future internet

MDPI

Article

SASLedger: A Secured, Accelerated Scalable Storage Solution for Distributed Ledger Systems

Haoli Sun [1,*], Bingfeng Pi [1], Jun Sun [2], Takeshi Miyamae [3] and Masanobu Morinaga [3]

[1] Fujitsu R&D Center Co., Ltd., Suzhou 215123, China; winter.pi@fujitsu.com
[2] Fujitsu R&D Center Co., Ltd., Beijing 100022, China; sunjun@fujitsu.com
[3] Fujitsu Limited, Kawasaki 211-8588, Japan; miyamae.takeshi@fujitsu.com (T.M.); morinaga@fujitsu.com (M.M.)
* Correspondence: sunhaoli@fujitsu.com; Tel.: +86-512-62925255

Abstract: Blockchain technology provides a "tamper-proof distributed ledger" for its users. Typically, to ensure the integrity and immutability of the transaction data, each node in a blockchain network retains a full copy of the ledger; however, this characteristic imposes an increasing storage burden upon each node with the accumulation of data. In this paper, an off-chain solution is introduced to relieve the storage burden of blockchain nodes while ensuring the integrity of the off-chain data. In our solution, an off-chain remote DB server stores the fully replicated data while the nodes only store the commitments of the data to verify whether the off-chain data are tampered with. To minimize the influence on performance, the nodes will store data locally at first and transfer it to the remote DB server when otherwise idle. Our solution also supports accessing all historical data for newly joined nodes through a snapshot mechanism. The solution is implemented based on the Hyperledger Fabric (HLF). Experiments show that our solution reduces the block data for blockchain nodes by 93.3% compared to the original HLF and that our advanced solution enhances the TPS by 9.6% compared to our primary solution.

Keywords: blockchain; scalability; storage; data integrity; performance

Citation: Sun, H.; Pi, B.; Sun, J.; Miyamae, T.; Morinaga, M. SASLedger: A Secured, Accelerated Scalable Storage Solution for Distributed Ledger Systems. *Future Internet* **2021**, *13*, 310. https://doi.org/10.3390/fi13120310

Academic Editors: Ahad Zare Ravasan, Taha Mansouri, Michal Krčál and Saeed Rouhani

Received: 5 November 2021
Accepted: 29 November 2021
Published: 30 November 2021

1. Introduction

Blockchain systems are categorized into permissionless blockchains (a.k.a. a public blockchain) and permissioned blockchains (a.k.a. a consortium/private blockchain). In a permissionless blockchain system, any computer or user that can access the blockchain network can join it or quit at will. The nodes and users of the permissionless blockchain networks are identified by their public keys. Cryptocurrencies such as Bitcoin [1], Ethereum [2], and EOS [3] are constructed as permissionless blockchain systems. Permissioned blockchains are always used for information-sharing among several stakeholders. The nodes and users that want to join the network need to be authorized. Certification Authority (CA) can be used to perform the authorization. HLF [4] and Quorum [5] are typical permissioned blockchain systems.

Blockchain provides a tamper-proof distributed ledger by mechanisms such as decentralized architecture, consensus algorithm, asymmetric encryption, and so on. Transactions that are sent to the blockchain system are packed into blocks by certain rules. A block contains a block header and a block body, the transactions are recorded in the block body. Each block contains the hash of its previous block, so that the blocks form a chain structure, which means that tampering with a historical transaction can be obtained by comparing the block hash saved in the next block. The blocks are usually stored as files in the file system of blockchain nodes. Typically, each node of the blockchain network retains a full copy of the entire chain of blocks, so that each node can check the integrity of the data locally.

Since the size of the chain of blocks increases continually, the blockchain systems are facing a storage scalability issue. For permissionless blockchains, the data size of a

Bitcoin node has reached 366.51 GB [6] and the data size of a Ethereum node has reached 987.54 GB [7] by 28 September 2021; for permissioned blockchains, the issue has also caused concern [8,9]. The storage burden is becoming more onerous for individual participants who run blockchain nodes with their personal computers. This situation may injure the decentralized character of blockchain systems since only wealthy individuals or organizations can afford the increasing storage scaling demand.

Methods proposed to solve the storage scalability issue of blockchain systems are divided into "on-chain" solutions and "off-chain" solutions. On-chain solutions reduce the contents stored in each node by altering the range of consensus or the contents of transactions [10–13]. Off-chain solutions provide off-chain storage devices to store data, but most of them focus on reducing the data size before the data are uploaded to blockchains [14–17]. This approach uses blockchain as a proof repository to ensure the integrity of off-chain data, but the off-chain data cannot be retrieved through blockchain node, on the contrary, block archiver [18] reduces the data size of the data that has been uploaded to the blockchain and stored in the blockchain nodes, but block archiver fails to ensure the integrity of the data stored off-chain, which severely damages the tamper-proof property of the original blockchain system. To bridge this gap, an off-chain solution for the data that has been uploaded to the blockchain with tamper-proof property is proposed in this paper. The data are eliminated from the blockchain node and transmitted to a remote DB server while the node saves a concise vector commitment (VC) [19] to ensure the integrity of the data stored off-chain.

The contributions made by the present research are as follows:

1. An off-chain solution to solve the data scalability issue of blockchain systems while ensuring the integrity of the data stored off-chain is proposed, the target of this solution is to reduce the size of the data that has been uploaded to the blockchain and stored in the blockchain nodes. Each node saves a concise VC to ensure the integrity of the data of a block while the raw data are eliminated from the node.
2. The performance of our solution is improved compared to our primary solution "xFabLedger" [20] by separating the transaction-processing phase and the data-reduction phase. The data reduction is performed when a blockchain node is backed up to avoid read-write conflicts and negative impact on performance.
3. The solution is implemented based on Hyperledger Fabric (HLF) V2.3.2 [21], and storage experiments and performance experiments are conducted to prove the effectiveness of our solution.

The rest of the paper is organized as follows: Section 2 describes different approaches used when solving the scalability issues of blockchain. Section 3 introduces the transaction-processing model and storage structure of the HLF blockchain system and the mechanism of VC. Section 4 introduces our solution and then analyzes the security and performance features thereof. Section 5 demonstrates the details of our implementation based on HLF. Section 6 shows our experimental results and their evaluations. Section 7 concludes with recommendations for future research.

2. Related Work

Blockchain technology is not only underpinning cryptocurrencies, but also widely adopted in other industries, such as finance [22–24], IoT [14,25,26] and healthcare [27–29]. Although blockchain benefits the systems that use this technology, it brings scalability issues to them [30–32]. The methods to solve the storage scalability issue of blockchain are categorized into "on-chain" solutions and "off-chain" solutions.

2.1. On-Chain Solutions

"Sharding" is a method that divides the nodes of the blockchain network into subgroups called shards, each shard acts as an independent blockchain network to process transactions and store data [10,11]. This approach reduces the data need to store

for blockchain nodes, but each node still needs to store the full copy of the blocks of the subgroup.

Several works propose storing the data distributedly similar to traditional peer-to-peer content distribution networks [33–35] do. CUB [12] introduces the concept of "Consensus Unit" (CU), in which nodes combine their resources to maintain the blockchain data together rather than based on one copy per node. The differences between a CU and a shard are: (1) as a whole, each CU stores identical data while each shard stores their unique data; (2) each node in a shard stores identical data while each node in CU stores different blocks. Jidar [13] proposes a method allowing Bitcoin blockchain nodes to store only those transactions that are related to themselves. To verify a transaction, a proof is attached with the transaction when it is sent. Moreover, a bloom filter is added to each block to check whether a transaction has been affected.

On-chain solutions can reduce the storage burden of blockchain, but each node still needs to store at least part of the blocks. On-chain solutions also tend to introduce extra network overhead since the nodes must communicate to decide which node stores which part of the entire data.

2.2. Off-Chain Solutions

Most of the off-chain solutions focus on reducing the data size before the data are uploaded to blockchains [14–17]. In these solutions, blockchains work as proof repositories to ensure the integrity of off-chain data, but the problems are: (1) off-chain data cannot be retrieved through a blockchain node; (2) each node still need to store the full copy of the blocks.

Block archiver [18] is a solution used to reduce the data size of the data that has been uploaded to the blockchain and stored in the blockchain nodes by transmitting the data to an off-chain block archiver repository. It addresses the storage issue for HLF. In an organization (formed by serveral blockchain nodes) of HLF, a block archiver repository is deployed off-chain to store archived block files, a "block archiver" is deployed on the anchor/leader peer (a kind of blockchain node), and "block archiver clients" are deployed on other peers. The "block archiver" is responsible for transmitting block files from the peer's local file system to the block archiver repository, deleting local block files and notifying the "block archiver clients" to delete relevant block files from their local file systems. Fast fabric [36] also mentioned an approximate idea that storing the blocks in a distributed storage cluster. However, these solutions have not considered the method to keep the integrity of the block data after they are stored off-chain.

Our solution is proposed to address the aforementioned problems in current solutions. A remote DB server is used to store the data that has been stored in the blockchain nodes, while the off-chain stored data are still retrievable through blockchain nodes. Concise commitments are stored in each node to ensure the data integrity for the blocks, therefore the size of data that stored on blockchain nodes are extremely reduced. The concept of CU is also adopted to avoid centralization and reduce network overhead.

3. Background

To help understand our solution better, several relevant technologies are introduced as background in this section.

3.1. Hyperledger Fabric

3.1.1. A Modular Pluggable Blockchain System

Hyperledger Fabric is a popular open-sourced permissioned blockchain system. In a permissionless blockchain system, the fundamental components (e.g., consensus algorithm and block generation rules) must be pre-determined before the system starts to provide services, since each node in a blockchain system must obey the same rules. If a change is needed after the blockchain system begins to run, the so-called "fork" [37] operation is required. On the contrary, components in HLF are modular and pluggable. Provided the

stakeholders reach a consensus, changes to the system can be easily realized. The nodes in HLF networks are divided by their different functions. The nodes that are responsible for saving blocks and answering query requests are called peers, some of the peers are also obtained to carry out the calculations defined in the transactions, these peers are called endorsers. The peers in the network always belong to different organizations: each organization represents one or several stakeholders in the real world. The nodes that are responsible for ordering the transactions and packing them into blocks are called orderers, it is recommended that each organization deploys one orderer to represent the organization. The orderers provide ordering services together follow certain consensus algorithms such as Raft [38] or PBFT [39]. The orderers are referred to as Ordering Service Nodes (OSN). Figure 1 illustrates the structure of a typical HLF network.

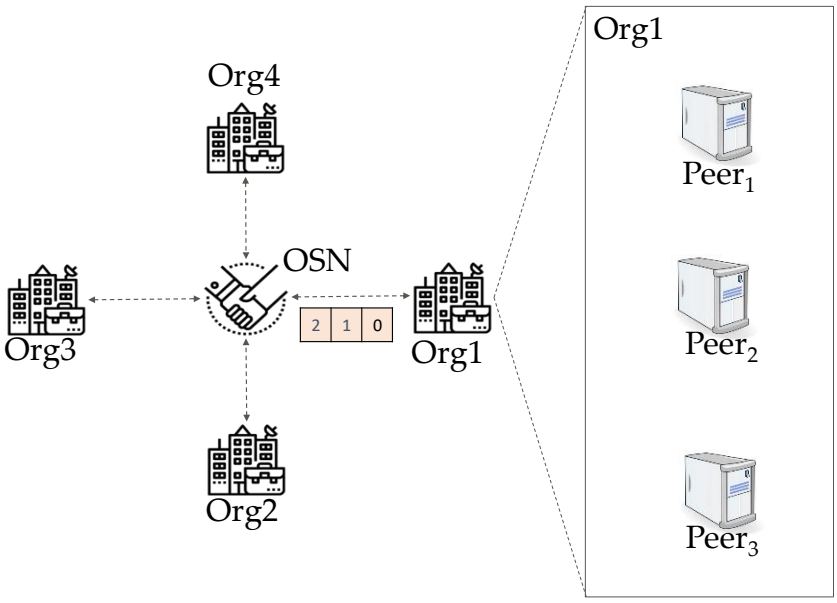

Figure 1. The structure of Hyperledger Fabric network. The Ordering Service Nodes (OSN) order blocks and deliver ordered blocks to each organization (Org). Each organization contains several peers.

3.1.2. Transaction-Processing Architecture

For most of the blockchain systems, a consensus of the order of transactions in a block must be reached before calculating the results of the transactions. Since each node will calculate the results independently to verify the results, the order of the transactions must be determined to maintain consistency among nodes. This architecture renders the computational resources of the blockchain system un-scalable. To solve this problem, HLF adopts a different transaction-processing architecture called "execute-order-validate" [4], in which transactions are calculated concurrently by multiple endorsers then ordered by OSN and later validated by peers.

3.1.3. Storage Structure of A Peer

The storage structure of an HLF peer comprises a chain of blocks and several local databases. In HLF, a block consists of three parts [21]:

- *Block Header* comprises three fields
  - *Block Number* is the sequence of the current block. The block number is counted from 0. The first block is called genesis block.
  - *Data Hash* is the hash value of all the transactions that are recorded in the current block.
  - *Previous Hash* is the Data Hash of its previous block.

- *Block Data* contains the ordered transactions. The transaction creators' and the endorsers' signatures are attached with each transaction. The count of transactions stored in a block is called the "block size" which can be configured before the network is started.
- *Block Metadata* contains the validation codes used to verify transactions and the signature of the block creator.

The blocks are chained together through the "Previous hash", and they are stored in the file system as block files. The size of block files is configured before the network is started so that the number of blocks that each block file contains is determined.

The blocks preserve all original information pertaining to each transaction, but for data retrieval, local databases are needed to provide the necessary functionality:

- *StateDB* records the newest status of all the objects defined in the applications of the blockchain system. The status is recorded as <Key, Value> pairs. It can be chosen that whether CouchDB [40] or LevelDB [41] is used as the state DB before the HLF network is started.
- *IndexDB* indicates where to find each block and transaction in the block files. It records the sequence number of the block file retaining the indexed blocks and transactions and offsets of those blocks and transactions.
- *HistoryDB* can be used to track each change for each key. The sequence numbers of key-related blocks and transactions are recorded.

3.1.4. Snapshot

A snapshot mechanism [42] is officially included in the HLF V2.3 to back-up a peer. A snapshot comprises several files that are exported from local databases of a peer. It can be used for checking whether ledger forks occur among different organizations and letting a new peer join the channel without synchronizing previously committed blocks from other peers. When a peer is going to generate a snapshot, the committing of blocks is stopped to prevent read-write conflicts.

3.2. Vector Commitment

For a given element e and a sequence number i, VC [19] can be used to identify whether e is equal to the ith element of an ordered set $\mathbf{e}$ $(e_1, \ldots e_n)$. In blockchain systems, the transactions in a block form an ordered set of elements, thus VC can be used to identify whether a given transaction is in a block and its position is also correctly provided. VC demonstrates the features of hiding, position binding, conciseness and an ability to be updated. The following six algorithms are used to define VC:

1. Given security parameter k and the size n of ordered set $\mathbf{e}$, the "*KeyGen*" algorithm outputs the public parameters pp.

$$pp \leftarrow KeyGen(1^k, n) \tag{1}$$

2. Given the public parameters pp and the ordered set $\mathbf{e}$, the "Commitment" algorithm outputs the commitment $\mathbf{C}$ of $\mathbf{e}$.

$$\mathbf{C} \leftarrow Commitment_{pp}(e_1, ..., e_n) \tag{2}$$

3. The "Open" algorithm is used to generate the proof π_i for the ith element.

$$\pi_i \leftarrow Open_{pp}(i, e_1, ..., e_n) \tag{3}$$

4. The "Verify" algorithm is employed to check whether the given element e is equal to e_i through the proof π_i. If the proof is accepted, the algorithm outputs 1.

$$\{0, 1\} \leftarrow Verify_{pp}(\mathbf{C}, i, e, \pi_i) \tag{4}$$

5. When the ith element of **e** needs to be updated with $e^{'}$, the "Update" algorithm can be run to output a new commitment $\boldsymbol{C}^{'}$ and the update information U.

$$\mathbf{C}^{'}, U \leftarrow Update_{pp}(\mathbf{C}, e, e^{'}, i) \tag{5}$$

6. When the ith element of **e** is updated, the proofs of other elements (e.g., e_j) also need to be updated. The algorithm "ProofUpdate" outputs the updated proof $\boldsymbol{\pi}_j^{'}$ of element e_j.

$$\boldsymbol{\pi}_j^{'} \leftarrow ProofUpdate_{pp}(\boldsymbol{\pi}_j, e^{'}, i, U) \tag{6}$$

4. SASLedger: A Secured, Accelerated Scalable Storage Solution

To relieve the storage burden for blockchain nodes, we focus on reducing the size of the chain of blocks since it alone is responsible for most of the storage burden of a blockchain node. Figure 2 shows the system architecture of our solution. An off-chain solution within each CU is used to achieve lower network overhead compared to on-chain solutions or solutions without adopting CU [13] or solutions without adopting CU [10–13]: an off-chain remote DB server for each CU is added to store the blocks; VC is used to ensure the integrity of each transaction in a block stored in the remote DB server. Although the blocks are stored in the remote DB server, the nodes still maintain their local databases respectively and still be able to check the data integrity of the blocks that they transmit to the remote DB server, thus the decentralized character and tamper-proof character of blockchain system are retained.

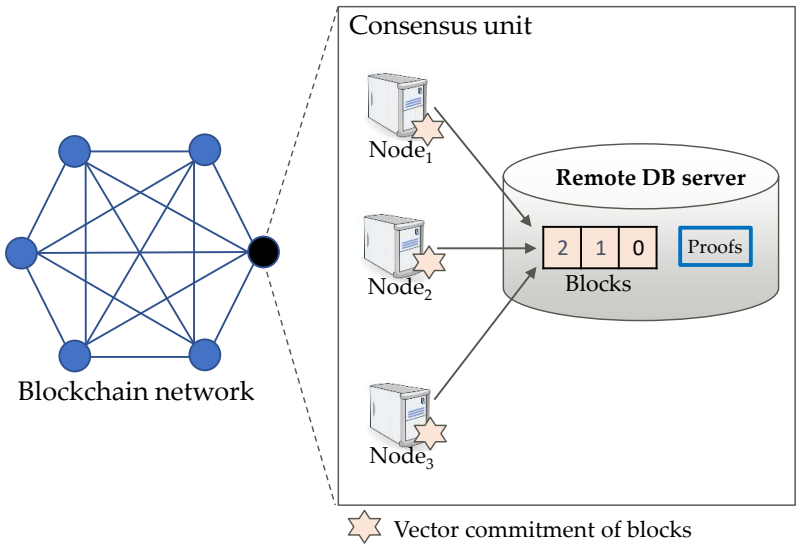

Figure 2. System architecture of our solution. The entire blockchain network is divided into multiple Consensus Unit (CU), in which nodes combine their resources to maintain the blockchain data together.

4.1. Proving Data Integrity through VC

To prove the integrity of the data in a block, Jidar [13] attaches a proof for each transaction, which requires extra storage space and generates significant network overhead. Compared to Jidar, in our solution, nodes only store a concise VC of a block, the proof for a transaction is generated and provided by the remote DB server only when the transaction is requested.

A Merkle tree [43] is used to generate VC for the set of transactions in a block and proofs for the transactions. As shown in Figure 3, the hashes of the transactions in a block $(tx_1, \ldots, tx_n)$ form the leaf nodes of the Merkle tree, the root of the Merkle tree is considered to be VC (**C**) of the block (7). When the proof of the ith transaction tx_i is requested, the list

which includes adjacent sibling nodes of the corresponding leaf and its ancestor nodes is returned as the proof π (8), in Figure 3, the list $[Hash_3, Hash_{12}, Hash_{5678}]$ (shown in yellow) is the proof of transaction tx_4.

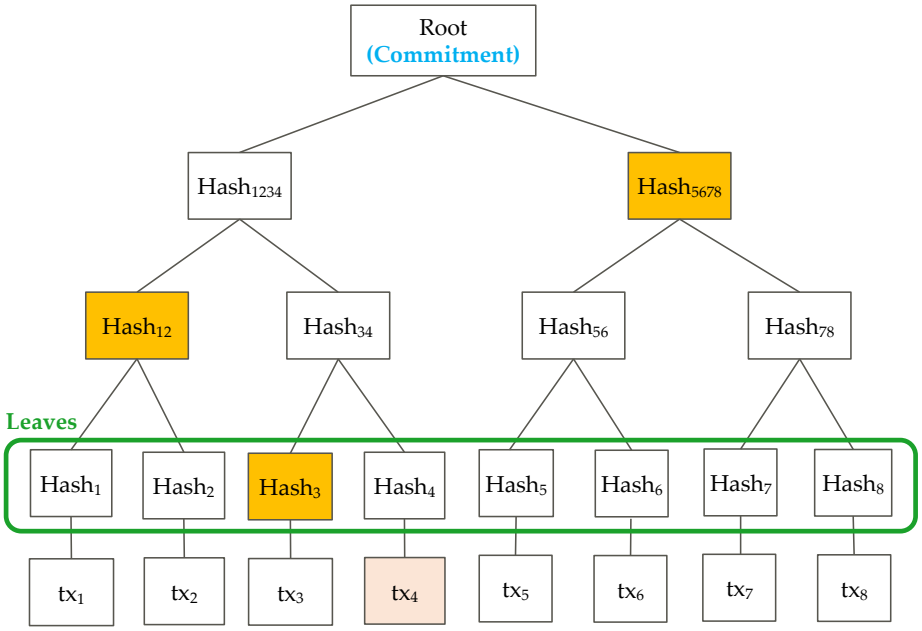

Figure 3. Merkle tree-based vector commitment. The hashes of the transactions in a block $(tx_1, \ldots, tx_n)$ form the leaf nodes of the Merkle tree, the root of the Merkle tree is considered to be VC of the block. The list $[Hash_3, Hash_{12}, Hash_{5678}]$ (shown in yellow) is the proof of transaction tx_4.

When a transaction is requested, the transaction is retrieved from the remote DB server with its proof and verified by calculating the Merkle tree root using the received proof and comparing the root with the previously stored commitment (9). If the root equals the VC, the node can believe that the transaction has not been subject to tampering on the remote DB server. In the case of a block being requested, the node retrieves the block from the remote DB server and validates its integrity by calculating its hash and comparing it with the previously stored hash of the block. The size of a VC is 32 B when SHA256 is used as the hash calculating algorithm while the size of a block with default settings containing 10 transactions is 34 KB [20]. Since the size of a VC is much less than the size of a block, storing its VC instead of the original block can save much storage space.

$$\mathbf{C} \leftarrow Commitment(tx_1, ..., tx_n) \tag{7}$$

$$\pi \leftarrow Open(i, tx_1, ..., tx_n) \tag{8}$$

$$\{0, 1\} \leftarrow Verify(\mathbf{C}, i, tx, \pi) \tag{9}$$

Since the transactions in a block are fixed, the "Update" and "ProofUpdate" algorithms in conventional VC definition are not applicable. The "KeyGen" algorithm is unnecessary neither since there are no other public parameters to be generated. The algorithms are shown in detail with Algorithms A1–A4 in Appendix A.

4.2. A Primary Solution: xFabLedger

xFabLedger [20] is our primary solution in which after each node receives a block, the node generates commitment for the block and transmits the block to the remote DB server. The block is never stored in the file system of the node. Figure 4 illustrates how a block is processed. xFabLedger solves the excessive storage growth issue and security issue, but

experiments indicates that the performance is decreased compared to that of the original blockchain system.

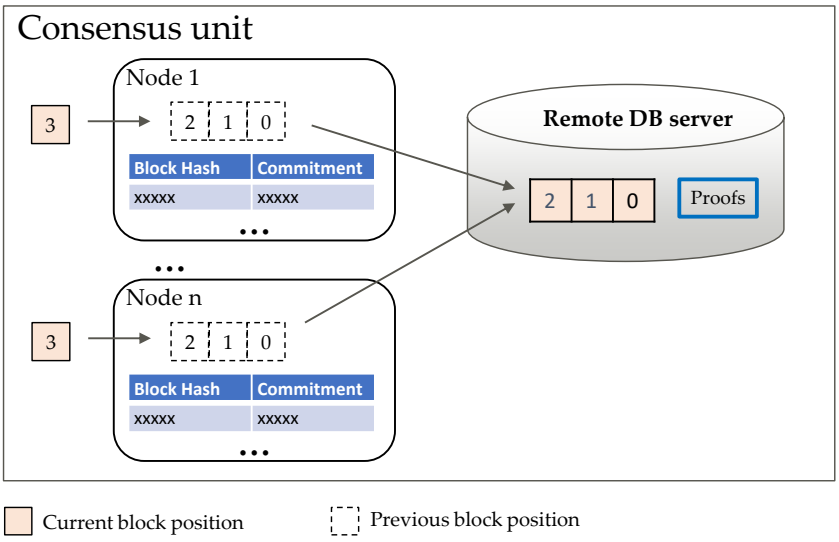

Figure 4. The primary solution. After a block is received by a blockchain node, the node generates commitment for the block and transmits the block to the remote DB server immediately.

4.3. The Advanced Solution: SASLedger

In our advanced solution, the data-reduction operation is separated from the transaction-processing phase. The data-reduction operation should be performed when the system is not busy or when the system is being backed up (e.g., the snapshot mechanism in HLF). Figure 5 shows the two phases of our solution: during the transaction-processing phase, the nodes process the transactions as vanilla blockchain nodes do and save the blocks locally, thus the TPS remains unaffected. During the data-reduction phase, the nodes perform the following operations for each committed block:

- Calculate a VC for the set of transactions contained in the block;
- Save the hash value of the block and its commitment to local storage as a $\langle Key, Value \rangle$ pair which is used as an index of the block: $\langle blockHash, commitment \rangle$;
- Transmit the content of the block to the remote DB server;
- Delete the block from its local storage.

When a query is sent to a node to retrieve a block or a transaction, the node retrieves the block or transaction from the remote DB server and transmits it to the client after verification.

A snapshot of the blocks can be generated from the block indices: this can be used to build block indices for a newly joined node in the future.

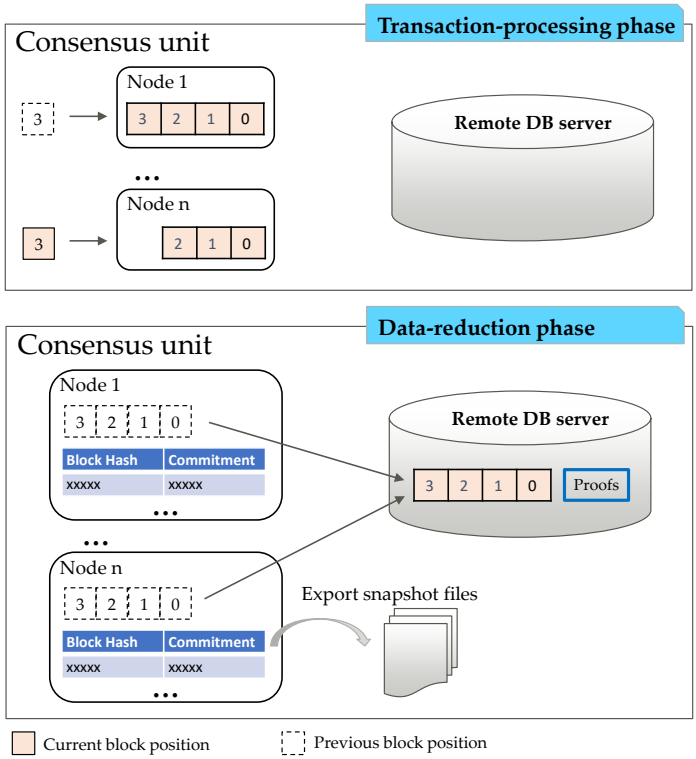

Figure 5. Two phases of our advanced solution. In the transaction-processing phase, the nodes process the transactions as vanilla blockchain nodes do and save the blocks locally. In the data-reduction phase, the nodes generate commitments for the blocks, transmit the blocks to the remote DB server, and delete the blocks locally.

5. Implementation Based on Hyperledger Fabric

Our advanced solution is implemented based on HLF V2.3.2. The reasons why HLF is chosen as the basis for our implementation are:

1. We are focusing on enterprise-level applications while HLF is a popular permissioned blockchain solution for information-sharing among companies due to its high performance and rich privacy preserving mechanisms [4,21];
2. The nodes in a HLF network are already divided into "organizations", which is suitable for building a CU since the peers in an organization are managed by the same administrator, and they are identified by each other through digital signatures;
3. The snapshot mechanism introduced in HLF V2.3.2 provides a perfect timing to perform the "data-reduction" operation of our solution.

A remote DB server is deployed in each organization of HLF to store all the blocks previously stored in the peers in this organization. When a new generated block is arrived, the peers commit the block (saving the block to its file system and saving the relevant information to its local databases) as vanilla HLF peers do, so that the TPS of the blockchain system is unaffected. The data-reduction operation is conducted when a snapshot is generated.

5.1. Peer Node

As mentioned in Section 3, a peer node of HLF performs the following operations:

- *Endorsing*: performing the calculations defined in transactions;
- *Committing*: preserving blocks to the file system of the peer and preserving some block-related information to the peer's local databases;
- *Responding query requests*: searching blocks and local databases to find information that is requested by users and answering the requests;
- *Generating snapshot*: stop committing and then exporting information from local databases to snapshot files.

The "Generating snapshot" operation is modified to transmit the blocks to the remote DB server and delete the blocks from its local file system. The "Responding query requests" operation is also modified by adding verification process after the blocks and transactions are retrieved from the remote DB server. The communications between peers and the remote DB server are realized by way of a GRPC protocol [44]. The "Endorsing" and "Committing" operations remain unmodified.

5.1.1. Modification of Generating Snapshot

Before a block is transmitted to the remote DB server, the commitment of the block is generated and the indexDB of the peer is updated, so that the peer can retrieve transactions and blocks from the remote DB server after transmitting the blocks. A comparison of contents in indexDB before and after the update is provided (Table 1).

Table 1. Contents of indexDB before and after updating.

Key	Before Update	After Update
blkHash	block file, block offset	VC for the block, hash of the entire block [a]
blkNum	block file, block offset	blkHash
txID, blkNum,txNum	block file, block offset, tx offset,tx validation code	tx validation code

[a] MetaData is not included when blockHash is calculated.

In HLF implementation, a peer who joins a channel through a snapshot can obtain the latest world states when the snapshot is generated; but it cannot access the contents of blocks that have been committed before the snapshot is generated since the local databases of the peers that generate the snapshot only contain information of those blocks that are stored locally. However, in our implementation, since the indexDB is updated for the blocks that are going to be transmitted to the remote DB server, those blocks can be accessed by new peers if they join the channel through copying the information from the snapshot to their local databases. To support this function fully, the snapshot contents exported from different local databases are extended: besides the contents exported from stateDB and configHistoryDB which are also included in the snapshot of HLF, the snapshot of SASLedger also contains the history records of states that are exported from the historyDB and all the <Key, Value> records in the indexDB after it is updated.

5.1.2. Modification of "Responding Query Requests"

In HLF, peers provide four query interfaces for users:

- *QueryTransaction* is used to retrieve a transaction-by-transaction ID;
- *QueryBlock* is employed to retrieve a block-by-block number;
- *QueryBlockByHash* is to obtain a block-by-block hash;
- *QueryBlockByID* is to acquire a block using the transaction ID of one of the transactions that is compacted within the block.

Since SASLedger retrieves blocks and transactions from the remote DB server, the retrieved data must be verified to ensure that they are not subject to tampering on the remote DB server. The verification is performed by the peer as described in Section 4.1.

5.2. Remote DB Server

The remote DB server is implemented based on leveldb. It comprises three databases: blockDB, txIndexDB, and merkleTreeDB. When a block is received by the remote DB server, first, the redundancy of the block is checked through its hash; secondly, if the received block has not already been saved, the blockData are saved to the blockDB, the transactions in the block body are traversed to save the indices of the transactions to the txIndexDB,

and the Merkle Tree is calculated for the block and saved to the merkleTreeDB. Table 2 lists the contents of these databases.

Table 2. Contents of remote databases.

Database	Key	Value
blockDB	blkHash	blockData
txIndexDB	txID	blkHash, txNum [a]
merkleTreeDB	merkleNode	adjacent sibling node

[a] The sequence number that the tx is stored in the block.

The remote DB server provides four APIs for the peers to invoke:

- *AddBlock*: uploads a block to remote DB server;
- *GetBlockByHash*: retrieves a block by blockHash and returns the block;
- *GetBlockByTxId*: retrieves a block by transaction ID and returns the block;
- *GetTransactionByTxId*: retrieves a transaction-by-transaction ID and returns the transaction and its proof.

6. Experiment and Evaluation

Experiments are conducted to elucidate how much storage space the SASLedger can save compared with HLF. Meanwhile, experiments are performed to compare the TPS and query latencies of SASLedger, xFabLedger, and HLF.

6.1. Basic Experimental Settings

Table 3 shows the configuration of the testing machines that we use to perform these experiments. We use six computers, each with identical configurations: three of them are used as blockchain nodes, one of them as the remote DB server. and the other two as Performance Traffic Engine (PTE) [45] machines for sending transactions to the blockchain networks. Table 4 lists the settings for blockchain networks.

Table 3. Testing machine configuration.

Configuration Item	Value
OS	Ubuntu 16.04
CPU	i5-9400 (2.9 GHz) 6 cores
Memory	8 GB
Hard Disk	SSD 512 GB
Network Bandwidth	1000 Mbps

Table 4. Blockchain network settings.

Blockchain Network Component	Value
Orderer node number	One
Consensus algorithm	Raft
Organization number	Two
Peer number in each organization	One
StateDB	leveldb

6.2. Storage Experiments

Experiments are carried out to understand to what level the SASLedger can relieve the storage burden of a blockchain node. In Figure 6, the grey bars and red bars show the block file storage demand of HLF and SASLedger, respectively; the yellow bars indicate the disk size of the remote DB server; the green line shows the storage consumption reduction rate (10) of SASLedger compared with HLF. When the number of transactions reaches 480,000, the reduction rate of peer block file size reaches 93.3%. Since the first block file containing

the genesis block and the latest block file are kept when deleting block files in peers, the SASLedger holds storage consumption records (red bars in Figure 6) of a constant value when the number of transactions changes.

$$reduction\_rate = \frac{HLF\_size - SASLedger\_size}{HLF\_size} \quad (10)$$

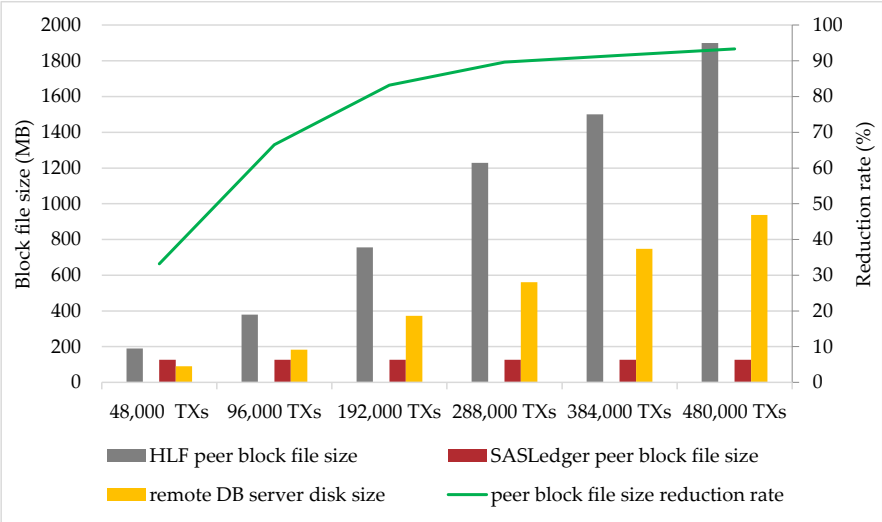

Figure 6. Storage comparison between HLF and SASLedger. The grey bars and red bars show the block file storage demand of HLF and SASLedger, respectively; the yellow bars indicate the disk size of the remote DB server; the green line shows the storage consumption reduction rate of SASLedger compared with HLF.

6.3. Performance Experiments

To assess the performance of blockchain networks, two PTE machines are used to send a certain number of write-only transactions to the two organizations. The configuration of the PTE is summarized in Table 5.

Table 5. PTE settings.

Setting Item	Value
PTE machine number	Two
Process number per PTE	Eight
Tx number per Process	3000

The TPS values of HLF, xFabLedger, and SASLedger are compared when they adopt different block sizes. Figure 7 shows the experimental results. For each of the three blockchain systems, the TPS increases as the block size increases. The reason for this is that over the same number of transactions, the larger the block size, the fewer blocks are generated thus the number of block-oriented operations is reduced. Since xFabLedger transmits the block to the remote DB server during the transaction-processing phase, its TPS is lower than that when using HLF. SASLedger separates the block transmission from transaction-processing phase, which leads to a TPS improvement of 9.6% compared with xFabLedger.

Figure 8 shows the block commitment latency comparison of HLF, xFabLedger, and SASLedger when the block size is changed. The block commitment latency is the time at which a block and its relevant information are recorded. It can be divided into three parts:

- *BlockCommitTime* is the time span to record the block itself;
- *StateCommitTime* is the time span to write relevant information to stateDB;
- *HistoryCommitTime* is the time span to write relevant information to historyDB.

Figure 8 indicates that the total block commitment latency of xFabLedger with each block size is the highest among the three blockchain systems and the commitment latencies of HLF and SASLedger are similar. This observation can explain the result shown in Figure 7.

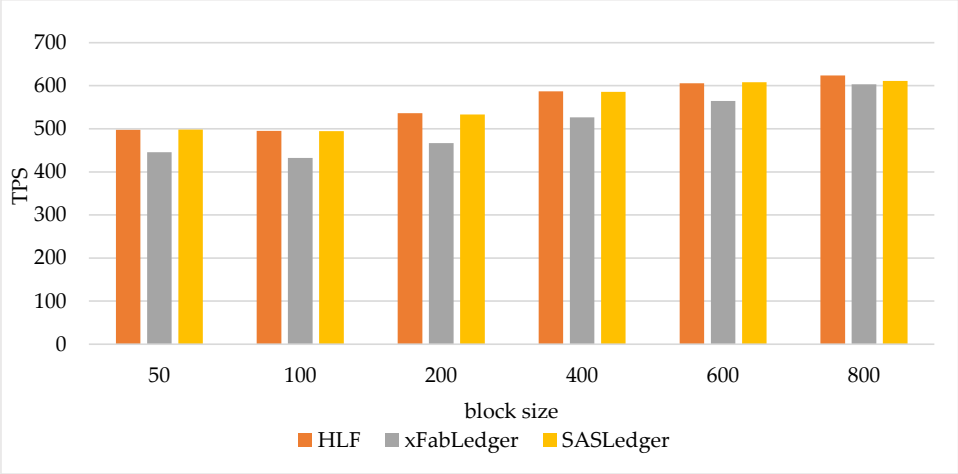

Figure 7. Throughput comparison of HLF, xFabLedger, and SASLedger.The TPS of xFabLedger is lower than that when using HLF. The TPS is improved to the same level with HLF in SASLedger.

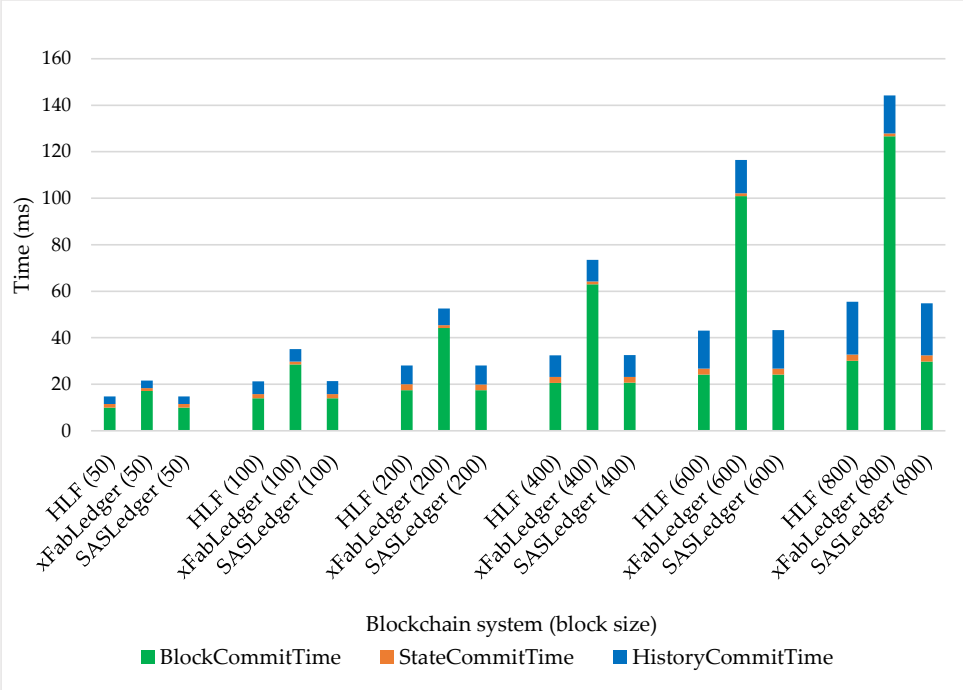

Figure 8. Block commitment latency comparison: HLF, xFabLedger, and SASLedger. The total block commitment latency of xFabLedger with each block size is the highest among the three blockchain systems and the commitment latencies of HLF and SASLedger are similar.

For the performance of query requests, the query latencies of the three blockchain systems are tested by calling the four query interfaces: QueryTransaction, QueryBlock, QueryBlock, and QueryBlockByID.

In SASLedger a block is stored in the peers before it is transmitted to the remote DB server. Figure 9 shows that if a block is still stored in the local file system of a peer, the time to query it and the transactions therein is similar to the time cost when using HLF, while

for a block that has been transmitted to the remote DB server, the query latency is similar to that when using xFabLedger. Since the blocks are either stored locally in the peers or stored in the remote DB server, the overall query performance is significantly affected by the distribution of stored locations. In the worst situation that all blocks are stored in the remote DB server, the block-query-latency and transaction-query-latency of SASLedger are 31% and 67% higher than those of HLF, respectively.

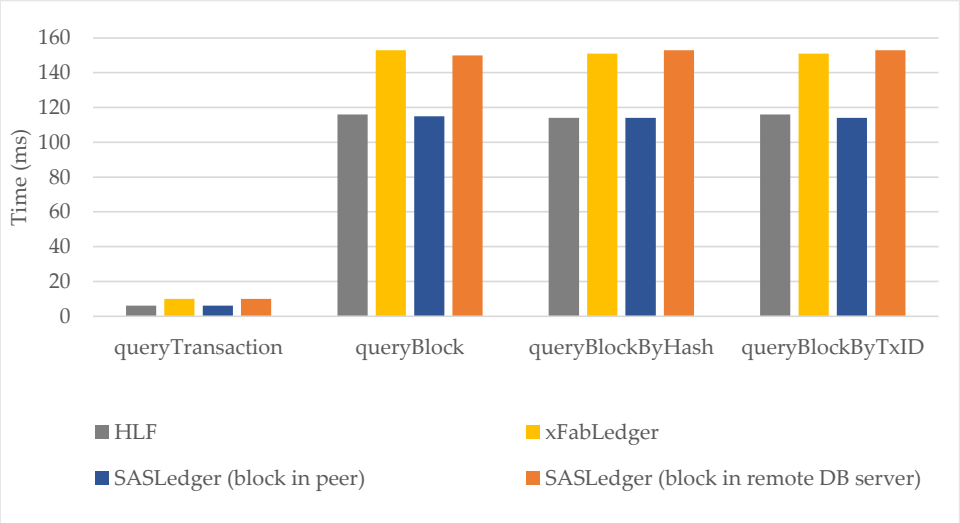

Figure 9. Query latency comparison: HLF, xFabLedger, and SASLedger. When using SASLedger, for a block stored in the local file system of a peer, the time to query it and the transactions therein is similar to the time cost when using HLF, while for a block that has been transmitted to the remote DB server, the query latency is similar to that when using xFabLedger.

7. Conclusions

In this paper, a secured and accelerated scalable storage solution for blockchain systems is proposed. Our solution relieves the storage burden of blockchain nodes by adding an off-chain remote DB server for each CU in blockchain network to store the blocks while ensuring data integrity. Our solution also improves the performance compared to our previous work by separating the data-deduction operation from the transaction-processing phase. Experiments show that our solution can reduce the size of block files for peers by 93.3% compared to the original HLF blockchain system and that our solution enhances the TPS by 9.6% compared to xFabLedger.

Our solution can be applied in blockchain systems constructed with HLF to enhance its storage scalability by simply replacing the vanilla HLF peers with the modified SASLedger peers and deploying the remote DB server. For conventional HLF blockchain systems, if the storage spaces of peers are insufficient, additional hard disks need to be installed to each peer; however, in a SASLedger blockchain system, new hard disks only need to be installed in the remote DB server, which reduces the budget.

There are several limitations left in our solution:

- Retrieving blocks or transactions from the remote DB server is slower than retrieving them from the local file system of a peer, thus our solution leads to a decrease of the query performance.
- The data-reduction phase of our solution depends on the snapshot operation of HLF, since the commitment of transactions are stopped when taking the snapshot, the performance is decreased due to the snapshot operation.

In the future, we would like to study the mechanism of the local databases of HLF peers to solve the read-write conflict problem when exporting the contents of the local

databases. This may accelerate the current snapshot generation process by keeping the commitment of transactions unstopped when a snapshot is taken.

Author Contributions: Conceptualization, B.P., J.S. and H.S.; methodology, H.S. and B.P.; software, H.S.; validation, H.S. and B.P.; formal analysis, B.P.; investigation, H.S. and B.P.; resources, H.S. and B.P.; data curation, B.P.; writing—original draft preparation, H.S.; writing—review and editing, B.P., J.S., T.M. and M.M.; visualization, H.S.; supervision, J.S.; project administration, B.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Not Applicable, the study does not report any data.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

DB	Database
HLF	Hyperledger Fabric
Org	Organization
OSN	Ordering Service Nodes
BlkNum	Block Number
BlkHash	Block Hash
Tx	Transaction
PBFT	Practical Byzantine Fault Tolerant
VC	Vector Commitment
CU	Consensus Unit
BAO	Blocks Assignment Optimization
TPS	Transactions Per Second
OS	Operating System
CPU	Central Processing Unit
SSD	Solid-State Driver
GB	Gigabytes
MB	Megabytes

Appendix A.

In the following algorithms, "=" means "equals to" and "||" means "concatenation".

Algorithm A1: Initialization

$txArray \leftarrow [tx_1, ..., tx_n]$
$MerkleTree.layerArray \leftarrow emptyArray$ `/* Initialize the layers of the MerkleTree with an empty array */`

Function `CalcHashForTxs(`$txArray$`)`**:**
  $i \leftarrow 0$
  while $i <$ *sizeOf(txArray)* **do**
    $hashArray[i] \leftarrow sha256(txArray[i] \,||\, i)$
    $i \leftarrow i + 1$
  end
  return $hashArray$

Algorithm A2: Commitment and Construction of MerkleTree

```
Function Commitment(txArray):
    hashArray ← CalcHashForTxs(txArray)
    return CommitRecursively(hashArray)

Function CommitRecursively(layer):
    i ← 0
    defaultSibling ← sha256(constantA)
    while i < sizeOf(layer) do
        if layer[i + 1] exists then
            if layer[i] < layer[i+1] then
                upperLayer[i/2] ← sha256(layer[i] || layer[i + 1])
            else
                upperLayer[i/2] ← sha256(layer[i + 1] || layer[i])
            end
            layer[i].parent ← upperLayer[i/2]    /* Record the node's parent */
            layer[i].sibling ← layer[i + 1]      /* Record the node's sibling */
            layer[i + 1].parent ← upperLayer[i/2]
            layer[i + 1].sibling ← layer[i]
        else
            if layer[i] < defaultSibling then
                upperLayer[i/2] ← sha256(layer[i] || defaultSibling)
            else
                upperLayer[i/2] ← sha256(defaultSibling || layer[i])
            end
            layer[i].parent ← upperLayer[i/2]
            layer[i].sibling ← defaultSibling
        end
        i ← i + 2
    end
    MerkleTree.layerArray.append(layer)          /* Save current layer to
     MerkleTree */
    if 1 = sizeOf(upperLayer) then
        MerkleTree.layerArray.append(upperLayer)   /* Save the upper layer
         to MerkleTree */
        return upperLayer[0]
    else
        return CommitRecursively(upperLayer)
    end
```

Algorithm A3: Open

```
Function Open(i, txArray):
    hashArray ← CalcHashForTxs(txArray)
    proofArray ← emptyArray
    return GetProofRecursive(hashArray[i], proofArray)

Function GetProofRecursively(node, proofArray):
    sibling ← MerkelTree.getSibling(node)      /* Find the node's sibling */
    if sibling dose not exist then
        return proofArray
    else
        proofArray.append(sibling)
        parent ← MerkelTree.getParent(node)    /* Find the node's parent */
        return GetProofRecursively(parent, proofArray)
    end
```

Algorithm A4: Verify

```
Function Verify(C, i, tx, proofArray):
    result ← sha256(tx || i)
    j ← 0
    while j < sizeOf(proofArray) do
        if result < proofArray[j] then
            result ← hash256(result || proofArray[j])
        else
            result ← hash256(proofArray[j] || result)
        end
        j ← j + 1
    end
    if C = result then
        return 1
    else
        return 0
    end
```

References

1. Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. Available online: https://bitcoin.org/bitcoin.pdf (accessed on 15 October 2021).
2. Ethereum Home. Available online: https://ethereum.org/en/ (accessed on 15 October 2021).
3. Home—EOSIO Blockchain Software & Services. Available online: https://eos.io/ (accessed on 5 November 2021).
4. Androulaki, E.; Barger, A.; Bortnikov, V.; Cachin, C.; Christidis, K.; De Caro, A.; Enyeart, D.; Ferris, C.; Laventman, G.; Manevich, Y.; et al. Hyperledger fabric: A distributed operating system for permissioned blockchains. In Proceedings of the Thirteenth EuroSys Conference, Porto, Portugal, 23–26 April 2018; Association for Computing Machinery: New York, NY, USA, 2018; pp. 1–15.
5. Quorum. Available online: https://consensys.net/quorum/ (accessed on 8 October 2021).
6. Bitcoin Blockchain Size. Available online: https://ycharts.com/indicators/bitcoin_blockchain_size (accessed on 29 September 2021).
7. Ethereum Chain Full Sync Data Size. Available online: https://ycharts.com/indicators/ethereum_chain_full_sync_data_size (accessed on 29 September 2021).
8. 10 Practical Issues for Blockchain Implementations. Available online: https://www.hyperledger.org/blog/2020/03/31/title-10-practical-issues-for-blockchain-implementations (accessed on 24 November 2021).
9. Ledger Snapshot and Checkpoint. Available online: https://jira.hyperledger.org/browse/FAB-106 (accessed on 24 November 2021).

10. Chow, S.S.; Lai, Z.; Liu, C.; Lo, E.; Zhao, Y. Sharding blockchain. In Proceedings of the 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), Halifax, NS, Canada, 30 July–3 August 2018; p. 1665.
11. Wang, J.; Wang, H. Monoxide: Scale out blockchains with asynchronous consensus zones. In Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Boston, MA, USA, 26–28 February 2019; pp. 95–112.
12. Xu, Z.; Han, S.; Chen, L. CUB, a consensus unit-based storage scheme for blockchain system. In Proceedings of the 2018 IEEE 34th International Conference on Data Engineering (ICDE), Paris, France, 16–19 April 2018; pp. 173–184.
13. Dai, X.; Xiao, J.; Yang, W.; Wang, C.; Jin, H. Jidar: A jigsaw-like data reduction approach without trust assumptions for bitcoin system. In Proceedings of the 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), Dallas, TX, USA, 7–10 July 2019; pp. 1317–1326.
14. Honar Pajooh, H.; Rashid, M.; Alam, F.; Demidenko, S. Hyperledger Fabric Blockchain forSecuring the Edge Internet of Things. *Sensors* **2021**, *21*, 359. [CrossRef] [PubMed]
15. Hwang, H.C.; Shon, J.G.; Park, J.S. Design of an Enhanced Web Archiving System for Preserving Content Integrity with Blockchain. *Electronics* **2020**, *9*, 1255. [CrossRef]
16. Galiev, A.; Prokopyev, N.; Ishmukhametov, S.; Stolov, E.; Latypov, R.; Vlasov, I. Archain: A novel blockchain based archival system. In Proceedings of the 2018 Second World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4), London, UK, 30–31 October 2018; pp. 84–89.
17. Miyamae, T.; Kozakura, F.; Nakamura, M.; Zhang, S.; Hua, S.; Pi, B.; Morinaga, M. ZGridBC: Zero-Knowledge Proof based Scalable and Private Blockchain Platform for Smart Grid. In Proceedings of the 2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC), Sydney, Australia, 3–6 May 2021; pp. 1–3.
18. Hyperledger Fabric Block Archiving. Available online: https://github.com/hyperledger-labs/fabric-block-archiving (accessed on 18 October 2021)
19. Catalano, D.; Fiore, D. Vector commitments and their applications. In Proceedings of the International Workshop on Public Key Cryptography (PKC), Nara, Japan, 26 February–1 March 2013; Springer: Berlin/Heidelberg, Germany, 2013; pp. 55–72.
20. Pi, B.; Pan, Y.; Zhou, E.; Sun, J.; Miyamae, T.; Morinaga, M. xFabLedger: Extensible Ledger Storage for Hyperledger Fabric. In Proceedings of the 2021 IEEE 11th International Conference on Electronics Information and Emergency Communication (ICEIEC), Beijing, China, 18–20 June 2021; pp. 5–11.
21. A Blockchain Platform for the Enterprise. Available online: https://hyperledger-fabric.readthedocs.io/en/release-2.3/index.html (accessed on 8 October 2021).
22. Chen, B.; Tan, Z.; Fang, W. Blockchain-based implementation for financial product management. In Proceedings of the 2018 28th International Telecommunication Networks and Applications Conference (ITNAC), Sydney, Australia, 21–23 November 2018; pp. 1–3.
23. Chen, Y.; Bellavitis, C. Blockchain disruption and decentralized finance: The rise of decentralized business models. *J. Bus. Ventur. Insights* **2020**, *13*, e00151. [CrossRef]
24. Caldarelli, G.; Ellul, J. The Blockchain Oracle Problem in Decentralized Finance—A Multivocal Approach. *Appl. Sci.* **2021**, *11*, 7572. [CrossRef]
25. Sun, H.; Hua, S.; Zhou, E.; Pi, B.; Sun, J.; Yamashita, K. Using ethereum blockchain in Internet of Things: A solution for electric vehicle battery refueling. In Proceedings of the International Conference on Blockchain, Seattle, WA, USA, 25–30 June 2018; pp. 3–17.
26. Xu, R.; Nagothu, D.; Chen, Y. EconLedger: A Proof-of-ENF Consensus Based Lightweight Distributed Ledger for IoVT Networks. *Future Internet* **2021**, *13*, 248. [CrossRef]
27. Połap, D.; Srivastava, G.; Yu, K. Agent architecture of an intelligent medical system based on federated learning and blockchain technology. *J. Inf. Secur. Appl.* **2021**, *58*, 102748. [CrossRef]
28. Wang, B.; Li, Z. Healthchain: A Privacy Protection System for Medical Data Based on Blockchain. *Future Internet* **2021**, *13*, 247. [CrossRef]
29. Yaqoob, I.; Salah, K.; Jayaraman, R.; Al-Hammadi, Y. Blockchain for healthcare data management: Opportunities, challenges, and future recommendations. *Neural Comput. Appl.* **2021**, 1–16. [CrossRef]
30. Zhou, Q.; Huang, H.; Zheng, Z.; Bian, J. Solutions to scalability of blockchain: A survey. *IEEE Access* **2020**, *8*, 16440–16455. [CrossRef]
31. Khan, D.; Jung, L.T.; Hashmani, M.A. Systematic Literature Review of Challenges in Blockchain Scalability. *Appl. Sci.* **2021**, *11*, 9372. [CrossRef]
32. Antal, C.; Cioara, T.; Anghel, I.; Antal, M.; Salomie, I. Distributed Ledger Technology Review and Decentralized Applications Development Guidelines. *Future Internet* **2021**, *13*, 62. [CrossRef]
33. Stoica, I.; Morris, R.; Karger, D.; Kaashoek, M.F.; Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM Sigcomm Comput. Commun. Rev.* **2001**, *31*, 149–160. [CrossRef]
34. Maymounkov, P.; Mazieres, D. Kademlia: A peer-to-peer information system based on the xor metric. In Proceedings of the International Workshop on Peer-to-Peer Systems, Cambridge, MA, USA, 7–8 March 2002; Springer: Berlin/Heidelberg, Germany, 2002; pp. 53–65.
35. Flooding. Available online: https://www.cs.yale.edu/homes/aspnes/pinewiki/Flooding.html (accessed on 21 October 2021).

36. Gorenflo, C.; Lee, S.; Golab, L.; Keshav, S. FastFabric: Scaling hyperledger fabric to 20 000 transactions per second. *Int. J. Netw. Manag.* **2020**, *30*, e2099. [CrossRef]
37. Blockchain Forks Explained. Available online: https://medium.com/digitalassetresearch/blockchain-forks-explained-8ccf304b97c8 (accessed on 21 October 2021).
38. The Raft Consensus Algorithm. Available online: https://raft.github.io/ (accessed on 21 October 2021).
39. Castro, M.; Liskov, B. Practical byzantine fault tolerance. In Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, LA, USA, 22–25 February 1999; Volume 99, pp. 173–186.
40. CouchDB. Available online: http://couchdb.apache.org/ (accessed on 21 October 2021).
41. Google/Leveldb. Available online: https://github.com/google/leveldb (accessed on 21 October 2021).
42. Taking Ledger Snapshots and Using Them to Join Channels. Available online: https://hyperledger-fabric.readthedocs.io/en/release-2.3/peer_ledger_snapshot.html (accessed on 21 October 2021).
43. Merkle, R.C. A digital signature based on a conventional encryption function. In Proceedings of the Conference on the Theory and Application of Cryptographic Techniques, Santa Barbara, CA, USA, 16–20 August 1987; Springer: Berlin/Heidelberg, Germany, 1987; pp. 369–378.
44. gRPC. Available online: https://grpc.io/ (accessed on 21 October 2021).
45. Performance Traffic Engine—PTE. Available online: https://github.com/hyperledger/fabric-test/tree/main/tools/PTE (accessed on 21 October 2021).