Received August 17, 2020, accepted September 17, 2020, date of publication September 29, 2020, date of current version October 9, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3027610

HyperBSA: A High-Performance Consortium Blockchain Storage Architecture for Massive Data

XIAO CHEN[1], KEJIE ZHANG[2], XIUBO LIANG[3], WEIWEI QIU[2], ZHIGANG ZHANG[1], AND DING TU[1]
[1]CFETS Information Technology (Shanghai) Company Ltd., Shanghai 201210, China
[2]Hangzhou Qulian Technology Company Ltd., Hangzhou 310051, China
[3]College of Software Technology, Zhejiang University, Ningbo 315048, China

Corresponding author: Xiubo Liang (xiubo@zju.edu.cn)

This work was supported in part by the National Key Research and Development Program of China under Grant 2019YFB1404903.

ABSTRACT With the considerable exploration of blockchain in various industrial fields, the storage architectures of mainstream consortium blockchains exhibit significant performance limitations, which can't meet the requirements of efficient data access with massive data storage in enterprise-level business scenarios. In this paper, we creatively divided the underlying data of the consortium blockchain into two categories: continuous data and state data and proposed a new storage architecture to store and operate these two types of data efficiently. For continuous data, we designed a specialized index-based storage engine. For state data, we proposed a multi-level cache mechanism with a secure and integrated data persistence policy. In addition, a pluggable Client/Server mode is employed to achieve flexible distributed extension. A series of experiments are conducted to show the effectiveness of our architecture. Compared with mainstream consortium blockchain storage architecture based on LevelDB, the average time-consuming decreases 81.85%/82.47% for reading/writing continuous data and 22.21%/48.99% for reading/writing state data. Compared with the storage architecture based on distributed database TiKV, the time-consuming decreases more significantly. This storage architecture has been integrated into the enterprise-level consortium blockchain platform Hyperchain, which has supported the efficient running of dozens of large-scale commercial blockchain projects with massive data.

INDEX TERMS Consortium blockchain, storage architecture, continuous data, state data, distributed extension.

I. INTRODUCTION

Blockchain is a distributed storage technological paradigm which integrates many technologies such as encryption algorithm [1], peer-to-peer network [2], consensus algorithm [3], etc. It is regarded as one of the most subversive and revolutionary technological innovations in recent years because of its characteristics of decentralization, non-tampering of information and traceability of the whole process [4]. However, large-scale commercial blockchain applications require high-performance and extensible storage architecture, the traditional architectures of public blockchain cannot meet the requirement. Therefore, the consortium blockchain which has better storage performance than public blockchain is designed to deal with large-scale transactions for business scenarios [5]. However, there are still some storage bottlenecks in existing consortium blockchain platforms. The underlying storage of consortium blockchains are based on key-value database, such as LevelDB. With the increasing of data volume, it will cause constant reading/writing amplification in LevelDB. According to our test, the read amplification rate reaches more than 300% when the data volume reaches 100G. The increasing data results in the constant compaction of LevelDB, which reduces the storage efficiency and makes blockchain reach the performance bottleneck. In addition, key-value databases are not suitable for the storage of big single pieces of data. With the increasing of the size of the transaction data, the storage delay increases exponentially. Therefore, the mainstream consortium blockchain platforms are still very poor at handling massive data storage.

The associate editor coordinating the review of this manuscript and approving it for publication was Chakchai So-In.

Some researches have been carried out to improve the storage performance of the consortium blockchain. Most of them tried to reduce data storage volume rather than reforming the underlying architecture. Compressing node data [6] and processing data collaborative storage [7] are the two main kinds of ways to reduce the amount of data in blockchain nodes. However, reducing the amount of data excessively will lead to incomplete record in the blockchain, which is not a good idea. Some researchers also consider distributed extension [8] to solve the problem of storing massive data, among which distributed file system is widely adopted. However, the proposed architectures focus on the storage extension rather than storage performance, which cannot meet the requirement of the efficient reading/writing for enterprise-level business scenarios.

Compressing data to improve storage performance is an obvious solution, but not a good one. Mature commercial consortium blockchain platforms (e.g. Ethereum [9] and Hyperledger Fabric [10]) haven't adopt this kind of solution. In essence, the above researches didn't fundamentally solve the storage performance problem because they didn't notice the different characteristics of blockchain data and fail to improve the underlying storage architecture according to different data types. On the basis of carefully analyzing the characteristics of the underlying block data, we found that some data has continuous characteristic while the other does not. With different storage technologies adopted to process the two types of data separately, high performance of reading/writing in large data volume can be achieved. Therefore, we creatively divide data[1] in blockchain into two categories: continuous data and state data. A new storage architecture named HyperBSA is proposed to store and operate these two types of data efficiently. For continuous data (its key increases continuously, such as block data, transaction receipts and transaction logs), we designed an index-based storage engine to quickly access this kind of data. For state data (its key has no characteristic of continuity, such as the account balance), we designed multi-level caching mechanisms to improve reading/writing performance. Since we focus on improving the underlying storage architecture in the blockchain system, the existing researches can even achieve better performance based on our work. HyperBSA has been integrated into the commercial consortium blockchain platform Hyperchain and support dozens of large-scale commercial blockchain projects to efficiently deal with massive data [11].

The main contributions of this paper are as follows:

1) We creatively divided the blockchain data into continuous data and state data and proposed an efficient storage architecture for these two types of data. The corresponding parts of the underlying storage can be efficiently improved according to different characteristic of the blockchain data.
2) We proposed an index-based storage engine dedicate to continuous data, named Filelog. It saves continuous data as files and adopts a double-layer index mode for fast access. In addition, it uses sparse index and adjusts the step size according to different random-read requirements of storage scenarios.
3) We designed a multi-level cache mechanism for state data based on its characteristic of frequent I/O. Special read/write caches are designed to improve the storage efficiency. Combination with persistence policy, the security and integrity of massive state data can be ensured.
4) We constructed distributed extension schemes for two different types of data respectively to solve the storage extension problem under large data volume.

The remainder of this paper is organized as follows. We firstly review the related work in Section II and present the overview of HyperBSA in Section III. Then, we elaborate the continuous data storage engine—Filelog in Section IV and introduce the multi-level cache mechanism with persistence policy in Section V. After that, we describe the distributed extension of HyperBSA in Section VI. The experimental results are shown in Section VII and a brief discussion of our approach are given in Section VIII.

II. RELATED WORK

In the mainstream blockchain platforms, block data is stored in key-value databases (e.g. LevelDB, RocksDB etc.), and the storage structure is relatively simple. Performance and extension problems occur when there is a large amount of data. Therefore, a series of explorations on the improvement of storage performance and storage expansion of blockchain have been made by researchers to deal with massive data. The related work can be summarized into three aspects: performance improvement by compressing node data, performance improvement by collaborative storage and distributed extension of blockchain storage.

A. PERFORMANCE IMPROVEMENT BY COMPRESSING NODE DATA

Some researchers proposed methods of compressing blockchain data to improve the storage efficiency.

One way to compress data is to modify the structure of the stored data. S. Nakamoto discussed a method of keeping the latest trading records by abandoning other executed transactions [12]. R. Géraud *et al.* gave a solution based on graph algorithm and lattice reduction to compress transaction account ledger [6]. X. Chen proposed a method to replace the hash pointer with index pointer to reduce the storage space required by Bitcoin storage [13]. The Bitcoin storage space can be reduced by 12.71% after adopting it.

Another attempt to compress data is to adopt new encoding technologies. R. K. Raman and L. R. Varshney designed a new encoding method [14], which combined with secret sharing, private key encoding and distributed storage. M. Dai *et al.* proposed NC-DS framework [15] to reduce the data stored in blockchain. A blockchain with rewritable block content is studied by Ateniese *et al.* [16]. This kind of blockchain

[1]Data is stored with a unique key in the blockchain.

can guarantee that any number of blocks can be compressed. Z. Guo introduced an optimization scheme based on the redundant residual number system and fault tolerance mechanism [17], which can dynamically reduce the storage capacity of nodes on the blockchain system.

Although these methods can reduce the data volume of the blockchain, there are two main disadvantages. One is that data compression has the risk of being irreversible, which may cause data on the blockchain being unretainable. The other is that the process of data compression is very complicated, there are many complex rules for communications between blocks to reach sufficient consistency and security, which may highly interfere the working efficiency of the blockchain.

B. PERFORMANCE IMPROVEMENT BY COLLABORATIVE STORAGE

By processing data collaborative storage, the requirement of improving storage efficiency can also be met. Y. Xu introduced the "Section Blockchain" to reduce the amount of storage on each blockchain node [18]. Z. Xu proposed a shared unit in the blockchain system and several nodes in a unit can store the data together, which solves the problem of storing large amount of data in one node [7]. Y. Ren proposed a blockchain-based wireless sensor network. It uses the method of saving hash function [19], which enables the new data to be stored on the node closest to the current data. It only stores different sub-blocks, thus greatly saves the storage space of nodes in the blockchain network. T. Liu proposed a heuristic algorithm named FMA to solve the storage imbalance problem [20]. This solution has a good storage fitness with a small number of nodes. A special erasure code for the blockchain was provided by Perard *et al.* [21], blockchain data was transformed into fragments. The original block data can be recovered by downloading these fragments.

The above researches mainly try to divide the whole blockchain data into slices and store them on different nodes to reduce the storage of a single node. However, they ignored the communication cost between nodes. In other word, it is an exchange of computation time for storage space, which causes additional performance consumption.

C. DISTRIBUTED EXTENSION OF BLOCKCHAIN STORAGE

There are some works focusing on the design of distributed scheme to store massive data. Currently, most blockchain platforms use the key/value database (e.g., LevelDB or RocksDB) to implement their storage architectures [9], [10], [22]. Distributed file systems (e.g., Swarm and IPFS) can be used to extend their storage for massive transaction data. Q. Zheng proposed an IPFS-based blockchain storage architecture [23], which stores the transaction data into IPFS network and packs the IPFS hash of transactions into block. In this way, it reduces the amount of blockchain data effectively. S. Wang proposed Forkbase [8], an efficient storage engine specifically designed for blockchain and pluggable applications, which integrated core blockchain attributes into storage and adopted an index structure to identify as well as eliminate duplicate data.

Although these schemes can solve the blockchain storage extension problems in some ways, they all have a problem that the database is tightly coupled. Their databases employed for the underlying storage are all integrated in the blockchain system, which are not flexible to scale up. In addition, they do not optimize the storage architecture based on the different characteristics of the underlying data.

III. OVERVIEW OF HyperBSA

In order to maximize the storage performance of consortium blockchain, we creatively divided the blockchain data into continuous data and state data. A new extensible storage architecture named HyperBSA is proposed to deal with the two types of data separately, which is shown in Figure 1.

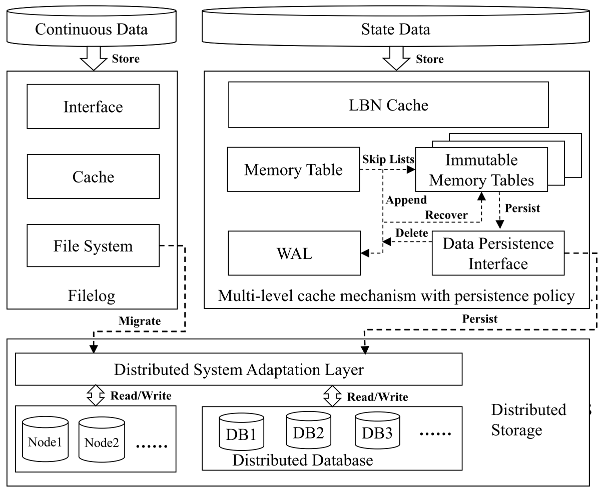


FIGURE 1. The overall design of HyperBSA.

There are mainly three modules in HyperBSA: Filelog to deal with continuous data, MCMPP(multi-level cache mechanism with persistence policy) to deal with state data and distributed extension of the architecure. With the organic integration of the three modules, the performance and scalability of the consortium blockchain storage architecture can be significantly improved.

1) Filelog: Since continuous data has requirements to read/write sequentially, we proposed a new storage engine called Filelog. The main function of Filelog is to cache the continuous data and store it as files, then migrate them to distributed storage asynchronously.
2) MCMPP: Since state data requires reading/writing frequently, we designed a multi-level cache mechanism with persistence policy to improve the performance. This mechanism is to cache state data according to our proposed different caching strategies and implement corresponding complete countermeasures for various abnormal situations.

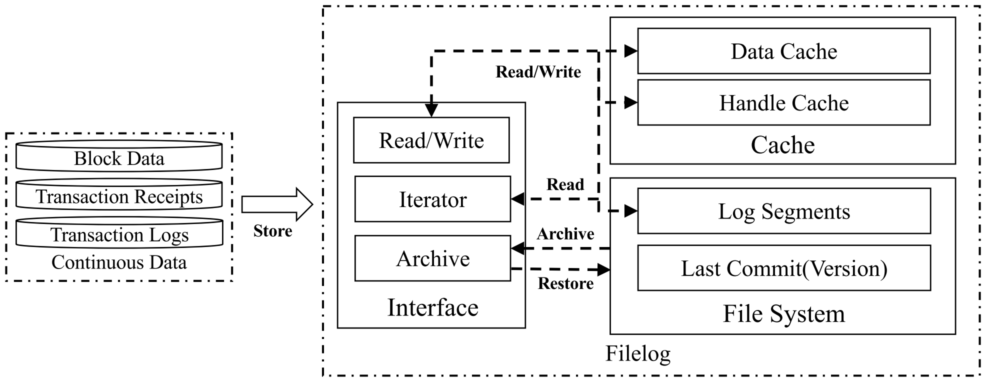


FIGURE 2. The structure of Filelog.

3) Distributed storage: HyperBSA has the ability to implement distributed storage flexibly through DSAL (Distributed System Adaptation Layer) based on Client/Server model.

IV. CONTINUOUS DATA STORAGE: FILELOG

In a consortium blockchain platform, many kinds of data can be treated as continuous data, such as block data, transaction receipts and transaction logs. For these types of data, there is no possibility of modification and deletion. Instead, frequent reading/writing and much storage space are required.

There are several performance issues in traditional storage architecture when storing continuous data, such as performance problem of unbalanced continuous data access and storage performance degradation. Because the traditional storage architectures based on key-value databases (e.g., LevelDB, RocksDB) are too clumsy to meet both input and output requirements, as the performance of writing is high while reading is low. To store massive continuous data, it is necessary to constantly merge data in LevelDB based on the LSM-Tree model [24] and constantly perform compaction in SStable (Sorted String Table), which causes rapid storage performance degradation with a large amount of I/O consumption and write amplification.

To solve the above problems, we designed a new solutions. For unbalanced access performance, according to the continuity of continuous data, we adopted respective cache strategies for different types of data and designed a two-layer index-based mode which can adjust index step for different I/O requirements. To solve the storage performance degradation, we designed one way of archiving data without downtime based on state transition.

We aggregated these solutions and designed a storage engine dedicated to continuous data, named Filelog. The structure of Filelog is composed of three parts: Cache, File System and Interface, shown in Figure 2. The Cache is designed to cache continuous data and file handlers, which can meet frequent I/O requirements, and the details are given in Section IV-A. The File System mainly consists of Log Segments to store continuous data stably, implementation of which is in Section IV-B. The Interface is the implementation of I/O and archiving/restoring of continuous data, which is introduced in Section IV-C. The interaction between the Cache and the File System is through The Interface.

A. CACHE: LOGICAL CACHE STRUCTURE

The Cache shown in Figure 2 is divided into two parts, Data Cache and Handle Cache. For frequent reading/writing, we adopted two different cache strategies for them. The Data Cache holds the most recently written blocks, receipts and logs. Because the data stored in blockchain is constantly accumulating, the probability of new blocks being queried is high. Therefore, the FIFO algorithm is employed in the data cache. The Handle Cache holds the file handles in Log Segments of the File System. It implements ARC (Adaptive Replacement Cache) algorithm, which is one of the best cache algorithms that adjusts the cache elimination strategy according to file handle and increases cache hit ratio. The file handle takes the form of key/value pairs. If hit, the file handle is taken directly in the cache, or taken from the Log Segments and added into the cache.

B. FILE SYSTEM: CUSTOMIZED INDEX-BASED FILE SYSTEM

The file system contains Last Commit and Log Segments. Last Commit is an important file which is used to save the last submitted record while ensuring the atomic storage. Only after the version being updated can the store process be completed. The principle of Log Segments is a two-layer index-based mode. Each of them is segmented by the size of the file (500MB, in default) to store continuous data.

The structure of Log Segments is shown in Figure 3. Both two layers are based on the key of continuous data. In the first layer, we set the key of the first continuous data as the name of Log Segment. In the second layer, we designed two files in the Log Segment, an index file and a log file. The index file is divided into two parts, Offset and Position, which is the key and relative position of continuous data in the log file

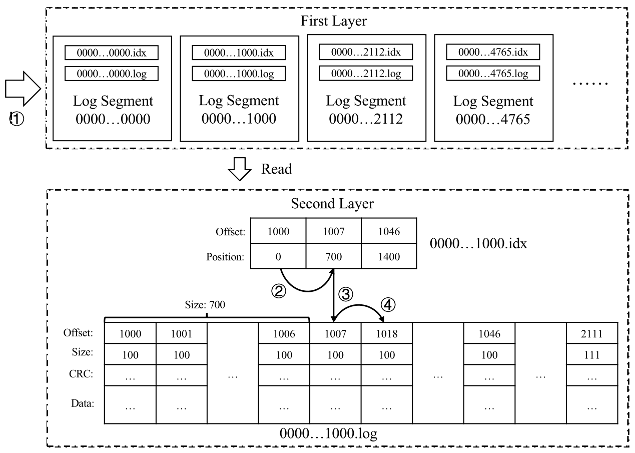


FIGURE 3. The structure of Log Segments.

respectively. The choice of Offset is related to the sparse step according different data sizes, which can greatly reduce the data volume. For storing continuous data, the log file includes four parts: Offset, Size, CRC (Cyclic Redundancy Check) and Data, where Size is the size of current data and used for iterator.

Let's take the example of searching data of key 1018. The searching operation is shown as Step 1, 2, 3 and 4 in Figure 3, where Step 1 is in the first layer and the rest steps are in the second layer. In Step 1, we can find the data is in Log Segment 0000…1000, where 1018 is greater than 1000 and less than 2112. In Step 2, we get the index file, 0000…1000.idx in Log Segment 000…1000. According to the sparse index, we designed a binary search algorithm to find the Position of the entity with Offset 1018, where 1018 is greater than 1000 and less than 1046. And the value of Position is 700. Then in Step 3, we can locate the respective position in the log file according to 700. In Step 4, we start traversal from this Position until hitting the Offset 1018.

With the design of our file system, the computing time is short. In particular, with the assumption that getting the value of the Offset is treated as the computing unit, the average computing time is as described in formula(1), where L is the number of Log Segment, N is the number of Offset in the index file, and M is the number of Offset in the log file.

$$average\_computing\_time = L + log_2 N + M/N \qquad (1)$$

C. INTERFACE: IMPLEMENTATION OF I/O AND ARCHIVING INTERFACE

In the Interface part, there are three modules: Read/Write, Iterator and Archiving. The Read/Write is the major module for general random I/O and interaction between the Cache and the File System. The Iterator is for fast index in the Log Segment. The Archiving is designed to archive/restore files in the Log Segments to reduce continuous data storage.

Both the Read/Write module and the Iterator module are designed for data reading/writing. The difference between them is that the data reading/writing in the Read/Write module is random and the Iterator is sequential. There are two functions in the Read/Write module. One is for general random I/O, the other is the processing of continuous data and file handle in the Cache and the File System. Due to the two-layer index-based mode of Log Segments, general I/O can be implemented by the Offset in the index and log files. The processing of continuous data and file handle is mainly for reading. Firstly, the Read/Write module searches data in the Data Cache. If data can't be found, it will need to search the Log Segment in which the data stores. Secondly, because there is one file handle for each Log Segment, the Read/Write module searches the file handle of the target Log Segment in the Handle Cache. If the handle isn't in this cache, the module will only search in the Log Segments. Finally, the Read/Write module gets the data and put corresponding data and file handle into the Cache. Because of the continuity feature of continuous data, it's necessary to have the function of fast sequence reading. Thus, we designed the Iterator based on the Size in the log file. It calculates according to the value of the Size and implements quick index in sequence.

The Archiving module is one of our major innovations. Because of the particularity of the blockchain structure, nodes on the blockchain need to complete archiving without downtime. Meanwhile, we found that file archiving involves many different types of files. Only when it ensures the atomicity of the archiving process can we meet the requirements. For this, we designed two intermediate states of the file. One state of file saves the contents of the target file, and the other state of file guarantees that the entire amount of data in the target file can be recovered whatever happens at any stage. It should be noted that two abnormal situations occur during archiving process. When the writing of archived meta files are not completed and all intermediate state files are deleted, the process needs to be initialized again. Instead, when archived meta files exist and the deleting of archived meta files are pending, it deletes all intermediate files, modifies the status of archived meta files and completes this archiving process after restarting the engine.

The data recovery process is the reverse of data archiving. Therefore, all data in the offline database must be moved to the online database during the recovery process. In the end of restoring, Filelog needs to be restarted to restore new Log Segments to the storage engine, which enables that data can be read successfully.

V. MULTI-LEVEL CACHE MECHANISM WITH PERSISTENCE POLICY FOR STATE DATA

State data, which also named World State, is stored in the account/balance model of consortium blockchain. It is in the form of key/value, mainly includes the account status in all blockchains and some related data of Merkle Root [25]. It's extremely frequent to read/write state data in the blockchain, but the traditional storage architecture of blockchain can't meet the requirement of reading/writing massive state data as it is suitable for the situation of writing more and reading less.

To handle the above issue, we designed a multi-level cache mechanism with a persistence policy to achieve the purpose of reading more and writing more. The multi-level cache mechanism uses customized methods for refreshing cache to improve the hit ratio of data reading, which is divided into LBN (Latest Block Number) cache mainly for reading state data and memory cache mainly for writing. The persistence policy ensures that massive state data can be persisted to distributed database. The structure of MCMPP is shown in Figure 4, which is composed of three parts:

1) LBN cache: Mainly for frequent state data reading. It is the special designed cache for state data reading, which can improve the read rate. In addition, the customized strategy in LBN cache can control the occupation of memory space.
2) Memory cache: Mainly for saving massive state data. The Memory DB, actual write cache, is its main module. It can be divided into MT (Memory Table) and multiple IMT (Immutable Memory Tables). Based on Skip Lists, the MT can be converted to the IMT and related logs will be appended in the WAL (Write-Ahead Logging) [26] in order to recover the IMT.
3) Persistence policy: The WAL can guarantee the security of state data, which will be stored in Filelog. The DPI (Data Persistence Interface) is the actual writing component to perform persistence policy, through which the integrity of state data can be ensured when persisting and the logs in the WAL will be deleted after persisting.

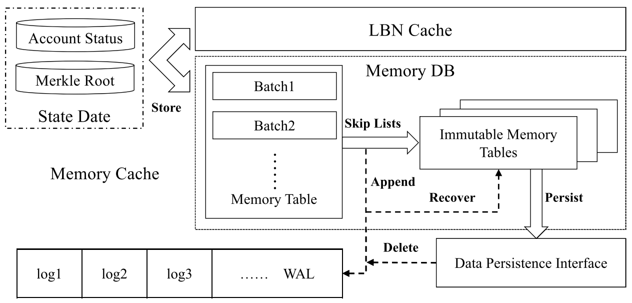


FIGURE 4. The structure of multi-level cache mechanism with persistence policy.

A. LBN CACHE FOR STATE DATA

As the special-purpose read cache for state data, LBN cache supports frequent reading of state data, which can improve reading performance and guarantee the reasonable management of memory space. Since the block data has the characteristics of continuous accumulation, new blocks are more likely to be queried. The LBN cache is designed to replace the LRU (Least Recently Used) [27] cache which is used for the storage of state data and Bucket-tree data in traditional consortium blockchain platforms. In addition, LRU may lead to the loss of writing because it is possible that the data in the cache is replaced while not being persisted and LRU may suffer from the problem of continuous increasing of occupied memory space when the data volume is increasing.

To overcome the disadvantages of LRU, we designed the LBN cache which can expand/shrink memory space on demand through HashMap and replace the state data according to the customized policy through Cache Manager. It adopts a two-layer doubly-linked list to implement LBN replacement policy, as shown in the Figure 5. The first layer is a doubly-linked list composed of block number, and the second layer is a doubly-linked list composed of state data that has inserted into LBN cache. The state data is storing in the bucket of HashMap and can be replaced through Cache Manager.

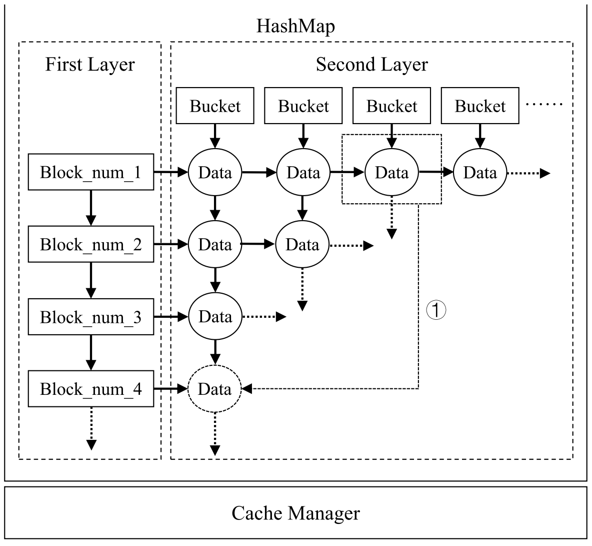


FIGURE 5. The structure of LBN cache.

The HashMap is used to store specific state data and perform implicit scaling control when writing. We initialize three variables to perform the scaling control: the quantity of initialized buckets N, the maximum value of data items in each bucket M and the maximum overflowed value of data items in each bucket $M*$. In addition, the cache expansion threshold can be initialized to $G = N * M$ and the cache shrink threshold S can be initialized to 0. When the total value of actual data items in the cache is greater than G, or their overflowed value from the hit bucket is greater than $M*$, N and G should be doubled, S should be reduced to half of N. When the total number of data items in the cache is less than S, N and G should be reduced to half of original number, S is set to half of N.

The Cache Manager is used to manage the life cycle of state data in the HashMap. According to the replacement policy, when the capacity reaches the threshold, partial state data in the LBN cache whose block number with the largest difference from the latest inserted block number will be replaced to restrict the occupation of memory space. If new state data is inserted, it will be directly added to the secondary-linked list with the corresponding block number. When the state data stored in the cache are updated with the new block number, they will be moved as Step 1 shown in Figure 5.

According to our design, the read operation is relatively simple, that is to search according to the time sequence after data processing. In general, begin with LBN cache, if the data we need has been replaced in the cache, Memory DB and distributed database should be read in turn.

B. MEMORY CACHE FOR STATE DATA

As the write cache for state data, it's mainly responsible for writing state data to Memory DB. The Memory DB includes MT and multiple IMT. Batch slice is used to integrate single write operation in this storage structure. We adopt MVCC (Multi-version Concurrency Control) [28] mode for concurrent read/write control and support atomic write of the Batch as well as key/value write.

In order to persist data into a distributed database, the following steps should be performed. Firstly, the MT will convert to the IMT based on Skip Lists when the state data is not written to the MT within the stipulated time, or the space of the MT is insufficient, or the amount of Batch in the MT reaches the threshold value. Only the key with the same ID but the largest sequence will be retained during the implementation of Skip Lists. Secondly, a log will be appended in the WAL to ensure state data security of the IMT. Finally, the state date in the IMT can be persisted through DPI.

As system crash is inevitable, it is needed to ensure the integrity of state data in memory cache. To handle this, continuous data will be persisted firstly, then write all state data to the multi-level cache atomically. If the storing process of state data fails, system will quickly roll back the block data in continuous data to keep consistency between continuous data and state data. But system collapse will also lead to state data loss in the IMT. For this, we can query the integrated WAL, then recover all the IMT in chronological order. It's the reason that we append the logs to WAL after Skip Lists.

C. PERSISTENCE POLICY

Sate data in the IMT will be persisted to the distributed database asynchronously. Since all the keys in the IMT are unique, we use Goroutine Pool to read from the same IMT continuously. The standard DPI writes data to the distributed database in a multi-threaded way. It should be noted that the persistence of different IMT must be executed serially to ensure that the latest state data will be written to the database. When finished, the corresponding logs in the WAL will be deleted. But the IMT persisted successfully don't need to be deleted immediately when the upper memory limit of multi-level cache does not reach the threshold, so as to improve the reading efficiency.

In multi-level cache, the logs in WAL are also continuous, so we store them in the Filelog. As WAL does not require random read, we adjust the sparsity of sparse index in the Filelog to the maximum in view of the storage efficiency. In addition, since the database supports idempotent storage, when the system goes down after persisting state data successfully. But the logs in WAL is not deleted, it is still considered that the IMT have not completed all the processes of persistence. In this case, it is necessary to recover the IMT and re-execute all the persistent operations of them.

VI. THE DISTRIBUTED EXTENSION OF HyperBSA

Although the archive function of Filelog can solve massive continuous data storage to some extent, the archived files cannot be searched in real time. It is urgent to extend the underlying database in a distributed way, yet the design of traditional consortium blockchain storage architecture based on the embedded database model is tightly coupled, which can't suffice for flexible distributed storage extension.

With consideration of that, we proposed a pluggable database mode based on Client/Server model to connect different distributed systems flexibly. For continuous data and state data, we set DSAL (Distributed System Adaptation Layer) to connect the distributed storage systems. As shown in Figure 6, DSAL is mainly composed of 4 parts: Adaptation Manager, Filelog Manager, MCMPP Manager and Client Manager. The Adaptation Manager is the pivot of DSAL, which controls data and client adaption. The Filelog Manager is for the migration configuration of files in Filelog. The MCMPP manager is for the persistence configuration accordingly. The Client Manager manages the clients of various distributed storage systems.

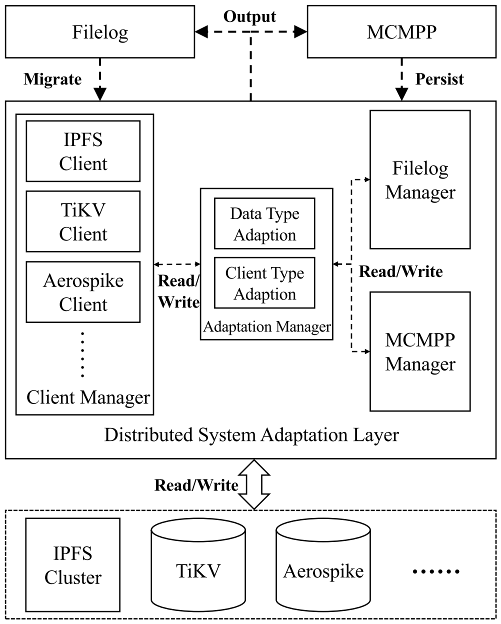


FIGURE 6. The structure of distributed system adaptation layer.

A. ADAPTATION MANAGER

The Adaptation Manager is the critical module designed to control the interaction process for the distributed extension. It consists of DTA (Data Type Adaptation) and

CTA (Client Type Adaptation). The DTA determines the type of data that needs to be stored in the distributed system, so as to perform corresponding operation, such as migration or persistence. It also needs to interact with Filelog Manager and MCMPP Manager to obtain or update relevant information of migration and persistence. And the function of CTA is to select the client of the respective distributed system in DTA according to data type and relevant information.

B. FILELOG MANAGER

The Filelog Manager is designed to set the configuration of Filelog's migration. Since continuous data exists in Filelog in the form of file, we migrate these files asynchronously to the distributed file system. Filelog Manager ensures the I/O performance and data integrity by managing configuration information. Two factors need to pay greatly attention in this migration. One is the maximum capacity in Filelog, the other is the file migration time.

The maximum capability indicates the threshold of starting migration. If the capability is set too small, it will cause frequent migration, resulting in I/O performance reduction. Conversely, the continuous data volume in single node will be too large if it is high, which interferes the storage performance of single node. The file migration time is how long it takes for a Log Segment to be migrated after no space is left. By setting the appropriate time, the system can guarantee efficient asynchronous migration without affecting data I/O operations.

C. MCMPP MANAGER

The MCMPP Manager is designed to set the configuration of MCMPP's persistence. For the distributed extension of state data, we designed the MCMPP Manager. It manages relevant information during the persistence process. Different from Filelog Manager, the persistence process of state data is real-time according to the actual business requirements. Thus, there are also two factors, time record and data limitation. The time record is the average time of finishing persistence which quantifies the storage performance of state data. And the data limitation is to limit the capacity of state data to avoid storage performance reduction when state data increases.

D. CLIENT MANAGER

The Client Manager is designed to help the Adaptation Manager to interact with the distributed storage systems. To integrate more distributed databases, we can just set the corresponding clients and add them into the Client Manager. It's convenient to take away one database by removing its client. Thus, adding this manager is easy for the distributed extension to achieve one swift and economical pluggable mode.

For Filelog, we selected the distributed file system IPFS, which consists of several IPFS nodes (storage nodes). When files are stored in the IPFS Cluster, the unique hash value can be calculated, which is easy to search. And plugging the IPFS Cluster can extend file storage while achieving real-time search.

For MCMPP, we connected two typical distributed databases (TiKV and Aerospike). By converting into the Client/Server model, we set the client for TiKV, which achieves seamless connection with Put, Get, Delete and Scan operations of TiKV. And we do the same with Aerospike.

VII. EXPERIMENTAL RESULTS AND DISCUSSION

In this section, several quantitative experiments are given to evaluate the performance of HyperBSA. The experiments were carried out for the continuous data and the state data respectively. Since the main work of this paper is a new consortium blockchain storage architecture, the quantitative experiments compared the performance of HyperBSA and the mainstream consortium blockchain storage engine based on LevelDB or distributed database TiKV. All the experiments were done on an in-house blockchain cluster with 4 machines. Each machine runs CentOS Linux release 7.3.1611 (Core) and is equipped with Intel (R) Core (TM) i7-7700 CPU @ 3.60GHz, 16GB memory. In order to show the ability of our storage architecture to process massive data, the data volume in the experiments for continuous data is very huge and the data size in the experiments for state data is very big. In order to minimize the test errors, we did each experiment 10 times and reported the average query and running time.

A. EXPERIMENTS FOR CONTINUOUS DATA

The experiments in this part evaluate the performance of our continuous data storage engine Filelog. We mainly compared the performance of our Filelog with the LevelDB-based storage engine, which is widely used in the mainstream consortium blockchain platforms. As we know, the traditional blockchain storage architectures simply store the block data in key/value databases such as LevelDB, regardless of data type. Therefore, we just need to compare the I/O time-consuming between our Filelog and LevelDB. In this way, we intuitively show the advantages of using Filelog to store continuous data.

According to actual data storage in enterprise-level business scenarios, we conducted several experiments for LevelDB and Filelog in the case that data volume is 64G, 128G, 256G and 512G respectively. In each case, we set single data size to 200B, 20K and 1M respectively. The results are given in Table 1. To show the experimental results more intuitively, we compared the read/write time-consuming of LevelDB and Filelog in Figure 7 and Figure 8 respectively.

It can be seen that, the storage performance improves significantly after adopting our Filelog. In comparison with LevelDB, the time-consuming of Filelog reduces by 81.85%/82.47% on average for reading/writing continuous data. Filelog performs better when the data volume is huge. With the increasing amount of data, the advantages are more prominent. In addition, Filelog has better stability than LevelDB. There is a huge gap between Filelog and LevelDB in stability of experimental results. The standard deviation

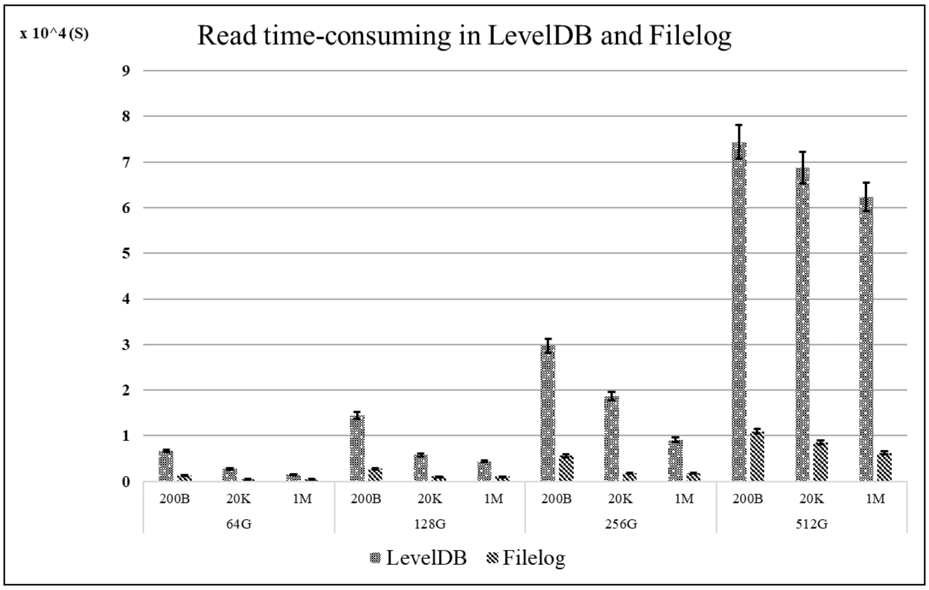


FIGURE 7. Read time-consuming in LevelDB and Filelog.

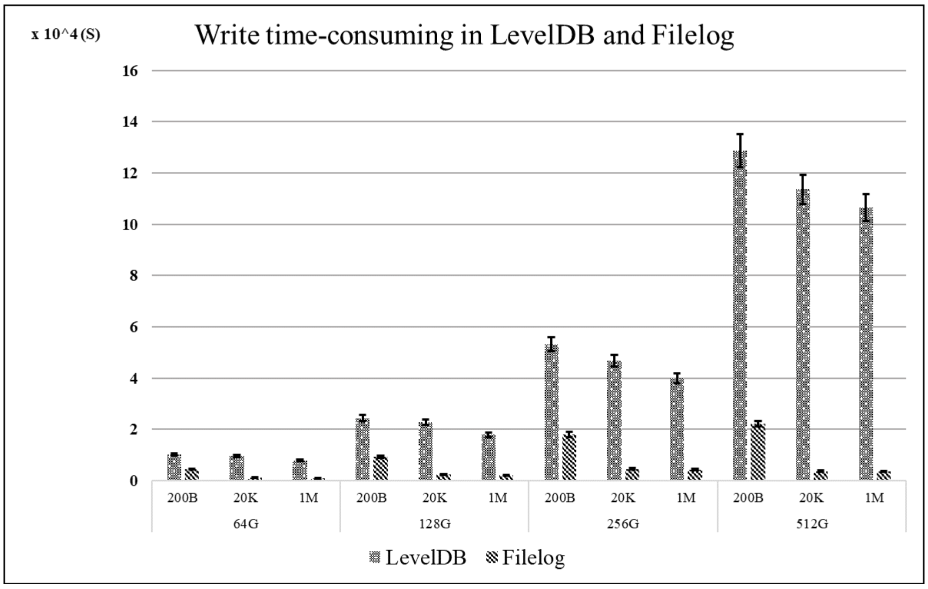


FIGURE 8. Write time-consuming in LevelDB and Filelog.

TABLE 1. Write and read time-consuming in LevelDB and Filelog.

data volume	single data size	read time(s)		write time(s)	
		LevelDB	Filelog	LevelDB	Filelog
64G	200B	6689.3	1341.4	10189.4	4573.6
	20K	2744.1	499.2	9635.9	1195.7
	1M	1479.7	487.5	7938.5	1051.6
128G	200B	14469.7	2741.8	24401.2	9250.5
	20K	5773	1063.5	22871	2392.1
	1M	4384.1	1055.3	17911.1	2250.4
256G	200B	29735.3	5600.2	53238.3	18036.1
	20K	18688	1814.8	46862.3	4664.1
	1M	9101.4	1762.5	39871.8	4385.9
512G	200B	74384.6	10950.3	128736.9	22297
	20K	68762.5	8588.3	113642.8	3761
	1M	62341.4	6234.7	106549.2	3664.5

of Filelog is 702.07s, while that in LevelDB is 13045.88s, when the single data size is 1M. When the total amount of data reaches a certain threshold, the storage performance of LevelDB degrades drastically. However, Filelog will not suffer from performance instability when the data volume changes. Therefore, we can draw the conclusion that Filelog has much better performance for the storage of continuous data.

B. EXPERIMENTS FOR STATE DATA

Since our MCMPP is specially designed to optimize the storage of state data, the experiments in this part evaluate the performance of MCMPP for state data. Under the traditional consortium blockchain storage architecture, the state data is stored directly in the persistence engine (e.g. LevelDB or TiKV). While we use MCMPP to reduce the read/write delay.

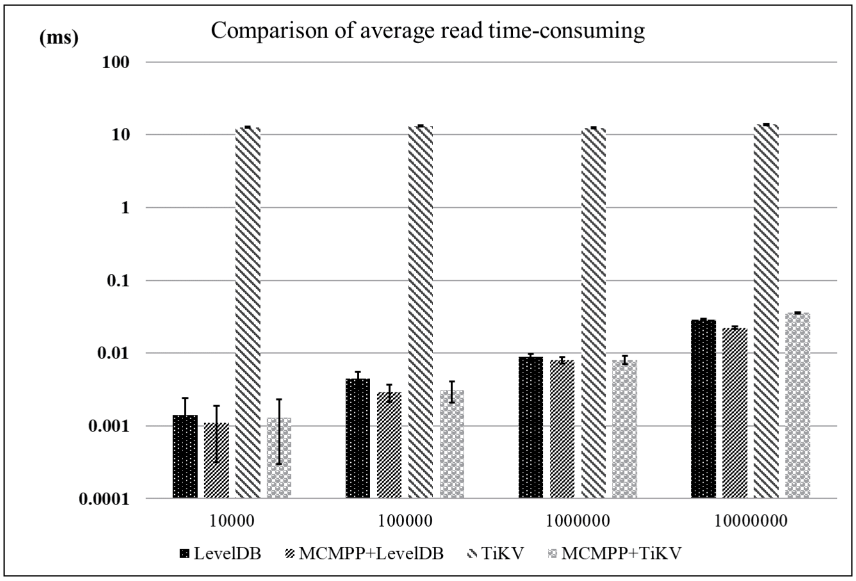


FIGURE 9. Comparison of average read time-consuming.

We compared the storage performance of our MCMPP and the traditional storage mode based on LevelDB or TiKV.

According to actual data storage in enterprise-level business scenarios, we conducted several comparative experiments in the case that the number of data record is 10000, 100000, 1000000 and 10000000 respectively.The results are shown in Table 2 and Table 3.

TABLE 2. Comparison of read time-consuming.

Storage method / Data size	LevelDB(s)	MCMPP + LevelDB(s)	TiKV(s)	MCMPP + TiKV(s)
10000	0.014	0.011	127	0.013
100000	0.45	0.29	1322	0.31
1000000	8.83	7.98	12450	8.11
10000000	287.63	223.67	138742	356.78

TABLE 3. Comparison of write time-consuming.

Storage method / Data size	LevelDB(s)	MCMPP + LevelDB(s)	TiKV(s)	MCMPP + TiKV(s)
10000	0.043	0.021	165	0.022
100000	0.48	0.3	1564	0.36
1000000	8.84	4.42	13540	4.68
10000000	122.28	52.2	128762	82.68

It can be seen that, MCMPP improves the read/write performance of state data greatly compared with the storage structure of traditional consortium blokchain directly based on LevelDB or TiKV. In addition, as the amount of data increases largely, the performance improves significantly. Especially in the case of using distributed database TiKV, the existence of network consumption will greatly increase the delay of data access. Our MCMPP solve this problem very effectively, which lays a solid foundation for the distributed extension of blockchain storage. To show the experimental results more intuitively, we compared the average read/write time-consuming of a piece of data for the storage modes in Figure 9 and Figure 10 respectively.

As can be seen from the figures, MCMPP improves the storage performance of state data to a certain extent. Compared with LevelDB, the average time-consuming of MCMPP reduces by 22.21%/48.99% for reading/writing state data. Compared with the traditional distributed databases TiKV, the average time-consuming of MCMPP reduces by 99.91%/99.97% for reading/writing state data. Therefore, We can draw the conclusion that MCMPP has much better performance for the storage of state data.

C. FURTHER DISCUSSION

By the above comparative experiments, it can be seen that our storage architecture has much better performance than that of the existing consortium blockchain platforms. The performance improvement is achieved by optimizing the storage mechanism for data of different characteristics. Our storage architecture needs some additional memory overhead, but the overhead is completely acceptable. The memory space occupied by the cache can be controlled by parameter configuration. The storage architecture will never suffer from memory overflow. In Filelog, the default size of data cache is set to 100M. In MCMPP, the default size of the multi-level cache is set to 500M, among which the LBN Cache and

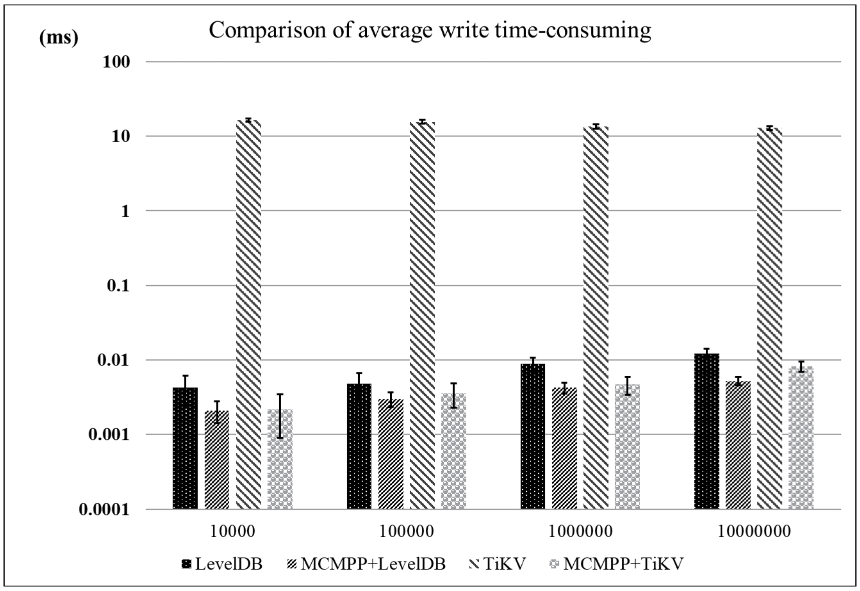


FIGURE 10. **Comparison of average write time-consuming.**

Memory Cache are set to 100M and 400M respectively. The above experiments were done with such default configuration parameters. In actual applications, users can configure the size of the memory space occupied by the cache according to business requirements, server performance, etc.

The high-performance storage architecture proposed in this paper has been integrated into the well-known consortium blockchain platform Hyperchain, which has supported the efficient and stable running of dozens of actual blockchain projects with massive data from many large financial institutions (e.g., China UnionPay, Industrial and Commercial Bank of China, Agricultural Bank of China, Bank of China, China Construction Bank, etc.). With the support of this high-performance storage architecture, Hyperchain ranks first in performance indicators in several authoritative third-party technical evaluations [29].

VIII. CONCLUSION AND FUTURE WORK

In order to break through the storage performance bottleneck of the existing consortium blockchain platforms, we designed and implemented a new storage architecture named HyperBSA, which deals with the continuous data and state data of blockchain separately. The continuous data storage engine Filelog and the multi-level cache for the state data are proposed. To improve the scalability of blockchain storage, we adopted the pluggable scheme for the corresponding distributed storage. Experimental results show that our design has greatly improved the I/O performance. Our storage architecture has been integrated into the commercial consortium blockchain platform Hyperchain and support dozens of large-scale commercial blockchain projects with massive data.

In the long term, there are some points to be improved in the design and implementation of this paper, which are our main works in the future. First of all, we will design one more reasonable method for file segmentation and movement which may affect the normal reading of Filelog in the archiving process. Secondly, to implement overall distributed extension, the distributed cache mechanism will be added. Finally, we will append more distributed storage systems, such as SeaweedFS for Filelog, OceanBase and TiDB for multi-level cache.

REFERENCES

[1] C.-D. Lee, B.-J. Choi, and K.-S. Park, "Design and evaluation of a block encryption algorithm using dynamic-key mechanism," *Future Gener. Comput. Syst.*, vol. 20, no. 2, pp. 327–338, Feb. 2004.

[2] G. Yoshihiro, A. Shingo, and M. Masayuki, "Methods on logical network construction in peer-to-peer services based on traffic measurements," Inst. Image Inf. Telev. Eng., ITE Tech. Rep., 2002, pp. 43–48, vol. 26, no. 13.

[3] T. Song and Y. Zhao, "Comparison of blockchain consensus algorithm," *Comput. Appl. Softw.*, vol. 35, no. 8, pp. 1–8, Aug. 2018.

[4] X. Liu, X. Du, N. Wang, and S. Li, "Research progress of blockchain technology and its application in information security," *J. Softw.*, vol. 29, no. 7, pp. 2092–2115, 2018.

[5] Y. Yuan and F.-Y. Wang, "Blockchain: The state of the art and future trends," *Acta Autom. Sinica*, vol. 42, no. 4, pp. 481–494, 2016.

[6] R. Géraud, D. Naccache, and R. Roşie, "Twisting lattice and graph techniques to compress transactional ledgers," in *Proc. 13th Int. Conf. Secur. Privacy Commun. Netw.* Cham, Switzerland: Springer, 2017, pp. 108–127.

[7] Z. Xu, S. Han, and L. Chen, "CUB, a consensus unit-based storage scheme for blockchain system," in *Proc. IEEE 34th Int. Conf. Data Eng. (ICDE)*, Apr. 2018, pp. 173–184.

[8] S. Wang, T. T. A. Dinh, Q. Lin, Z. Xie, M. Zhang, Q. Cai, G. Chen, B. C. Ooi, and P. Ruan, "Forkbase: An efficient storage engine for blockchain and forkable applications," *Proc. VLDB Endowment*, vol. 11, no. 10, pp. 1137–1150, Jun. 2018.

[9] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," Ethereum Project Yellow Paper, 2014, pp. 1–32, vol. 151, no. 1.

[10] C. Cachin, "Architecture of the hyperledger blockchain fabric," in *Proc. Workshop Distrib. Cryptocurrencies Consensus Ledgers*, Ruschlikon, Switzerland, 2016, pp. 310–312.

[11] *The Commercial Consortium Blockchain Platform Hyperchain*. Accessed: Aug. 8, 2020. [Online]. Available: https://www.hyperchain.cn/

[12] S. Nakamoto. (Oct. 2008). *Bitcoin: A Peer-to-Peer Electronic Cash System*. Accessed: Nov. 2018. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[13] X. Chen, S. Lin, and N. Yu, "Bitcoin blockchain compression algorithm for blank node synchronization," in *Proc. 11th Int. Conf. Wireless Commun. Signal Process. (WCSP)*, Xi'an, China, Oct. 2019, pp. 1–6.

[14] R. K. Raman and L. R. Varshney, "Dynamic distributed storage for scaling blockchains," 2017, *arXiv:1711.07617*. [Online]. Available: http://arxiv.org/abs/1711.07617

[15] M. Dai, S. Zhang, H. Wang, and S. Jin, "A low storage room requirement framework for distributed ledger in blockchain," *IEEE Access*, vol. 6, pp. 22970–22975, 2018.

[16] G. Ateniese, B. Magri, D. Venturi, and E. Andrade, "Redactable blockchain-or-rewriting history in bitcoin and friends," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&P)*, Paris, France, Apr. 2017, pp. 111–126.

[17] Z. Guo, Z. Gao, H. Mei, M. Zhao, and J. Yang, "Design and optimization for storage mechanism of the public blockchain based on redundant residual number system," *IEEE Access*, vol. 7, pp. 98546–98554, 2019.

[18] Y. Xu, "Section-blockchain: A storage reduced blockchain protocol, the foundation of an autotrophic decentralized storage architecture," in *Proc. 23rd Int. Conf. Eng. Complex Comput. Syst. (ICECCS)*, Dec. 2018, pp. 115–125.

[19] Y. Ren, Y. Liu, S. Ji, A. K. Sangaiah, and J. Wang, "Incentive mechanism of data storage based on blockchain for wireless sensor networks," *Mobile Inf. Syst.*, vol. 2018, pp. 1–10, Aug. 2018.

[20] T. Liu, J. Wu, J. Li, and J. Li, "Secure and balanced scheme for non-local data storage in blockchain network," in *Proc. IEEE 21st Int. Conf. High Perform. Comput. Commun., IEEE 17th Int. Conf. Smart City, IEEE 5th Int. Conf. Data Sci. Syst. (HPCC/SmartCity/DSS)*, Zhangjiajie, China, Aug. 2019, pp. 2424–2427.

[21] D. Perard, J. Lacan, Y. Bachy, and J. Detchart, "Erasure code-based low storage blockchain node," in *Proc. IEEE Int. Conf. Internet Things (iThings), IEEE Green Comput. Commun. (GreenCom), IEEE Cyber, Phys. Social Comput. (CPSCom), IEEE Smart Data (SmartData)*, Jul. 2018, pp. 1622–1627.

[22] E. Androulaki *et al.*, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proc. 13th EuroSys Conf.*, Apr. 2018, pp. 1–15.

[23] Q. Zheng, Y. Li, P. Chen, and X. Dong, "An innovative IPFS-based storage model for blockchain," in *Proc. IEEE/WIC/ACM Int. Conf. Web Intell. (WI)*, Dec. 2018, pp. 704–708.

[24] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree," *Acta Inf.*, vol. 33, no. 4, pp. 351–385, 1996.

[25] M. Szydlo, "Merkle tree traversal in log space and time," in *Proc. Int. Conf. Theory Appl. Cryptograph. Techn.*, Interlaken, Switzerland, 2004, pp. 541–554.

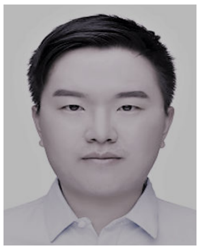
[26] D. Gawlick, J. Gray, W. Limura, and R. L. Obermarck, "Method and Apparatus for logging journal data using a log write ahead data set," U.S. Patent 4 507 751, Mar. 26, 1985.

[27] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," *ACM SIGMOD Rec.*, vol. 22, no. 2, pp. 297–306, Jun. 1993.

[28] P. A. Bernstein and N. Goodman, "Multi version concurrency control—Theory and algorithms," *ACM Trans. Database Syst.*, vol. 8, no. 4, pp. 465–483, 1983.

[29] *The Maturity Evaluation Report of Typical Blockchain Platforms in China*. Accessed: Aug. 16, 2020. [Online]. Available: https://mp.weixin.qq.com/s/DfMYvnc7NU0axfEDsWhDuw

XIAO CHEN received the B.S. degree in software engineering from East China Normal University, Shanghai, China, in 2011, and the Ph.D. degree in computer architecture from the Institute of Computing Technology, Chinese Academy of Sciences, China, in 2017. She is currently an Engineer with CFETS Information Technology (Shanghai) Company Ltd. Her research interests include distributed ledger technology, distributed database, and incentive mechanism in distributed corporation applications.



KEJIE ZHANG received the B.S. degree in computer science and technology from the Zhejiang University of Science and Technology, Hangzhou, China, in 2017, and the M.S. degree in software engineering from Zhejiang University, Hangzhou, in 2019. He joined Hangzhou Qulian Technology Ltd., in 2016, as a Senior Manager. His research interests include blockchain technology, distributed storage, cryptography technology, and cloud computing.

XIUBO LIANG received the B.S. and Ph.D. degrees in computer science and technology from Zhejiang University, Hangzhou, China, in 2006 and 2011, respectively. He is currently an Associate Professor with the College of Software Technology, Zhejiang University. His research interests include blockchain, intelligent information processing, big data, and mobile internet.

WEIWEI QIU received the Ph.D. degree in computer science and technology from Zhejiang University, Hangzhou, China, in 2015. From 2015 to 2018, she was a Postdoctoral Fellow with the College of Computer Science and Technology, Zhejiang University. She joined Hangzhou Qulian Technology Ltd., in 2016. She was a Co-Founder and a Senior Vice President. Her research interests include core algorithms of blockchain, software reliability, software engineering, cloud computing, and services computing.

ZHIGANG ZHANG received the B.S. and M.S. degrees in computer engineering from Nanjing Normal University, Nanjing, China, in 2011 and 2014, respectively, and the Ph.D. degree in software engineering from East China Normal University, Shanghai, China, in 2018. Since 2018, he has been with CFETS Information Technology (Shanghai) Company Ltd. His research interests include distributed database and time series data mining.

DING TU received the B.S. degree in software engineering and the Ph.D. degree in computer science and technology from Zhejiang University, Hangzhou, China, in 2009 and 2016, respectively. Since 2016, he has been with CFETS Information Technology (Shanghai) Company Ltd. His current research interests include distributed database and artificial intelligence.