the hit rate. Meanwhile, the page dynamic adjustment mechanism was to adjust the size

of clean and dirty cache for different workloads. Chen et al. designed an efficient KV storage engine named FlatStore for the hybrid memory constructed by NVM and DRAM [12]. The log was stored in NVM to ensure reliable of storage, and DRAM was used to store the index to improve the search efficiency. An OpLog for each core was proposed to cache the small KV pairs, and the pipeline horizontal batch processing mechanism was designed to improve throughput and reduce read latency. LosPem is a novel log-structure framework for persistent memory constituted by NVM and DRAM to address the performance challenge [13]. It deployed an efficient hash-index linked list to maintain the log contents and reduce the significant overhead of log content retrieval. In addition, LosPem improved the transaction throughput by decoupling a transaction into two asynchronous steps and creating a write buffer based on DRAM to cache frequent data writes. In terms of NVM hybrid storage. The NVMCFS constructed a hybrid storage system with NVM and SSD [14], which used the head-tail layout and space management based on two-layer radix tree to provide unified logic space between two type NVM devices and used complex file structures, dynamic file data distributed strategy, buffer for an individual file, and asymmetric call in strategy to speed up the access response and improve I/O performance. Strata are a file system for the hybrid NVM storage system [15], which utilized the byte addressable ability of NVM to merge logs and migrated them to the underlying SSD/HDD to minimize the write amplification. However, file data can only be allocated in NVM and be migrated from the NVM layer to the SSD/HDD layer. Ziggurat is a multitiered NVM-based file system that spans NVM and HDD [16]. It is based on NOVA. Ziggurat exploited the benefits of NVM through intelligent data placement during file write and data migration. Ziggurat included two placement predictors that analyze the file write sequences to predict whether the incoming writes were both large and stable and whether their update to the file is likely to be synchronous. Then, it steered the incoming writes to the most suitable tier based on the prediction and writes to synchronously updated files go to the NVM tier to minimize the synchronization overhead. Small random writes also go to the NVM tier to entirely avoid random writes to HDD. The remaining large sequential writes to asynchronously updated files go to HDD. vNVML addressed the problem of combining a smaller, faster byte-addressable NVM with a larger and slower storage device, such as SSD, to create the impression of a larger and faster byte-addressable NVM, which can be shared across multiple applications concurrently [17]. vNVML provided application transaction-like memory semantics that can ensure write ordering, durability, and persistency guarantees across system failures. vNVML exploited DRAM for read caching to improve performance and potentially to reduce the number of writes to NVM and extend the NVM lifetime, whereas these new technologies still have some problems when used in the current database systems [18]. For example, disk-oriented database systems (such as Oracle RDBMS, IBM DB2, and MySQL) use block devices for persistent storage, which need to maintain an in-memory cache for tuple blocks and try to maximize sequential reads and writes to HDD with the bad performance of random access. As for memory-oriented database systems (such as VoltDB and MemSQL), they contain certain components to overcome the volatility of DRAM, while, in a byte-addressable NVM system with fast random access, such components are unnecessary.

2.2. Database Management System Based on NVM. NVM can provide features such as persistent storage, byte-addressable, low dynamic energy consumption, and high I/O speeds close to DRAM, yet the traditional database management systems (DBMS) cannot make full use of this technology, because their internal architecture is designed for DRAM or HDD. If NVM is used directly to replace DRAM or HDD, many of the components in these database systems are unnecessary and will reduce the efficiency of data-intensive applications based on it. Therefore, researchers have reconsidered the data storage and management methods in both volatile memory and persistent storage in the current DBMS. For example, BzTree is a lock-free B-tree index structure for NVM [19], and it replaced the index structures such as the black-and-white tree and the jump table in memory databases. BzTree mainly optimized PMwCAS in memory by using the NVM to greatly reduce the complexity of the traditional single-word CAS-based lock-free index, to improve the indexing efficiency of the database. Arulraj implemented six engines based on different storage management architectures in the modular DBMS test platform [9]; they are in-place update (InP), copy-on-write update (CoW), and log structure update (Log), and their variants on NVM such as NVM-InP, NVM-CoW, and NVM-Log; these variants mainly remove or modify unnecessary protection mechanisms. Their test results indicated that the three variant engines have better performance than the basic structure, and NVM-InP engine achieved the best throughput with the least amount of wear on the NVM device. At the same time, they found that the NVM access latency has the most impact on the runtime performance of the engine, more than the workload skew or the number of modifications to the database in the workload. Wang et al. presented a passive group commit method for a distributed logging protocol extension of Shore-MT [20, 21]. Rather than issuing a barrier for every processor when committing, the DBMS is tracked when all records required to ensure the durability of a transaction are flashed to NVM and allocated log buffers from local memory, so as to avoid remote DRAM access. It makes transaction-level partitioning more advantageous. This work is based on the Shore-MT engine, which means that the DBMS records page-level undo logs before performing in-place updates and will result in high data duplication. Shimin et al. described the unique characteristics of PCM and analyzed their potential impact on database management system [22]. It particularly puts forward analytic metrics for PCM endurance, energy, and latency and illustrated that current approaches for common database index algorithms such as B+-trees and Hash Joins are suboptimal for PCM. And then it introduced a new B+-tree and hash join, which has reduced both execution time

on PCM and write time. wB+Tree focused on the performance overhead of NVM write and CPU cache refresh to design the write atomic B+ tree [23], which resulted in reducing the number data to store in NVM by using an indirection slot array to minimize the movement of index entries and adopted a redo-only logging algorithm to ensure consistency. HiKV is a KV storage engine based on NVM [24], which established and retained a hash index in NVM to maintain its inherent fast index search capabilities and built a B+ tree index in DRAM to support range queries, thereby avoiding long-term NVM write and guaranteeing the index consistency. RangeKV is a persistent KV storage engine based on RocksDB, which is built on a heterogeneous storage architecture [25]. It used RangeTab in NVM to manage L0 layer data and increased L0 capacity to reduce LSM tree level and compression time overhead. RangeKV prebuilt a hash index of RangeTab data to reduce NVM access time and used a double buffer structure to reduce LSM tree write amplification from compression. HiLSM is the KV storage engine of the hybrid storage system with NVM and SSD [26]. According to the features of hybrid storage mediums, HiLSM adopted the hybrid data structure consisting of the log-structured memory and the LSM-tree. It proposed a fine-grained data migration strategy, a multithreaded data migration strategy, and a data filter strategy to address the issues such as write stalls in write intensive scenario, the performance gap between NVM and SSD, and the LSM-tree's inherent issue of write amplification. However, the optimization of database management systems for NVM is only limited to relational databases and KV store. The emerging time series database is still designed for DRAM or HDD, so their internal architecture still cannot fully utilize the advantages of NVM.

2.3. Time Series Database Management System. In order to store and analyze massive time series data, researchers have developed the special time series management system (TSMS or TSDBMS) to overcome the limitations of using general DBMS to manage the time series data [8]. Log-Structured Merge Tree (LSM-Tree) is very popular in the current time series database management systems [27], such as KairosDB [28] and OpenTSDB [29]. Meanwhile, InfluxDB [30] storage engine TSM-Tree is also the optimization based on LSM-Tree. LSM-Tree is designed for block storage engine. Its advantage is the great write performance, while its disadvantages are the poor read performance and write amplification. It used an update log to convert random write operations into sequential write and takes full advantage of the sequential writing of HDD, but it must reorder the written data and compress it layer by layer to reduce the cost of reading. This operation caused more serious write amplification. There are many other improvements in time series database management system. Yagoubi et al. mentioned a distributed parallel indexing method for time series databases [31], which used the correlation between ordered dimensions and adjacent values to achieve high scalability when having high performance of similarity query processing. Gorilla is an in-memory distributed time series database [32]. It proposed a three-level in-memory data structure, which is a shared index in memory, which can achieve high throughput of searching. However, due to the volatility and high cost of memory, it must use HBase to regularly back up and map nodes to ensure its availability in the case of single node or regional failures, and it takes delta-of-delta and XOR-based compression mechanism to minimize memory space overhead. There are some time series database management systems based on HDD as well. To overcome the problems of slow read and write speeds of HDD, they have designed a series of optimization mechanisms. For instance, LittleTable is a distributed relational database optimized for time series data [1]. To improve write performance, LittleTable spends more than half of the time on seeking data in the tablets for flushing the data into the HDD at once time together. Therefore, it requires a large amount of buffer size. To enhance query efficiency, LittleTable used two-dimensional clustering of relational tables. One is to divide the rows by timestamp, so the latest time series data can be retrieved quickly. And the other is to divide the keys by hierarchical structure, so that each partition can be further sorted to realize a fast indexing of time series data and flexible data table model optimization. However, LittleTable's consistency and durability guarantees were weak, and there is no batch delete function, because its underlying layer is still a relational database storage engine. ModelarDB is a general-purpose modular distributed time series database management system [33], which stores time series data as a model; hence, all operations are optimized on the model's mode, and multiple models can dynamically adapt to the data set. ModelarDB relies on Spark and Cassandra to manage time series data for accomplishing high-speed writing capabilities, data compression, and scalable online aggregation query processing capabilities. Nevertheless, the model-based design also limits ModelarDB to only be used for fixed-frequency time series, and the compression of time series data is lossy compression. Meanwhile, ModelarDB's write performance dropped sharply when there were many preselected models, as it needed try all models for each segment of the time series. Peregreen is a distributed modular time series database management system deployed in a cloud environment [34]. It is designed for great-scale historical time series data in the cloud. Peregreen takes a dual storage method that divides the time series data into segments and blocks and then merges them into the three-tier index. It can achieve efficient query and calculation with small network overhead. Peregreen also requires a large amount of buffer size to provide great write performance, since it bundles the timestamp and value into a pair of compressions. Accordingly, when there is a mixed time series, its compression ability is poor, resulting in the fact that the performance of reading data will be sharply reduced when using a remote storage device.

In summary, although the current researches can improve the performance of the hybrid NVM storage system and can improve the efficiency of relation database management system, they are not available time series data management system based on NVM. Existing time

series data management systems are designed for memory or HDDs. They lack data read, write, and management strategies designed for NVM, and an optimization mechanism that integrates block and byte interface storage devices. Therefore, they cannot better utilize the advantages of the hybrid solid-state storage system. In addition, the existing time series data management systems generally are based on the traditional database management systems, which also limits the efficiency owing to lacking of the native storage engine with the characteristics of time series data.

3. Analysis of IoT Time Series Database

When the current time series database is used in the IoT system, the problems are as follows:

(1) Lack of native IoT time series data management engine. Most of the time series databases are developed from general databases, such as OpenTSDB and TimescaleDB, which are based on Hbase and PostgreSQL, respectively. An extra interface layer for reading and writing of time series data is achieved for the general database. It not only failed to improve the storage and management efficiency on the basis of the characteristics of IoT time series data, but also increased the complexity of the I/O software stack for IoT time series databases. Then, it is difficult to take advantage of the fast read and write speed of SSD and NVM, and IoT time series databases become the bottleneck of IoT time series data management efficiency.

(2) Lack of optimization strategy for solid-state storage systems. Compared with HDD, SSD and NVM have higher read and write speeds. Especially for NVM, its read and write speeds are close to DRAM and support byte-addressable, which brings many challenges to the current storage system, such as their cache mechanism and merging small data into blocks. SSD and NVM also have limited write lifetime, and there are some differences in access interface and I/O performance between SSD and NVM. The current time series databases lack the corresponding optimization strategy to improve the IoT time data management efficiency by the respective advantages of SSD and NVM.

(3) Serious write waste. Different from general data, the values of IoT time series data will be the same or close to an adjacent point during a period of time, so it causes great write waste by writing them all to the storage device. Although there are some data compression mechanisms in the current time series database such as InfluxDB, KairosDB, and OpenTSDB, they still lack optimization for IoT time series data due to the difference between IoT and social networks. This will not only affect the efficiency of IoT time series database, but also reduce the lifetime of SSD and NVM.

(4) Weak concurrency for read and write IoT time series data. There are a large number of IoT devices in the IoT system, and each IoT device will continuously write data to the time series database according to its frequency. The current time series databases usually mix multiple time series sequence data in one file for storage and management and cannot optimize the storage and management efficiency by the different characteristics of IoT devices, such as frequency and value range.

(5) Lack of optimization strategies for timeliness. Compared with time series data from other systems, IoT time series data is more time-sensitive, which will bring obvious changes of access frequency in the time range, while the current time series databases such as InfluxDB, RRDTool, and OpenTSDB still use static index mechanism and cannot optimize the store and index efficiency by the access frequency change.

4. Embedded IoT Time Series Database

4.1. Architecture. The IoT time series data is various and large, so a high I/O performance and large capacity of storage system are needed. However, the IoT time series data has timeliness, the write is more frequent than read, and the access frequency changes over time. Then, the static strategy is not adaptable for the IoT time series data. A new embedded IoT time series database structure for hybrid solid-state storage systems was designed as shown in Figure 1.

The new system call is designed to bypass the file system and shorten the I/O software stack for the embedded IoT time series database. The embedded database engine is used to manage the IoT time series data in the device driver; it consists of the redundancy elimination and compression fusion module, the hierarchical management module, and the heterogeneous time series data index module. The redundancy elimination and compression fusion module is used to reduce the size and amount of data that should be stored in the hybrid solid-state storage system. The hierarchical management module is used to distribute the IoT time series data between SSD and NVM. In our design, NVM is used as a fast storage device. The heterogeneous time series data index module is used to index the data stored in SSD and NVM with different ways on the basis of their characteristics.

4.2. Redundancy Elimination and Compression Fusion Strategy. The IoT time series data has some special features compared with time series data in social networks. As shown in Figure 2, each IoT time series data includes the generation time, the value of IoT time series data, and the IoT device identifier. Therefore, each IoT time series data requires such a large storage space. However, the time interval of time series data from the same IoT device is fixed, and the value of IoT time series data always has the regularity that its value range is small and does not change in a period of time. These characteristics can be used to reduce the storage space of IoT

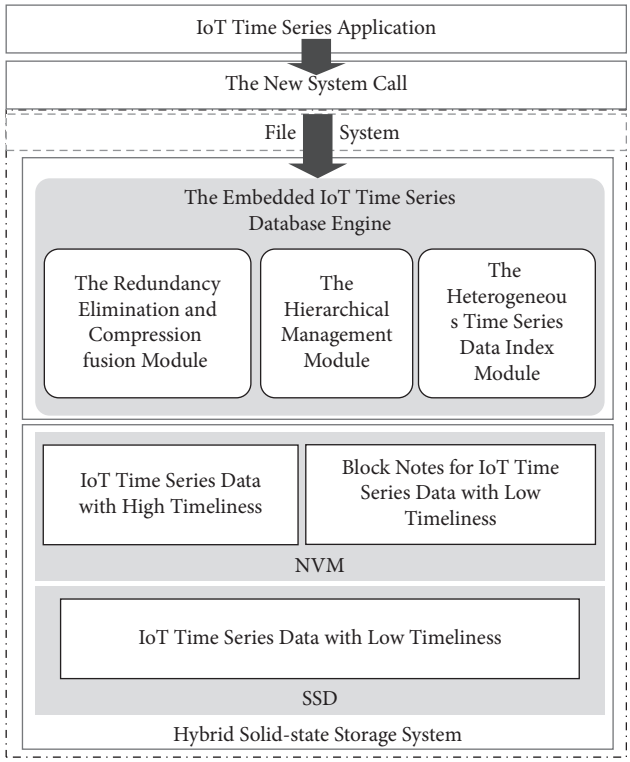

Figure 1: The structure of embedded IoT time series database for hybrid solid-state storage systems.

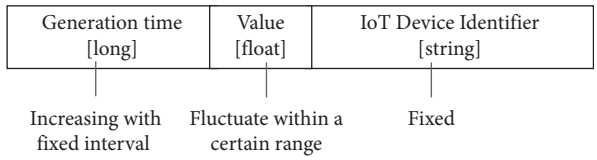

Figure 2: The original structure of an IoT time series data.

time series data. Then, a redundancy elimination and compression fusion strategy is designed.

Figure 3 shows the structure of the IoT time series data block. Each data block is 4 KB and consists of a block header area and a data area. The block header area includes start_time, start_value, p_num, last_point, and next_point, which, respectively, record the start time, the value of the first IoT time series data, repeat times of the first value, and pointers to the previous and next IoT time series data blocks. The data area includes 2032 short integers named D_0 to D_{2031}, which are used to store the compressed value of IoT time series data. In addition, the data in each IoT time series data block comes from the same IoT device.

On this basis, the redundancy elimination and compression rule is designed to reduce the size of the IoT time series data. Pseudocode 1 gives the flow of the rule.

When reading IoT time series data, the value can be restored according to formula (3) and formula (4) in 4.4.

By using the redundancy elimination and compression fusion strategy, most of the IoT time series data values represented by float numbers can be converted into a short integer for storage and thereby effectively will reduce the storage space consumption. Meanwhile, the first value of IoT time series data in each data block can be used to remove redundancy without complex calculation, which also can reduce the storage space consumption effectively. At the same time, the IoT time series data blocks still have good search performance, and the location of IoT time series data can be quickly obtained through simple calculations. In addition, it also can effectively improve the write performance of IoT time series data and reduce the consumption of storage device's lifetime.

4.3. Hierarchical Management Strategy. In IoT system, the access frequency of new IoT time series data is higher. The time interval and value range have differences for different IoT devices. On the basis of these characteristics, the hierarchical management strategy is designed.

Firstly, the storage space in NVM is divided into two parts, the IoT device area and data area. In the IoT device area, the structure shown in Figure 4 is used to store the features of each IoT device, which consists of IoT_device_ID, interval, value_precision, frequent_range, threshold, tags, block_address and start_time_SSD. IoT_device_ID is the IoT device identification. interval is the time interval of the IoT time series data. value_precision is the value precision of IoT time series data. frequent_range is the time domain of frequent query for each IoT device. *Threshold* is the migration threshold, which also means the maximum number of IoT time series blocks residing in NVM of each IoT device, calculated by formula (1). *Tags* are the address of IoT time series data's tags, which can be null, because some IoT time series data do not have tags. block_address is the address of the data area, the head IoT time series data block in NVM. start_time_SSD is the newest time of the IoT time series data stored in SSD corresponding to this IoT device.

Secondly, formula (1) is used to classify the IoT time series data as the high timeliness data category and low timeliness data category based on the access frequency for one IoT device. The high timeliness IoT time series data block is stored into NVM in the hybrid solid-state storage system, and the low timeliness IoT time series data blocks will be continuously migrated to the SSD from NVM.

$$\text{threshold} = \lceil \frac{\text{frequent\_range}}{\text{interval} * 2033} \rceil. \quad (1)$$

In formula (1), the migration threshold is the maximum number of high timeliness IoT time series data blocks. The earlier IoT time series data block will be migrated from NVM to SSD and converted to low timeliness data blocks.

Thirdly, when a high timeliness data block is converted into a low timeliness data block, the IoT_device_ID, start_time, and p_num stored in the IoT time series data block will be used to construct a key named Lblock_key. At the same time, the storage address of this IoT time series data block in the SSD is used as the value named Lblock_value. Then, as shown in Figure 5, the (Lblock_key, Lblock_value) KV pair is used as the block note of low timeliness IoT time series data block and stored in the NVM.

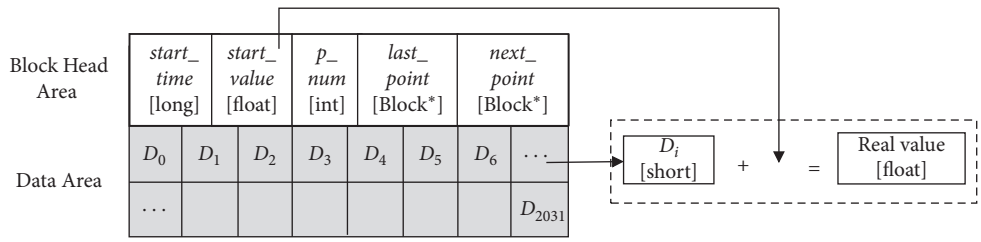

Figure 3: The structure of the IoT time series data block.

```
Void Block-compression (new_block, previous_block, current_time, current_value)
{   If (new_block is empty) //Initialized the IoT time series data block
{       The address of the previous IoT time series data block is filled by the value of next_point in the new_block.
   The current_time and current_value will be stored in the start_time and start_value in the new_block.
   block_start = 0; //Checking the top value of this IoT time series data block
   p_num = 0;}
else {
   if (new_block.start_value = = current_value) && (block_start = = 0) && (new_block is not full)
      p_num++; // The same value with the first IoT time series data is summarized and records in p_num.
   else
      if (new_block is not full)
         {The current_value and new_block.start_value will be amplified and the short integer delta of them will be calculated and
stored in D0 to D2031;
         block_start = 1; //Stopping to check the same value as the start_value. } }
```

Pseudocode 1: The redundancy elimination and compression rule.

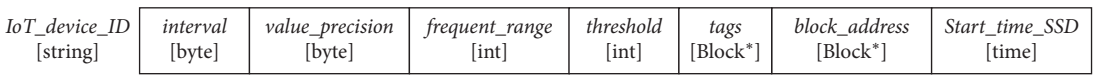

Figure 4: IoT device data structure diagram.

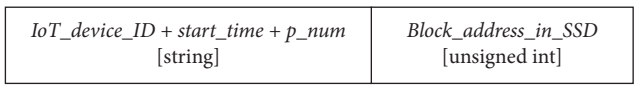

Figure 5: The structure of block note for low timeliness IoT time series data blocks.

On this basis, a migration algorithm for IoT time series data blocks is designed, which will migrate the older high timeliness IoT time series data blocks in the NVM into SSD to become the low timeliness IoT time series data blocks.

Figure 6 shows the diagram of IoT time series data distribution in the hybrid solid-state storage system. There are several independent storage areas for each IoT device in NVM to store the high timeliness IoT time series data blocks. The low timeliness IoT time series data blocks will be stored in SSD, but their block notes also are stored in NVM by several KV pairs.

By using the hierarchical management strategy, the IoT time series data can be distributed between SSD and NVM on the basis of their access frequency, which can be used to ensure the economics and efficiency of IoT time series database. At the same time, the time series data of different IoT devices are stored and managed independently, which can adapt to the characteristic difference of IoT devices and ensure the efficiency of the IoT time series data management. In addition, the block notes of IoT time series data block in SSD are stored in NVM, which can improve the search efficiency and reduce the write of SSD due to its block interface.

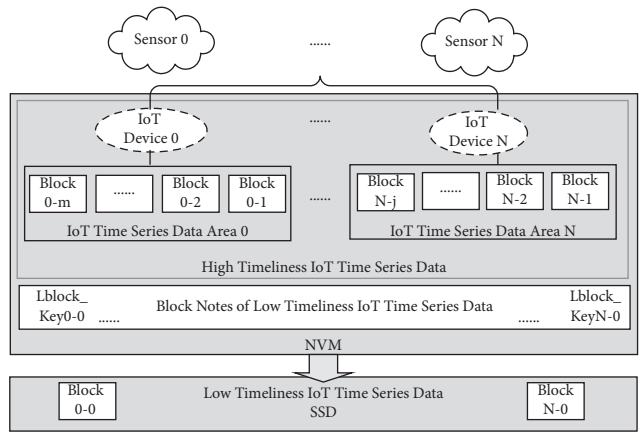

Figure 6: The diagram of IoT time series data distribution in a hybrid solid-state storage system.

4.4. Heterogeneous Time Series Data Index. In IoT systems, there are a large number of IoT devices, which bring a great challenge to search the IoT time series data in database. On the basis of 4.2 and 4.3, we design the heterogeneous time series data index mechanism, such that different methods are used to index the high and low timeliness IoT time series data. Its diagram is shown in Figure 7.

According to the distribution of IoT time series data in the hybrid solid-state storage system, the two-dimensional hybrid index is used to index the high timeliness IoT time series data on the NVM, and the block note index is used to the low timeliness IoT time series data on the SSD. At the same time, all indexes are stored in NVM to avoid the write amplification of SSD and use the I/O performance advantage of NVM, which can improve the efficiency of index for IoT time series database.

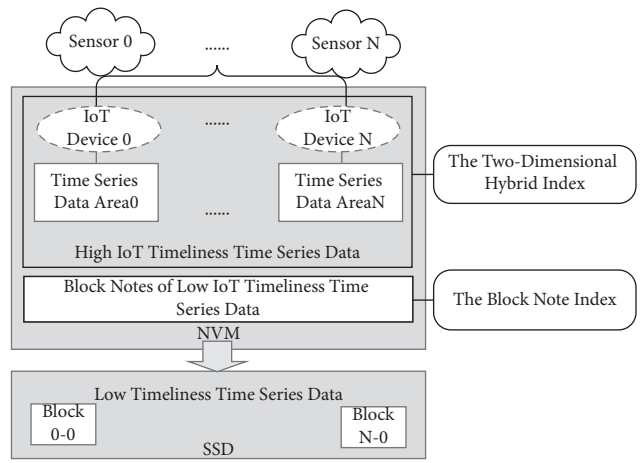

Figure 7: The diagram of heterogeneous time series data index.

4.4.1. Two-Dimensional Hybrid Index. Most of the searches in IoT systems target the high timeliness IoT time series data stored in NVM. The IoT time series database needs to identify the IoT time series data area at first and then search the corresponding IoT time series data block and the time series data. Meanwhile, most of the searches in IoT are the time range query.

A two-dimensional hybrid indexing is designed for the high timeliness IoT time series data. Firstly, the mid-square hash is used to identify IoT time series data area by the identifier of IoT device and get the address of the head IoT time series data block in this area. An inverted linked list is used to manage several IoT time series data blocks. Formula (2) is used to check whether the time series data block contains the time series data for search by start_time, start_value, and p_num stored in the head of each IoT time series data block.

$$\begin{cases} \text{search\_time\_start} \le \text{start\_time} + (p\text{\_num} + 2032) * \text{interval}, & (1), \\ \text{search\_time\_end} \ge \text{start\_time}, & (2). \end{cases} \tag{2}$$

In formula (2), search_time_start and start_time_end refer to the start and end time of the time range query. If formulas (1) and (2) both return true, this means that this data block is one of the target data blocks.

Then, formula (3) is used to calculate the position of the IoT time series data value in this block, and formula (4) is used to calculate the real value of the corresponding IoT time series data. If x is less than 0, the start_value is the value of query.

$$x = \frac{\text{search\_time} - \text{start\_time}}{\text{interval}} - p\text{\_unm} - 1, \tag{3}$$

$$\text{value} = \begin{cases} \text{start\_value} + D[x] * 10 - \text{value\_precision} & (x \ge 0), \\ \text{start\_value} & (x < 0). \end{cases} \tag{4}$$

In formula (3), search_time is the time point of IoT time series data query.

The two-dimensional hybrid index can be used to get the address of IoT time series data blocks efficiently. Compared with IoT time series data, the number of IoT devices is much smaller. The mid-square hash can be used to identify the IoT time series data area quickly, which can reduce the access times of NVM by using the powerful computing resource of CPU. The amount of high timeliness IoT time series data is also smaller compared to the total amount of time series data for each IoT device. Meanwhile, each time series data block can store more than 2033 values of IoT time series data, which also can reduce the time overhead of query. The access frequency of IoT time series data is decreased over time, and the interblock inverted linked list can reduce the time overhead of most query and avoid the frequent index change like B-tree.

4.4.2. Block Note Index. When the high timeliness IoT time series data block is converted to the low timeliness one, they will be migrated to the SSD. However, the low I/O

performance and block interface bring the challenge for query efficiency. Besides, the number of low timeliness IoT time series data blocks of one IoT device is much larger than that of high timeliness ones. However, the access frequency of the low timeliness IoT time series data is low as well, which provides some facilitates for the management of low timeliness IoT time series data blocks.

In 4.3, the block note of the low timeliness IoT time series data block is constructed and stored in NVM by KV pairs. As shown in Figure 8, the B-tree is used to index all block notes of low timeliness IoT time series data blocks in NVM. Then, all data for the query are stored in NVM. By the block note index, the start_time and p_num of the corresponding IoT time series data block can be got, and formula (2) can be used to check whether it contains the target time series data.

Therefore, the block node index can be used to reduce the access time of SSD to improve the efficiency of IoT time series database and improve the lifetime of SSD by using the NVM.

5. Prototype and Evaluation

Intel Optane DC Persistent Memory is a commercialized NVM storage device with the DIMM interface. PMEM [35] and NVMe [36] are open source device drivers for Intel Optane DC Persistent Memory and NVMe SSD. We modified the source code of PMEM and NVMe, and the redundancy elimination and compression fusion module, the hierarchical management module, and the heterogeneous time series data index module are added. Then, the IoT time series database engine can be embedded in PMEM and NVMe. Meanwhile, some new system calls are added in the Linux. Therefore, the prototype of the embedded IoT time series database for hybrid solid-state storage system is implemented, named TS-NSM. In TS-NSM, PEME is used as a fast raw storage device.

YCSB-TS is used as the test tool. It is a specialized performance testing tool for time series database. The Workloada and Workloadb in YCSB-TS are used as two workloads. There are a load stage and run stage in each workload. In the load stage, 1 million of time series data with one tag will be written into the database by the Workloada, and the same amount of time series data with three tags will be written into the database by the Workloadb. In the run stage, there are 1000 random queries in the Workloada. There are 1000 random range queries executed in the Workloadb, and there are 250 scans, 250 count, 250 sum, and 250 average operations for results.

The testing results will be compared with InfluxDB and OpenTSDB. InfluxDB is a time series database reimplemented from the bottom that does not rely on any other database. There is the batch mode to improve the write efficiency; we use InfluxDB-batch to delegate it. OpenTSDB is a distributed time series database based on HBase. During the test, HBase is run in all-in-one mode to avoid the network influence. The default configuration of InfluxDB and OpenTSDB is used.

The Intel Optane DC Persistent Memory is configured as App Direct mode and used as the fast block storage device for the prototypes of InfluxDB, influxdb-batch, and OpenTSDB. Meanwhile, the Ext4 is used as the file system for them. The hybrid solid-state storage system is used for TS-NSM constituent of Intel Optane DC Persistent Memory and NVMe SSD. Intel Optane DC Persistent Memory will retain 1000 of 4 KB IoT time series data blocks, and the rest of time series data will be stored in NVMe SSD.

One server is used for testing, which contains two 128 GB Intel Optane DC Persistent Memories and one NVMe SSD. Its configuration is shown in Table 1.

5.1. Write Performance. The load stage of Workloada and Workloadb in YCSB-TS is used to test the IOPS of write. The number of write threads is set to 1, 2, 4, 8, 16, and 32. The batch size of InfluxDB-batch is set to 10. The results are shown in Tables 2 and 3.

The results in Table 2 show that the written IOPS of TS-NSM is always the highest among all prototypes. Compared with InfluxDB, InfluxDB-batch, and OpenTSDB, its write throughput can increase by 86.5~243.6 times, 7.8~23.5 times, and 125.6~188.1 times, respectively. These results prove that TS-NSM can greatly improve the write IOPS of time series data. Although InfluxDB is a brand new time series database, its write throughput is still low. The batch mode can improve the write IOPS by about 10 times, but it is still lower than TS-NSM. OpenTSDB's write throughput is also lower compared with the all-in-one mode HBase. When there are two writing threads, OpenTSDB's write IOPS is close to that of the InfluxDB. The write IOPS of TS-NSM tends to increase firstly and then fall with more writing threads. When the number of writing threads is 8, the write IOPS of TS-NSM is the maximum. This is because the Intel Optane DC Persistent Memory still has performance limitations in multithreaded concurrent write. Meanwhile, the write IOPS of InfluxDB and InfluxDB-batch is continuously increased when increasing the number of write threads, but their write IOPS remains at a low level. OpenTSDB's write IOPS does not change significantly, despite the increase in the number of threads, and its highest and lowest throughput only differ by 54 ops. Meanwhile, OpenTSDB even has the lowest write IOPS after the write threads increased to 2. The reason is that OpenTSDB needs to store the data in Hbase, which seriously increases the complexity of the I/O software stack and extra time overhead. This also indicates that the I/O stack is a crucial factor affecting the write throughput of the time series database based on the solid state storage device. By contrast, TS-NSM's write IOPS increased by 29.3% after the number of writing threads increases from 1 to 8, and it can maintain high write IOPS after that. This is because TS-NSM embeds the time series database engine into the device drivers, which can shorten the I/O software stack and avoid unnecessary cache and replication.

Table 3 shows the test results of write throughput with the Workloadb. Similar to Workloada, the write IOPS of TS-NSM is consistently higher than that of InfluxDB, InfluxDB-

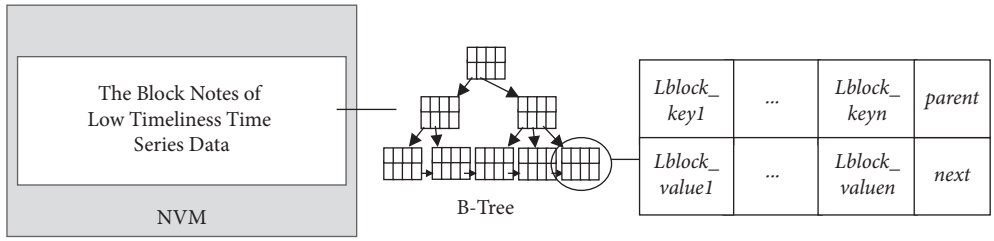

Figure 8: The block note index for low timeliness IoT time series data blocks.

Table 1: Test environment configuration.

Component	Configuration
CPU	Intel Xeon Platinum 8260 M 2.30 G
Memory	128 GB
NVDIMM	2 * 128 GB Intel Optane DC Persistent Memory
Disk	256 GB NVMe SSD
OS	CentOS 7.0 (Kernel Version 4.4.112)

Table 2: Write throughput on the Workloada.

The write throughput of workloada load stage (ops)						
TSDB	1thread	2thread	4thread	8thread	16thread	32thread
InfluxDB	459	981	1576	1442	1608	1652
InfluxDB-batch	4580	9248	15293	15192	15892	16112
OpenTSDB	887	894	867	840	853	870
TS-NSM	112283	130568	143730	158904	140697	151911

Table 3: Write throughput on the Workloadb.

The write throughput of Workloadb load stage (ops)						
TSDB	1thread	2thread	4thread	8thread	16thread	32thread
InfluxDB	440	665	1217	1322	1529	1601
InfluxDB-batch	4432	8145	13769	14897	15062	15867
OpenTSDB	803	841	821	820	809	816
TS-NSM	107415	118205	131873	143766	127855	139630

batch, and OpenTSDB. Compared with Workloada, there are more tags in each time series data in Workloadb, which brings more data written size. The write throughput of all prototypes is lower compared to that of Workloada. However, the difference of write IOPS between InfluxDB and InfluxDB-batch has increased; it increases from a maximum of 9.5 times under Workloada to 11.2 times under Workloadb, which indicates that the cache can reduce the write times and improve the efficiency surely when the amount of time series data is large. Compared with Workloada, the write IOPS of InfluxDB and InfluxDB-batch with Workloadb is reduced by 22.8% and 11.9%, while the write IOPS of TS-NSM is only reduced by 9.5%. This shows that TS-NSM has better adaptability, and the written IOPS is still high after increasing the amount of written data. However, the write IOPS of TS-NSM dropped even more significantly after the number of write threads increased more than 8 due to the bad adaptability of the Intel Optane DC Persistent Memory with concurrent write.

5.2. Query Performance. The run stages of Workloada and Workloadb in YCSB-TS are used to test the throughput of queries. The number of query threads is set to 1, 2, 4, 8, 16, and 32. The test results are shown in Figures 9 and 10.

Figure 9 shows the throughput of 1000 times random query with Workloada. It can be found that the random query throughput of TS-NSM is better than that of InfluxDB and OpenTSDB, and the query IOPS of TS-NSM improves 6.6~10.1 times and 2.2~14.4 times compared with that of InfluxDB and OpenSDB, respectively. The query IOPS of all prototypes shows a trend of first increasing and then decreasing with the increasing of query thread number. The differences are that InfluxDB's query IOPS reaches its peak when the number of query threads is 8, and it decreases by 4.5% and 9.0% when the number of query threads is 16 and 32. The query IOPS of TS-NSM and OpenTSDB reached its peak at 16 query threads, and the query IOPS of OpenTSDB decreased by 12.5% and that of TS-NSM only decreased by 4.2% with 32 query threads. Therefore, the TS-NSM has the

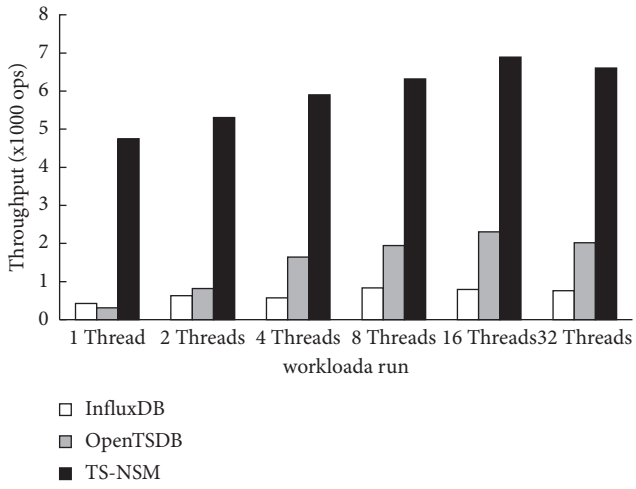

Figure 9: The query throughput with Workloada.

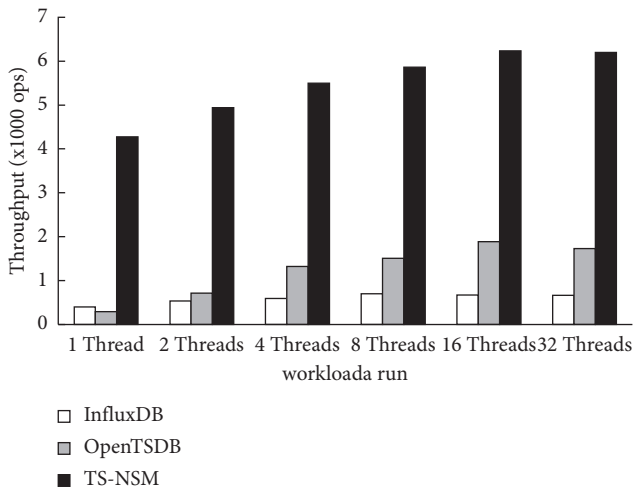

Figure 10: The query throughput of Workloadb.

more stable query throughput with several concurrent query threads, and it can better adapt to the large concurrent query for different IOT devices in the IoT system.

Time range query is very pervasive in the IoT time series database. Figure 10 shows the range query IOPS testing results by using Workloadb. Similar to Workloada, the range query IOPS of TS-NSM is much higher than that of InfluxDB and OpenTSDB, which is 7.5~9.8 times higher than that of InfluxDB and 2.4~13.8 times higher than that of OpenTSDB. This indicates that TS-NSM is also excellent in the range query throughput. Similar to the load stage, the query IOPS of using the Workloadb is generally lower than that of using Workloada. The range query IOPS of TS-NSM, InfluxDB, and OpenSDB also showed a trend of increasing first and then decreasing with the increase of query threads. The range query throughput of InfluxDB reaches a peak at 8 threads and then decreased by 4.3% and 5.8%. OpenTSDB and TS-NSM still reach a peak at 16 threads, and then OpenTSDB's IOPS is reduced by 8.2%, and TS-NSM's IOPS only decreases by 0.6%. Compared with Workloada, the

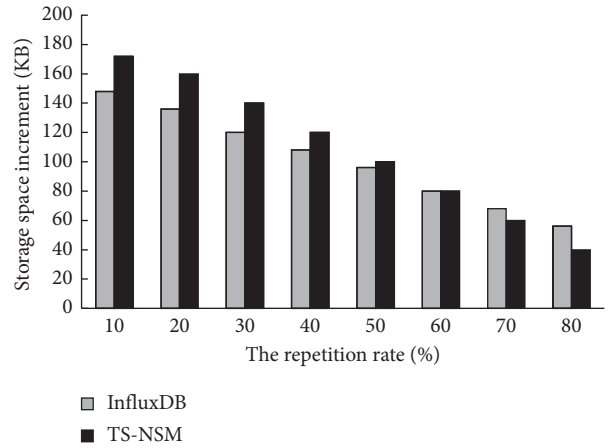

Figure 11: The storage space increment.

change rate of query IOPS is decreased with Workloadb, which indicates that TS-NSM has more advantages than InfluxDB and OpenSDB for range query. Meanwhile, the IOPS decrement of TS-NSM is also the lowest among the three prototypes, which further indicates that TS-NSM has better scalability of range query.

5.3. Compression Ratio. In general, the values of IoT time series data always do not change over a period of time. Therefore, a dataset containing 100, 000 of IoT time series data is built to test the compression ability of the prototype. The repetition rate of the dataset is from 10% to 80%. The storage space increments after inserting dataset are shown in Figure 11.

As shown in Figure 11, the compression ratio of TS-NSM is inferior to that of InfluxDB when the repetition rate of dataset is less than 50%, and the storage space increment of TS-NSM is much more than InfluxDB 4%–16%. This is because TS-NSM just uses the redundancy elimination and compression fusion strategy to achieve data compression and does not use common data compression algorithms such as XOR calculation difference of float point data and Snappy encoding of string used by InfluxDB. However, when the repetition rate of the dataset is more than 60%, the storage space increment of TS-NSM is always lower than that of InfluxDB. The space increment of TS-NSM is 29% less than that of the InfluxDB when the repetition rate of the dataset is 80%. Meanwhile, compared with InfluxDB, the compression of TS-NSM has less time overhead. These results indicate that TS-NSM can better adapt to the characteristics of IoT time series data to reduce the storage space overhead, because the repetition ratio is always high in IoT system, and there are a large number of IoT devices concurrently inserting time series data to the time series database.

6. Conclusion

Because IoT time series data is an important form of big data, storing and managing massive IoT time series data are a crucial task. NVM has the advantages of high I/O speed,

being nonvolatile, and being byte-addressable. And SSD has the advantages of high economy, large capacity, and higher I/O speed compared with HDD. Therefore, they can be mixed and constructed as a hybrid solid-state storage system to provide great support for the efficient storage and management of IoT time series data, while the current time series database lacks the corresponding optimization strategies. Based on the analysis of the characteristics of IoT time series data storage and management, an embedded IoT time series database for hybrid solid-state storage systems constructed by NVM and SSD is designed. Its structure and main algorithms are given. And the prototype named TS-NSM is implemented based on the device driver of NVM and SSD, which is verified by YCSB-TS, and the results show that the algorithms are effective.

Now, TS-NSM still lacks optimization mechanisms for multicore processors. In the future, we plan to make improvements in this regard to improve the storage and management efficiency of IoT time series database.

Data Availability

The data used to support the findings of this study are included within the article. For any further enquiries, the readers can contact the corresponding author.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was funded by the National Natural Science Foundation of China, grant no. 61806086, and the Project of National Key R&D Program of China, grant no. 2018YFB0804204.

References

[1] S. Rhea, E. Wang, E. Wong, N. Storer, and E. Atkins, "LittleTable: a time-series database and its uses," in *Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 125–138, ACM, New York, May 2017.

[2] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson, "A high performance storage array architecture for next-generation, non-volatile memories," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 385–395, IEEE, Washington DC, December 2010.

[3] J. Li and C. Lam, "Phase change memory," *Science China Information Sciences*, vol. 54, no. 5, pp. 1061–1072, 2011.

[4] K. Kuan and T. Adegbija, "Mirrorcache: an energy-efficient relaxed retention L1 STTRAM cache," in *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, pp. 299–302, ACM, Tysons Corner VA USA, May 2019.

[5] U. D. Dadmal, R. S. Vinkare, P. G. Kaushik, and S. A. Mishra, "3D X point technology," *International Journal of Electronics, Communication and Soft Computing Science and Engineering*, pp. 13–17, 2017.

[6] Intel, "Intel optane dc persistent memory," 2019, https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[7] S. Swanson and A. M. Caulfield, "Refactor, reduce, recycle: restructuring the I/O stack for the future of storage," *Computer*, vol. 46, no. 8, pp. 52–59, 2013.

[8] S. K. Jensen, T. B. Pedersen, and C. Thomsen, "Time series management systems: a survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 11, pp. 2581–2600, 2017.

[9] J. Debrabant, J. Arulraj, A. Pavlo, M. Stonebraker, S. Zdonik, and S. R. Dulloor, "A prolegomenon on OLTP database systems for non-volatile memory," in *Proceedings of the 40th International conference on Very Large Data Bases*, pp. 57–63, Hangzhou, China, 2014.

[10] P. Q. Jin, Z. L. Wu, X. L. Wang, X. Hao, and L. Yue, "A page-based storage framework for Phase change memory," in *Proceedings of the 2017 International Conference on Massive Storage Systems and Technology*, pp. 152–164, IEEE, Piscataway, NJ, USA, May 2017.

[11] Z. Q. Fan, F. G. Wu, D. Park, J. Diehl, D. Voigty, and D. Du, "Hibachi: a cooperative hybrid cache with NVRAM and DRAM for storage arrays," in *Proceedings of the 33rd International Conference on Mass Storage Systems and Technologies*, IEEE, Piscataway, NJ, USA, May 2017.

[12] Y. M. Chen, Y. Y. Lu, Y. Fan, Q. Wang, Y. Wang, and J. Shu, "An efficient log-structured key-value storage engine for persistent memory," in *Proceedings of the Architectural Support for Programming Languages and Operating Systems*, pp. 1077–1091, ACM, Lausanne, Switzerland, March 2020.

[13] S. Li and L. Huang, "LosPem," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 16, no. 3, pp. 1–17, 2020.

[14] T. Cai, D. J. Niu, Y. He, and Z. Yeqing, "NVMCFS: complex file system for hybrid NVM," in *Proceedings of the 2016 IEEE 22nd International Conference on Parallel and Distributed Systems*, pp. 577–584, IEEE, Wuhan, China, December 2016.

[15] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson, "Strata: a cross media file system," in *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, pp. 460–477, ACM, Shanghai, China, October 2017.

[16] Z. Shengan, H. Morteza, and S. Steven, "Ziggurat: a tiered file system for non-volatile main memories and disks," in *Proceedings of the17th Conference of File and Storage Technologies*, pp. 207–219, USENIX Association, Berkeley, CA, USA, February 2019.

[17] C. C. Chou, J. Jung, A. L. N. Reddy, P. V. Gratz, and D. Voigt, "Virtualize and share non-volatile memories in user space," *CCF Transactions on High Performance Computing*, vol. 2, no. 1, pp. 16–35, 2020.

[18] J. Arulraj, A. Pavlo, and S. R. Dulloor, "Let's talk about storage & recovery methods for non-volatile memory database systems," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 707–722, ACM, Melbourne, Australia, May 2015.

[19] A. Joy, L. Justin, F. M. Umar, and P.-A. Larson, "BzTree: a high-performance latch-free range index for non-volatile memory," *Proceedings of the VLDB Endowment*, vol. 11, no. 5, pp. 553–565, 2018.

[20] T. Wang and R. Johnson, "Scalable logging through emerging non-volatile memory," *Proceedings of the VLDB Endowment*, vol. 7, no. 10, pp. 865–876, 2014.

[21] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi, "Shore-MT: a scalable storage manager for the multicore era," in *Proceedings of the 12th International Conference on Extending Database Technology*, pp. 24–35, ACM, Saint Petersburg, Russia, March 2009.

[22] C. Shimin, P. B. Gibbons, and S. Nath, "Rethinking database algorithms for Phase change memory," in *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, pp. 21–31, CIDR, Asilomar, CA, USA, January 2011.
[23] S. Chen and Q. Jin, "Persistent B + -trees in non-volatile main memory," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, pp. 786–797, 2015.
[24] F. Xia, D. Jiang, J. Xiong, and N. Sun, "HiKV: a hybrid index key-value store for DRAM-NVM memory systems," in *Proceedings of the 2017 USENIX Conference on USENIX Annual Technical Conference*, pp. 349–362, USENIX, Berkeley, CA, USA, July 2017.
[25] L. Zhan, K. Lu, Z. Cheng, and J. Wan, "RangeKV: an efficient key-value store based on hybrid DRAM-NVM-SSD storage structure," *IEEE Access*, vol. 8, no. 99, p. 1, 2020.
[26] W. Li, D. Jiang, J. Xiong, and Y. Bao, "HiLSM: an LSM-based key-value store for hybrid NVM-SSD storage systems," in *Proceedings of the 17th ACM International Conference on Computing Frontiers*, pp. 208–216, ACM, Catania, Italy, May, 2020.
[27] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.
[28] D. B. Kairos, "Fast time series database on Cassandra," http://kairosdb.github.io/.
[29] "OpenTSDB scalable time series database (TSDB)," http://opentsdb.net/.
[30] "Influxdb.com: InfluxDB–Open Source Time Series, Metrics, and Analytics Database, ," , 2015.
[31] D. E. Yagoubi, R. Akbarinia, F. Masseglia, and T. Palpanas, "Massively distributed time series indexing and querying," *IEEE Transactions on Knowledge and Data Engineering*, vol. 32, no. 1, pp. 108–120, 2020.
[32] T. Pelkonen, S. Franklin, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan, "Gorilla: a fast, scalable, in-memory time series database," in *Proceedings of the VLDB Endowment*, pp. 1816–1827, Trondheim, Norway, September 2005.
[33] S. K. Jensen, T. B. Pedersen, and C. Thomsen, "ModelarDB," *Proceedings of the VLDB Endowment*, vol. 11, no. 11, pp. 1688–1701, 2018.
[34] V. Alexander, S. Alexey, Y. Semen et al., "Peregreen– modular database for efficient storage of historical time series in cloud environments," in *Proceedings of the 2020 USENIX Annual Technical Conference*, pp. 589–601, USENIX, Berkeley, CA, USA, July 2020.
[35] Peme: http://pmem.io/.
[36] NVMe: https://nvmexpress.org/.