SolsDB: Solve the Ethereum Bottleneck Cause by Storage

Hua Yang
Qingdao university of technology
Qingdao, China
yhua0919@gmail.com

ABSTRACT

This paper exploits the features of state data to design storage engines that address the throughput bottleneck of Ethereum caused by storage engine, based on the demands of Ethereum. We proposed the single level ordered log structure database (SolsDB) for storing Ethereum state data, which achieves higher read/write performance and significantly reduces read/write amplification.

Ethereum's block synchronization process interacts with the storage engine in two parts: waiting for state data to be read while processing transactions and waiting to write data when processing is complete. Currently Ethereum adopts LevelDB as storage engine, which is designed as write optimized data structure. However, its read performance does not match Ethereum's requirements. Our drilling analysis reveals that there are more mismatches between Ethereum and LevelDB: it does not take advantage of the features of state data and the read/write amplification is not suitable with Ethereum's gas mechanism. Consequently, it is not optimal for Ethereum to choose LevelDB storing data. Furthermore, these mismatches exacerbate LevelDB's read and write amplification, causing its performance degrade with the growth of data volume, which in turn limits Ethereum's block synchronization performance.

We introduce SolsDB, which fully leverages state data features to satisfy the workload requirements of the Ethereum block synchronization process. Besides, we propose a new interaction method to deliver the state data writing and reading features to the storage engine. The pass-through write feature is utilized to store state data in blocks and the file is globally ordered, which avoids write amplification resulting from compaction. Through delivering the read feature the Parser module is designed to parse the key of query, which can directly locate to the file without querying level by level, resolving the problem of read amplification. Extensive experiments were conducted to validate the performance of SolsDB. Compared with LevelDB, SolsDB improves read performance by up to 4.7x, reduces read tail latency by 68.7% to 83.3%, and reduces write amplification factor by 49.1% to 76.1%.

PVLDB Reference Format:
Hua Yang. SolsDB: Solve the Ethereum Bottleneck Cause by Storage. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at URL_TO_YOUR_ARTIFACTS.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

1 INTRODUCTION

Blockchain technology has evolved for a long time, and many implementations of blockchain systems have sprung up. For example, bitcoin[2] is designed as a distributed peer-to-peer digital currency. Ethereum[10] is designed as a decentralized blockchain platform that is capable of supporting the operation of smart contracts which can be employed to implement various businesses such as DeFi[32], IoT[12], Supply Chain[34] and so on. Although Ethereum provides more functionality, its role as the underlying platform to support various types of business has also undergone tremendous pressure. Specifically, as the underlying platform, it not only needs to provide sufficient throughput for the efficient operation of various types of business, but also needs to address the influence of the ever-increasing volume of data(Ethereum full node data size of 16TB as of November 2023[11]).

Each transaction in Ethereum is processed on the Ethereum network by determining the packed block, broadcasting it around the network, and replaying the transaction by the node that receives the block. Therefore in contrast to traditional centralized systems, Ethereum's throughput is technically determined by two factors: the performance of the consensus protocol and the performance of block synchronization. Although the performance of Ethereum's consensus protocol has been improved by changing from POW [16] to POS [25], the performance of block synchronization still limits the throughput of Ethereum.

We analyze the block synchronization process of Ethereum. The interaction with LevelDB [21] consists of two parts: 1. Transactions are executed with read-reproduce-write (similar to read modified write, but without modifying the original data) to deal with state data. 2. New state data is generated and written to LevelDB. Since the EVM executes transactions in a single thread, Ethereum's throughput is closely related to the read and write performance of LevelDB. Existing research has also demonstrated that the block synchronization bottleneck of Ethereum lies in the reading and writing of state data[22]. We analyze the data characteristics of the state data and the interaction process with LevelDB, from which we found many mismatches between Ethereum and LevelDB.

- **Read performance doesn't match Ethereum's needs.** The LSM-tree[26] structure is write-optimized to write data append-only, which improves write capability at the expense of read performance, but Ethereum equally requires high read performance.
- **Unutilized state data features.** Ethereum state data is written using hash as the key. Due to the randomness of

hash, the organization of the key by leadldb does not exploit some of the characteristics of the state data.

- **Read-write amplification is not compatible with Ethereum's gas mechanism.** Ethereum sets the amount of consumed gas[13] for various types of operations of smart contracts with the intention of controlling the use of computational resources, but the read amplification and write amplification of LevelDB makes the same operation consume inconsistent computational resources.

Based on the above findings we analyze that these mismatches exacerbate the read/write amplification problem of LevelDB and that the LevelDB read/write performance degrades severely with the growth of data volume, which in turn affects Ethereum's throughput. In this regard we utilize state data features to design SolsDB to solve the current mismatch problem.

Key challenges to address these mismatches. In order to design a storage engine that is more compatible with Ethereum, we need to analyze the Ethereum data features and design a method to deliver the features to the storage engine. Then leverage the data features to design storage engines that can address these mismatches, which in turn will solve the Ethereum throughput bottleneck caused by the storage engines.

Feature analysis. Investigate the block synchronization process of Ethereum and summarize the features of state data from the interaction procedures between MPT and LevelDB.

Feature transfer. Analyze the current methods used to interact state data with LevelDB and design new interactions to deliver the analyzed data features to the storage engine so that these features can be utilized.

Storage engine design. Leverage the analyzed data features to redesign the read operation of state data so as to minimize the number of IOs to improve query performance. Redesign the writing of state data to solve the write amplification caused by compaction. In addition, it is necessary to avoid performance degradation caused by increasing data volume.

From these mismatches, we analyze Ethereum's data features and design a storage engine that satisfies Ethereum's needs and is more in line with its design philosophy. To summarize, this paper makes the following contributions:

- We analyze and summarize the features of Ethereum state data, and design a new interaction method to efficiently transfer the analyzed data features for SolsDB to recognize and utilize them from both read and write aspects.
- We utilize data features to design a storage engine that resolves current mismatches. With the help of transferred write feature, SolsDB is able to establish the global order of files without compaction, thus solving the write amplification problem. Moreover, we design a new read operation using the delivered read feature so that each data query takes at most one I/O, which eliminates the read amplification.
- We implement the prototype of SolsDB and conduct extensive experiments with real Ethereum loads. In order to synchronize data from the main network to test the performance improvement under real workload, we modified Geth to replace the storage engine with SolsDB and modified Ethereum's block verification and other modules.

The remainder of this article is organized as follows. Section 2 introduces the background of LSM-tree and Ethereum, as well as the interaction process between Ethereum and LevelDB. Sections 3 and 4 introduce the design and implementation of SolsDB. Experiments and assessments are presented in Section5. Section6 discusses related work. Finally, the work of this paper is summarized in Section7.

2 BACKGROUND AND OBSERVATION

2.1 LSM-Tree

LSM-tree is called Log-Structure-Merge Tree, which writes data sequentially in an Append Only manner, and consists of Skiplist in memory and SSTable on disk. Different from the B+ tree, data is not globally ordered but internally ordered at each level. Therefore, an additional metadata table is maintained that records the metadata, minimum key, and maximum key of each SSTable for data query.

2.1.1 Reading process. The read operation starts with the latest data until the target key is found or does not exist. The MemTable and Immutable MemTable in memory are searched first when querying the data. If the target key is not found, the SSTable on disk is probed. Begin by scanning the metadata table that records the SSTable information and the SSTable whose recorded key range matches the target key is selected. For the selected table, query the index block to determine the data block where the target key is located, and then query bloom filter[3] to determine whether the target key exists. The query ends until the target key is found or all selected tables do not contain the target key.

Due to the overlapping key ranges of the SSTable for each Level, which leads to multiple I/Os by scanning multiple SSTables when querying cold data, i.e., read amplification.

2.1.2 Writing process. LSM-tree organizes data in file units when writing, and stops writing whenever the fixed size is filled. At this time, the MemTable is converted to Immutable MemTable, and a new MemTable is created to handle subsequent write requests. When the number of files in memory reaches the threshold, mem-compaction is triggered, which writes Immutable MemTable to disk as a new SSTable. The file is sorted internally by the key of the data, but due to the append-only writing method, the overall data organization is no longer orderly.

Therefore, in order to improve query efficiency, when the number of files at each level on the disk reaches the threshold, a sequence is established for each level through compaction. In addition to establishing the sequence, redundant data is also processed, but this also leads write amplification. compaction establishes the order within each level, but there is still an overlap in the key ranges of the SSTable across multiple levels, which leads to read amplification[24].

- **Observation 1.** Read and write amplification does not match the gas mechanism of Ethereum

Ethereum sets the amount of gas consumed for various operations of a smart contract, which is used to control the consumption of computing resources. For the same number of gas operations, the

computational resources consumed are different due to LevelDB's read and write amplification.

Take the SLOAD operation as an example, the difference in computational resources consumed can be several times if the data are at level 1 and level n, respectively. We measured the Ethereum read latency as shown in Figure 1. It was observed that there is a large difference in the average read performance and tail latency of Ethereum. In addition, some scholars have also demonstrated this problem [39].

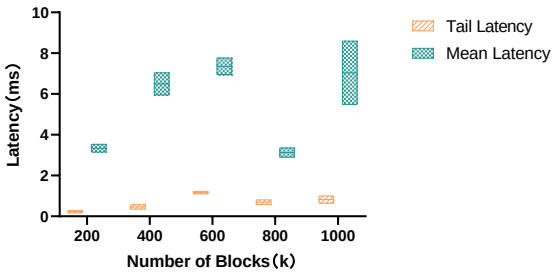

Figure 1: The read latency mean and long tail latency of state data.

2.2 Ethereum basic

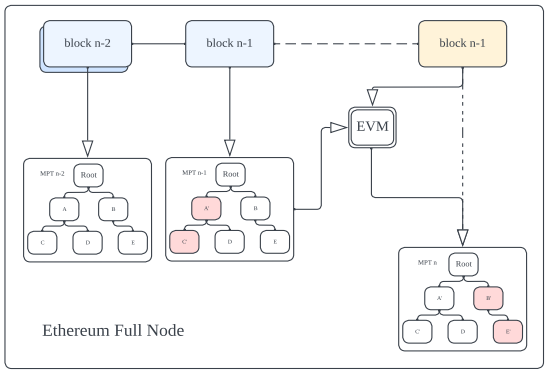

Figure 2: Ethereum state transition process.

Ethereum is a state transition system where nodes take the current world state and transactions in pending blocks as inputs and calculate a new world state as output after the checksum passes, the state transition process is shown in Figure 2. Ethereum organizes account data in The Modified Merkle Patricia Trie[36], called the world state. MPT uses the hash of the account address as the key, and the information of each account is stored in the leaf node of its path.

When the block is processed by the Full Node, EVM reads the world state data corresponding to the previous block to process the transactions in it and generate a new world state. At the time of submission, MPT performs collapse and replaces the node's child nodes with its hash, after the replacement of all child nodes is completed the node is encoded using RLP.

If the length is greater than 32byte, the hash is performed, otherwise it is written directly into its parent node in RLP encoded format. If the node has not been modified, its original hash will not be modified and will still remain in the parent node.

- **Observation 2.** Read performance is critical to Ethereum synchronization performance.

The full node first reads the account data from the previous state when processing transactions in the block. The MPT is expanded on read, and the root in the previous block header is used as the key to reinstate the world state MPT corresponding to that block. When MPT expanding, the child node data is read using the hash of the child node recorded in the parent node as the key until it reaches the leaf node.

The node executes the transaction after reading the account data, so the processing of the transaction waits for the read operation of the storage engine. As MPT maps accounts into multiple nodes, each transaction execution requires multiple read calls, so read performance is important for Ethereum synchronization performance.

2.3 Interaction between MPT and leveldb

From a holistic point of view, the Full Node synchronizes blocks in a way that MPT interacts with LevelDB in a manner similar to the COW B+ tree[30], with state data invoking LevelDB in a read-reproduce-write mode.

Ethereum is a state transition system in which each state change is triggered by a block, and different versions of the state are split in blocks, therefore blocks are the interval unit of the state data timeline, the write timeline is shown in Figure 3. The root of the MPT at each epoch of the timeline is stored in each block header.

Different from the COW B+ tree, the pointer is replaced by a hash and used as a key to interact with LevelDB.

2.3.1 Read process. The corresponding world state MPT is recovered from root when reading account data for a specific state. The root of each MPT is stored in the corresponding block header, so when reading the state belonging to a certain block, the account

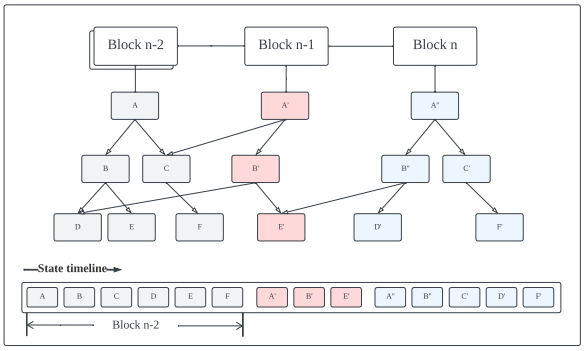

Figure 3: Ethereum state data timeline

information stored in the leaf nodes can be read by expanding the MPT using the root recorded in its block header as key. When MPT expanding, since the key of the child node is stored in the value of the parent node, the leaf nodes can be read according to the path of the account by calling **key read-value parse-key read** on each node in turn.

2.3.2 Writing process. The Full Node reads the current state for replication as it processes the block. If a node is changed, all nodes on its path recalculate the key and value (Key is the hash of the value, which is a 32byte hexadecimal number). State data is written after all transactions in the block have been processed. The modified node recalculates the key and value and writes them to LevelDB as KV pairs, so the original state data is not updated.

State data is written to LevelDB in batches, and LevelDB is called to write the data after each batch is full. Even if the last batch is not full, the storage engine will be invoked to ensure that all the state data is written.

- **Observation 3.** Unutilized state data features render both read amplification and write amplification more severely in LevelDB, affecting read and write performance.

LevelDB sorts these state data by the key of the hexadecimal number as it is received, rather than the timeline order of the state data. Due to the randomness of hash and the fact that the size of each state data is not consistent with the size of the SSTable, data from different states are mixed together. After compaction is triggered, the file is merged into different files in the lower layer.

We analyze the impact of the unutilized state data features and the randomness of the hash as follows.

Impact on writing. Ethereum's state data is written to LevelDB using hash as the key. Since hash is a random 32byte hexadecimal number and the key changes with the value, resulting in a serious overlap of key ranges of SSTables at different levels, this will frequently trigger compaction and thus exacerbate write amplification. Blocking writes while the compaction is incomplete, which in turn blocks Ethereum's block synchronization processing.

Impact on reading. Using hash as key leads to serious overlapping of key ranges between SSTables at different levels of LevelDB. At this point, for a cold account, the read of its nodes will select multiple SSTables for querying, and the performance is impacted by the read amplification of LevelDB. The nodes on the path of this account are distributed in different intervals of the timeline, and due to the randomness of the hash that causes each node to exist in a different SSTable, the cache cannot be effective at this time. Each node performs a top-down SSTable query, which aggravates the LevelDB read amplification.

3 DESIGN

3.1 Overview

Our design is divided into three parts: data features analysis, data features transfer, and storage engine design, which are explained in detail in sections 3.2, 3.3, and 3.4 respectively.

3.2 State data features

Ethereum is a state transition system, and changes in state data are triggered by blocks. We summarize the characteristics of state data as follows:

- **In units of block.** We analyze the Ethereum state data writing process, which takes place when all transactions in the block are processed, so state data is generated and written on a block basis.
- **With global orde.** Since state data is generated in blocks, state data at different intervals on the timeline of the state data is written in the order of the block number, so the state data is in units of blocks and globally in the order of the block number.
- **No update of data.** Full node reads the previous world state during block synchronization, if a node has modified it then recalculates the hash, writes the newly calculated node hash as key and the node's RLP as value to LevelDB instead of modifying the value of the original key, so the state data is not updated.

Ethereum state data interacts with LevelDB using hash as a parameter. Due to the randomness of hash the timeline sequence of state data is lost, resulting in non-utilization of data features. Consequently we need to devise a new interaction to deliver the state data features into the storage engine, so as to exploit the data features.

3.3 Features transfer

LSM-tree writes data sequentially by append only, so the write performance is excellent. Nevertheless, compaction is required because of the spatial redundancy of multiple versions of the key when updating the data and the need to establish the order between SSTables to improve the query performance.

However, Ethereum state data is not updated and there is a global order. If the timeline order of Ethereum state data can be passed into the storage engine and the intervals of the state data can be recognized, the data can be written in an append only manner without the compaction operation.

To do this, we design feature delivery for transferring the global order and timeline interval of the state data into the storage engine.

3.3.1 Writing feature transfer. According to our summarized features, the state data is written in blocks. During the block synchronization process of the Full Node, the modified data is written to LevelDB when all transactions are processed, and this block synchronization ends when writing is completed. The state data is written in batch, whenever the batch size reaches the threshold it will be written to LevelDB, and after the last batch is written even if the size does not reach the threshold it will be written to LevelDB to make sure that all the state data generated this time has been written.

In this case, we design the Turn function, which sends the current state data write completion identification to the storage engine after the state data is written, and also sends the current block number to the storage engine. In this way, the state data is transferred to the storage engine in the form of blocks.

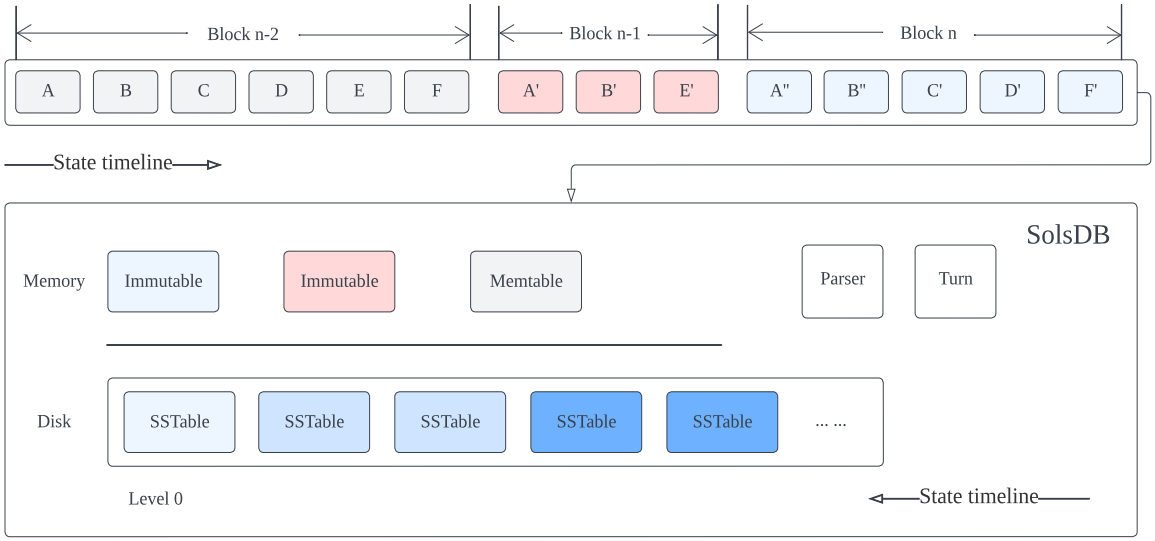

Figure 4: SolsDB architechture

As a result, the storage engine recognizes the write feature and uses the parsed block number to generate file metadata to store the state data. State data written in this way is ordered by block number among files and by hash within files.

3.3.2 Read feature transfer. In the read process of MPT, each node writes the hash computed after its RLP encoding as a key to its parent node when it is updated, and when it is read the node calls LevelDB to query the data with its key stored in the parent node. Originally, the key of the node is stored as hash, and at this point it is not possible to utilize the timeline sequence feature of the node update.

We consider that when a node is updated its data is written to LevelDB and the data file in which it resides is identified by the block number at the time of the update. Therefore we combine the block number with the recalculated hash when the node is updated and write it as the key to its parent node.

When reading the node, the storage engine is called with this key as a parameter, so that the storage engine can recognize the file where the key is located. This approach is similar to Block-LSM[5], but serves a different purpose.

Furthermore, we remove the added block number when the node is written to LevelDB as kv to reduce storage overhead, and to reduce the comparison time of redundant data when querying.

3.4 Storage engine design

After transferring the analyzed data features to the storage engine, the data features can then be exploited to devise a storage engine to solve the current mismatches.

3.4.1 Architecture. We present storage engine SolsDB consists of Parser module, Turn module, MemTable and Immutable MemTable in memory, and a single-level SSTable in disk, as detailed in the system architecture diagram Figure 4. Parser is used to parse the read incoming parameter key into block number and key, block number for generating file metadata to locate the file and Key for querying in the file. The Turn function was designed to transfer the features of the state data as units of blocks, with the block number as the parameter.

In-memory MemTable and Immutable MemTable are used to cache write data to improve write performance. The single level SSTable is used to persist the data, and the individual SSTables are identified and sorted by block number.

3.4.2 Data organization. The data is organized in SolsDB as shown in Figure 3. State data is written as a unit of block to a file, which is identified by the block number corresponding to the time when the state data is generated, and is stored in memory (for caching the written data) and disk, respectively.

Once the Ethereum state data is generated, SolsDB is able to recognize the completion of this state data write and retrieve the block number corresponding to this state data through the transfer of the write feature by the Turn function. In-memory MemTable receives the written data and is converted to Immutable MemTable after detecting the characteristics of the data passed by Turn.

After recognizing the features a new MemTable is created for writing the state data corresponding to the next block. When the number of files in memory reaches a threshold, it is written to the SSTable on disk. The block number of the Immutable MemTable is used to generate metadata for the SSTable, which is used to locate it while reading.

The order of single-level SSTable generation in the disk is consistent with the timeline order of the state data, and the order of SSTable's metadata is consistent with the order of the state data corresponding to the block number. SolsDB's organization of state data

is globally sorted at the file level, which is a satisfactory solution to the read-write amplification problem of LSM-tree.

Inside the file, the data is still sorted by key. Since the key prefix is removed when reading the feature deliver, there is no redundant block number to compare when reading inside the file, which makes reading more efficient and reduces the storage overhead.

3.5 SolsDB operations

One of the best ways to understand a storage engine is through its read and write processes, following which we introduce the interaction between the various components through read and write operations.

Algorithm 1: Write Algorithm

```
Function Write(batch)
    batch.PutMem(mdb)
    if mdb.Tag then
        Tagcompaction()
    end
return
Function putMem(mdb)
    for key := range batch do
        if key.type == keyTypeTurn then
            mdb.TurnTag()
        else
            mdb.put(key, value)
        end
    end
return
Function tagCompaction()
    db.rotateMem()
    if db.memsNum = threshold then
        db.flushMem();
    end
return
```

3.5.1 Writing process. Algorithm 1 shows the write operation. The memory component of SolsDB consists of MemTable and Immutable MemTable, and data is first written to MemTable. When the incoming identifier is recognized, it stops the writing of the current MemTable, transforms it into an Immutable MemTable, and parses out the block number passed in by the Turn method to identify the current MemTable. In this way, one file for one state data is achieved.

Then a new MemTable is created to handle subsequent write requests, and Tag Compaction is called to detect whether the current number of memory files reaches a threshold and determine determine whether the flush is to be performed. The global sequence of state data and the transfer of features in blocks allows SolsDB to establish the global order of files without compaction, which solves write amplification.

SolsDB's files are globally ordered as opposed to being sequential within each level of the LSM-tree. Since Ethereum state data is not updated, this approach realizes both the advantages of sequential writes and the advantages of queries due to global ordering.

Algorithm 2: Read Algorithm

```
Function Read(key)
    num, ikey := db.parser(key)
    if num > db.blockNum then
        db.memGet(ikey)
    else
        fd := db.locateTable(num)
        db.storageGet(fd, ikey)
    end
return
```

3.5.2 Reading process. Taking advantage of the global ordering property of SolsDB files, we redesigned the read operation to obviate the storage overhead of metadata and to enhance read performance. As shown in Algorithm 2 when processing a read request, SolsDB first invokes the Parser module to parse the incoming key into a block number and a key, which is used to locate the file where the key is stored, and the key is used to query within that file.

After parsing the key the block number is compared to the currently recorded persistent file number. If the block number is greater than the currently persisted block number, the in-memory Skiplists will be queried. If the block number is smaller than the current persistent block number, the block number will be processed to get the file metadata, and the target key will be queried inside the file after reading the file.

The read operation designed in conjunction with Parser exploits the global sequence of SolsDB files and avoids queries to metadata tables and overlapping SSTables enabling the read operation to be performed at most once for I/O, which solves the read amplification.

3.5.3 Tag Compaction. Tag Compaction detects if there are Immutable MemTable files to be written to the SSTable. When a Tag is detected during the write process, Tag Compaction is called to detect the number of files in memory. If the current number of files is less than the threshold, no action is taken. If the number of files in memory is equal to the threshold, the Immutable MemTable is selected for writing to the SSTable via a background thread.

When writing to the SSTable, the block number recorded in the current Immutable MemTable is read and file metadata is generated based on that block number. As the block number of Ethereum is globally unique and incremental, we can use it as file metadata with a slight modification. Using the generated file metadata, we create a file and write the data from Immutable MemTable to that file, and finally update SolsDB's persistent block number variable.

Tag Compaction takes care of persisting the data, so there is no write amplification.

4 IMPLEMENTATION

We implemented SolsDB in GO. Some of the data structures of LevelDB have been reused, the original compaction has been removed whereas the Parser and Turn modules have been added to receive and process Ethereum data features. Due to the transfer of write

feature, we use the block number passed in by Turn to identify the file to establish the global order of the file. Therefore, the metadata table recording SSTable information is no longer maintained. We also designed Tag Compaction for data persistence.

Further, we modified Geth to replace the original interaction to enable write, read data feature delivery. The modification of the interaction method makes the Root of MPT inconsistent with the block record in the main network, so to test the performance improvement under real load, we modified the block validation method of Geth and the connection of the Storage trie of the contract account to synchronize the data from the Ethereum mainnet.

5 EVALUATION

In this section we first describe the experimental setup and then the results of the experiment.

5.1 Experiment set up

Experiment environment. We modified it based on Geth version 1.9.19[14] to enable block synchronization from the Ethereum mainnet. Replace its storage engine from LevelDB to solsdb. We implemented the prototype of solsdb, and deployed Geth and our modified Geth on the same machine for experimental testing. The experimental machine configuration as Table 1:

Table 1: Computer Configuration.

Device	Specification
CPU	R5-3500x
Memory	DDR4 8G*2
SSD	500GB
System	Ubuntu22.04LTS

During the experimental test, the parameters for configuring LevelDB and solsdb were consistent.

Workload. In order to test SolsDB's solution to Geth's current problems, we modified Geth to be able to synchronize blocks from the Ethereum mainnet while modifying the storage engine and MPT. We use Geth and SolsDB to synchronize blocks of 5 different data sizes from the mainnet for performance evaluation: 200k, 400k, 600k, 800k, 1000k blocks. Additionally, considering that Ethereum's throughput is affected by consensus, network, and that Ethereum block processing relates to read/write operations of the storage engine, we conducted read/write tests on storage engines with different data sizes to evaluate the optimization for synchronous processing.

We fetched 50MB state data from Ethereum (this part of the data was retrieved after 100k state data to avoid the performance impact of repeated writes) for testing the write performance of the storage engine. Considering the write test under different data sizes, the written data should not be too large to avoid the impact of its size on the performance of the storage engine.Meanwhile the size of the state data should not be too small to represent the impact of compaction.

For read performance, we considered that the interaction between Ethereum and LevelDB is conducted through MPT, so we

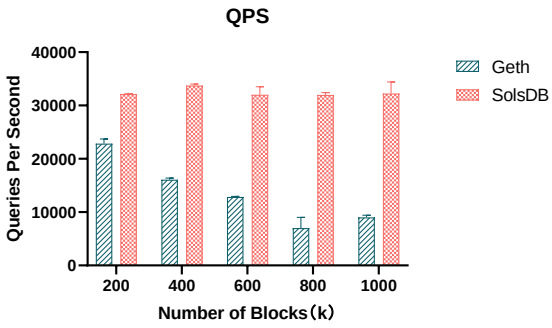

Figure 5: QPS.

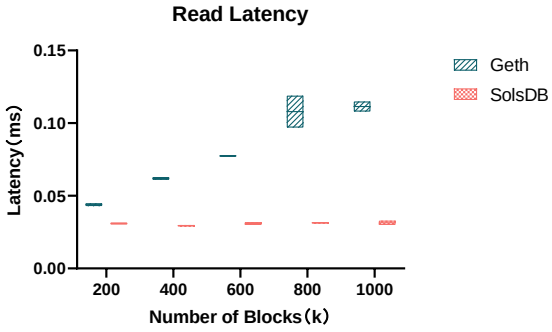

Figure 6: Read latency.

obtained 18k accounts numbers[41] and read the accounts via MPT to test the read performance of the storage engine. Each node of the MPT exists at different intervals on the timeline, so we select the root in the 200,000th block header and perform read queries with the MPT corresponding to the root. The selected account data is also the account created before the 200,000th block to ensure that the query is valid.

5.2 Experiment results

According to our analysis, those mismatches between Ethereum and LevelDB reside in three aspects: reading requirements, data feature utilization, and gas mechanism. We improve the data read performance to satisfy Ethereum demand with the help of Ethereum data features, so we test the read and write performance of SolsDB at different data sizes to evaluate the effect of the improvement on the read and write performance with real Ethereum workloads.

Regarding those mismatches between LevelDB's read and write amplification problem and Ethereum's gas mechanism, we tested the write amplification factor and read and write long tail latency during the synchronization process.

5.2.1 Synchronization performance. **Read performance.** In order to test the effect of Sols on read performance, we used 18k accounts to conduct read performance tests under different data sizes. The test results are shown in the Figure 5. We can observe that under different data sizes, the read performance of SolsDB is improved by

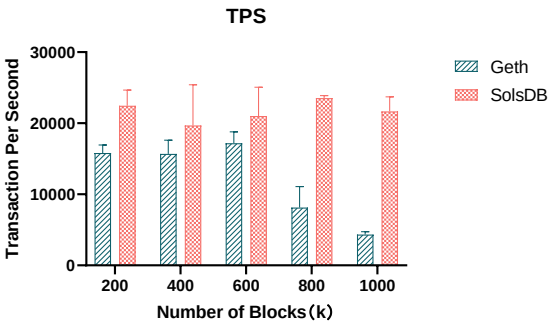

Figure 7: TPS.

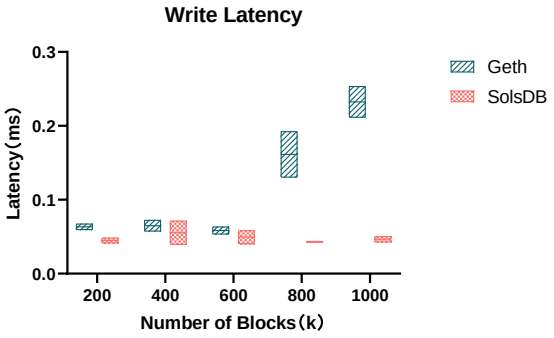

Figure 8: Write latency.

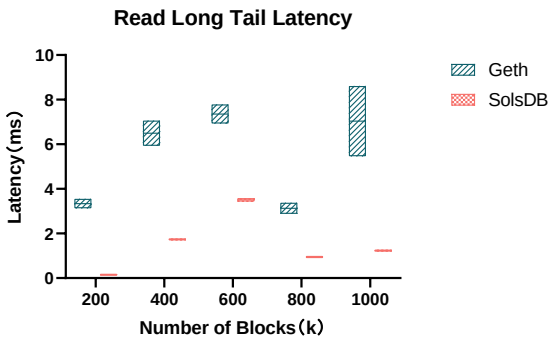

Figure 9: Read long tail latency.

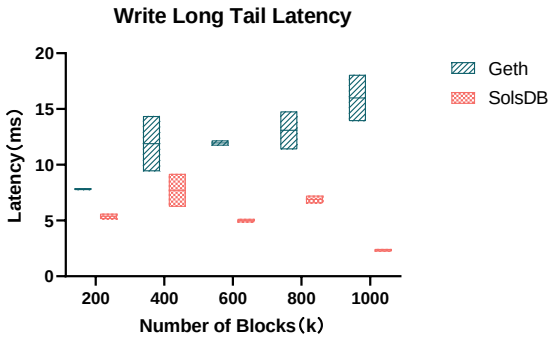

Figure 10: Write long tail latency.

44.5% to 475.2% compared to LevelDB. The impact of read latency on Ethereum synchronization performance is more intuitive. From the Figure 6, we observe that under different data scales, SolsDB's optimization of read latency ranges from 31.1% to 82.6%. As the data size increases, LevelDB's read performance decreases and read latency increases. However, Sols's read performance is relatively stable and its read latency is also relatively stable. This is due to Sols' redesign for reads, which require at most one I/O per access after locating the file via Parser.

Write performance. We tested the data write performance and write latency at different scales as shown in the Figure 8. In the Ethereum scenario, LevelDB triggers compaction more frequently as the data size increases, and compaction consumes more computing resources, blocking write operations, thus affecting its own writing performance.

We design each state data to be written to a file through Tag Compaction, thus avoiding the impact of writing performance caused by compaction. Due to the transfer of state data timeline characteristics, the overall order can be maintained without compaction, and the write performance is not affected by the data size, so the write operation will not be blocked. According to the Figure 7, the write performance of SolsDB is 25.6% to 399% higher than LevelDB.

5.2.2 Read and write amplification. **Read long tail latency** We tested the 99th percentile of read latency during the block synchronization process (representing that 99% of read latencies are less than this value) as the read long tail latency is shown in the Figure 9. Due to LevelDB's own read amplification, the number of I/O performed during each read access is different, resulting in different consumption of computing resources. Observed from Figure 9, Geth has a serious read amplification problem when reading accounts, which does not match its own designed Gas mechanism. SolsDB's read design allows each read to access the disk at most once. Experimental results also show that SolsDB' read long tail latency is 68.7% to 83.3% lower than LevelDB under different data sizes.

Write amplification. LevelDB's write amplification is caused by compaction, which not only affects the storage overhead of data, but also blocks write operations and affects write latency. Therefore, we tested the write long tail latency and write amplification factor of different data sizes when synchronizing blocks, as shown in Figure 10. Observed from the experimental results, as the data size increases, compaction has an increasing impact on writing, thereby blocking Ethereum's block synchronization.

We also tested the write amplification factor of Geth as shown inFigure 11. Since Ethereum data continues to be written, write amplification is a problem that must be solved. LevelDB's write amplification is triggered by compaction, which results in inconsistent storage resource consumption for the same data write, which also does not match Ethereum's Gas mechanism.

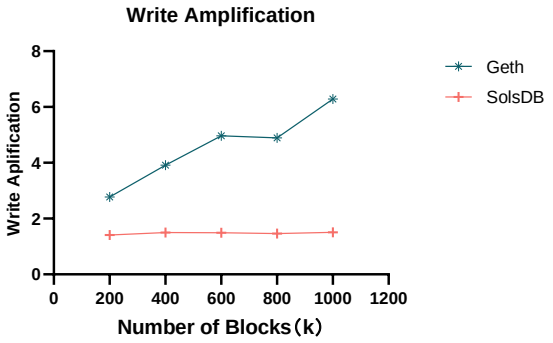

Figure 11: Write amplification coefficient.

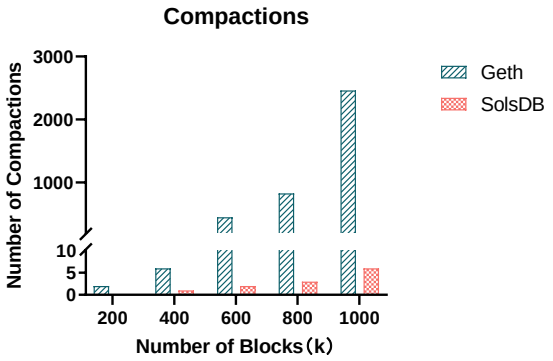

Figure 12: Number of compactions.

SolsDB's transfer of state data timeline features enables global data to be kept in order without the need for compaction, which improves read performance while also reducing write amplification, thus making write latency unaffected by data scale, thus reducing Scalability pressure on Ethereum. Experimental data shows that the write latency of Sols is 32.4% to 85.6% better than LevelDB. Since compaction is not required, the write amplification factor is reduced by 49.1% to 76.1%.

5.2.3 Number of compactions. We tested the triggering of compaction by the entire node after using Sols to store state data. The test results are as shown inFigure 12.

The compaction triggered by all data writing in Geth was tested. In addition to state data, it also includes transaction data, block data, metadata, etc. As the main body of the overall data, Geth's state data has a greater impact on the overall performance. Using Sols to organize the state data can significantly reduce the overall number of compaction triggers.

6 RELATED WORKS

This section briefly introduces related work, including the design of LSM-tree in different scenarios and solutions to the throughput bottleneck of Ethereum.

6.1 LSM-tree design in different scenarios

Many scholars have studied the read amplification[23],[17],[18] and write amplification[28], [38] problems existing in LSM-tree to better balance the trade-off between read and write performance. In addition to such general optimization solutions, there are some studies on optimization for special scenarios.

In order to solve the write amplification problem of LSM-tree in data-intensive application scenarios, Magma[20] is designed as a hybrid key-value storage engine, which combines LSM-tree with the segmented log method to significantly reduce write amplification and space amplification. question. SlimDB[17] defines semi-ordered load, uses the data characteristics in semi-ordered scenarios to design a more efficient compression index, and combines the Stepped-Merge algorithm to better balance the read-write amplification problem. Under spatially-local loads, EvenDB[15] combines spatial data partitioning with LSM-like batch I/O to achieve higher throughput and reduce write amplification. In the HTAP scenario, data of different periods are stored in different formats. LASER[31] made a key modification that allows different data layouts to be used at different levels. From purely row-oriented to purely column-oriented, a new storage engine design scheme in HTAP scenarios is given.

6.2 Ethereum throughput bottleneck

As the underlying platform, Ethereum has an increasing demand for throughput from various businesses running on Ethereum[8],[35]. Some research focuses on sharding solutions [1],[7],[40], which divide the Ethereum network to reduce the size of data stored in each node, thereby alleviating throughput pressure. The data separation storage strategy[9],[37] has also been proposed to solve the node storage pressure and divide the data into two categories, on-chain and off-chain, thereby reducing the scale of on-chain data.

There are also some works on storage engines by analyzing blockchain characteristics. Forkbase[33] proposes a forked storage engine to provide blockchain semantic support. HyperBSA[4] categorizes blockchain data and designs different storage engines based on different data characteristics. The matching between Ethereum and storage engines is an orthogonal direction to the above work. As Ethereum data continues to increase, solving these problems can fundamentally solve the throughput bottleneck.

Interaction between MPT and leveldb. Some scholars study the interaction between MPT and LevelDB and propose that MPT's amplification of account reading leads to a bottleneck in block synchronization.

Mlsm[29] proposed a method that combines LSM-tree with MPT to implement Ethereum's authentication query from within the storage engine. However, considering that the node world state in the Ethereum network should remain consistent, the LSM-tree's compaction is in each The firing times of nodes may be inconsistent, thus affecting the global consistency of the world state.

LMPT[6] proposes the separation of hot and cold accounts. It divides MPT into three parts and writes data in a manner similar to LSM-tree, thereby caching hot accounts in memory. This method improves the access efficiency of hot accounts, but also affects the authentication query of light nodes.

Rainblock[27] proposes a DSM Tree to remove IO from the critical path to improve transaction processing efficiency for the linear execution of EVM. However, this also introduces network delay, and as the data size increases, each storage node of the DSM Tree will face greater storage pressure. The above research optimized the interaction between MPT and LevelDB, but did not pay attention to the various problems caused by the storage engine.

Ethereum semantic transfer. Researching for LevelDB's write amplification, Block-LSM[5] proposed a new scheme to study Ethereum's storage bottleneck, but without re-investigating the storage engine from the point of view of the design principle of LSM-tree, instead, it passes through the order of key to avoid triggering compaction.

This approach is similar to Ethereum's processing of transaction data, by adding prefixes to differentiate the data. This solution will become less useful as the size of the data becomes larger and larger, and although it reduces the trigger of compaction, each level will also detect compaction when the number reaches the threshold, which also brings more serious metric storage overhead and affects the query efficiency.

Khipu[19] proposes to select the last 4 bytes of the key for each state data kv to establish an index with the file where it is located, write the data in the form of a log, and cache the index in memory to improve query efficiency. Although this solution improves read and write performance, it requires greater memory, which weakens the decentralized nature of Ethereum.

7 CONCLUSION

In this paper, we propose SolsDB, a storage engine designed to utilize Ethereum state data features that are more in line with the needs of Ethereum.

Specifically, we have summarized the data features that exist in Ethereum through in-depth analysis of its state data. And we have redesigned the interaction method so that these data features can be delivered to the storage engine. By exploiting these data features, SolsDB can establish the global order of data without compaction, thus avoiding the write amplification problem. In addition, using the global order of data to design a new read method, can quickly locate the data file without having to scan level by level, which solves the read amplification.

Experiments show that the read performance of SolsDB is higher than LevelDB 4.7x. SolsDB's optimization of read and write amplification is more in line with Ethereum's gas mechanism. We tested that the read tail latency was 68.7% to 83.3% better than Geth, and the write amplification was reduced by 32.4% to 85.6%.

Regarding the future, we intend to design a more efficient feature transfer method to reduce the data storage overhead while retaining the read feature delivery, and to design the storage engine for different data features of Ethereum separately, in order to realize the utilization of data features and thus improve the performance of reading and writing, and to improve the scalability of the system.

REFERENCES

[1] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. 2017. Chainspace: A Sharded Smart Contracts Platform. *ArXiv* abs/1708.03778 (2017). https://api.semanticscholar.org/CorpusID:1360317

[2] Bitcoin. 2023. https://bitcoin.org/en/.

[3] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (jul 1970), 422–426. https://doi.org/10.1145/362686.362692

[4] Xiao Chen, Kejie Zhang, Xiubo Liang, Weiwei Qiu, Zhigang Zhang, and Ding Tu. 2020. HyperBSA: A High-Performance Consortium Blockchain Storage Architecture for Massive Data. *IEEE Access* 8 (2020), 178402–178413. https://doi.org/10.1109/ACCESS.2020.3027610

[5] Zehao Chen, Bingzhe Li, Xiaojun Cai, Zhiping Jia, Zhaoyan Shen, Yi Wang, and Zili Shao. 2021. Block-LSM: An Ether-aware Block-ordered LSM-tree based Key-Value Storage Engine. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*. 25–32. https://doi.org/10.1109/ICCD53106.2021.00017

[6] Jemin Andrew Choi, Sidi Mohamed Beillahi, Peilun Li, Andreas Veneris, and Fan Long. 2022. LMPTs: Eliminating Storage Bottlenecks for Processing Blockchain Transactions. In *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 1–9. https://doi.org/10.1109/ICBC54727.2022.9805484

[7] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. 2019. Towards Scaling Blockchain Systems via Sharding. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 123–140. https://doi.org/10.1145/3299869.3319889

[8] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. 2017. BLOCKBENCH: A Framework for Analyzing Private Blockchains. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1085–1100. https://doi.org/10.1145/3035918.3064033

[9] Zhengyi Du, Xiongtao Pang, and Haifeng Qian. 2023. PartitionChain: A Scalable and Reliable Data Storage Strategy for Permissioned Blockchain. *IEEE Trans. on Knowl. and Data Eng.* 35, 4 (apr 2023), 4124–4136. https://doi.org/10.1109/TKDE.2021.3136556

[10] Ethereum. 2023. https://ethereum.org/en/.

[11] Etherscan. 2023. https://etherscan.io/.

[12] Mohamed Amine Ferrag, Makhlouf Derdour, Mithun Mukherjee, Abdelouahid Derhab, Leandros Maglaras, and Helge Janicke. 2019. Blockchain Technologies for the Internet of Things: Research Issues and Challenges. *IEEE Internet of Things Journal* 6, 2 (2019), 2188–2204. https://doi.org/10.1109/JIOT.2018.2882794

[13] Gas mechanism. 2023. https://ethereum.org/zh/developers/docs/gas/.

[14] Geth. 2023. https://github.com/ethereum/go-ethereum.

[15] Eran Gilad, Edward Bortnikov, Anastasia Braginsky, Yonatan Gottesman, Eshcar Hillel, Idit Keidar, Nurit Moscovici, and Rana Shahout. 2020. EvenDB: Optimizing Key-Value Storage for Spatial Locality. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 27, 16 pages. https://doi.org/10.1145/3342195.3387523

[16] Markus Jakobsson and Ari Juels. 1999. Proofs of Work and Bread Pudding Protocols. In *Proceedings of the IFIP TC6/TC11 Joint Working Conference on Secure Information Networks: Communications and Multimedia Security (CMS '99)*. Kluwer, B.V., NLD, 258–272.

[17] Olzhas Kaiyrakhmet, Song Yeon Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *USENIX Conference on File and Storage Technologies*. https://api.semanticscholar.org/CorpusID:73724918

[18] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 993–1005.

[19] khipu. 2019. https://github.com/khipu-io/khipu.

[20] Sarath Lakshman, Apaar Gupta, Rohan Suri, Scott Lashley, John Liang, Srinath Duvuru, and Ravi Mayuram. 2022. Magma: A High Data Density Storage Engine Used in Couchbase. *Proc. VLDB Endow.* 15, 12 (aug 2022), 3496–3508. https://doi.org/10.14778/3554821.3554839

[21] LevelDB. 2023. https://github.com/google/leveldb.

[22] Ao Li, Jemin Andrew Choi, and Fan Long. 2020. Securing Smart Contract with Runtime Validation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 438–453. https://doi.org/10.1145/3385412.3385982

[23] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. WiscKey: Separating Keys from Values in SSD-Conscious Storage. *ACM Trans. Storage* 13, 1, Article 5 (mar 2017), 28 pages. https://doi.org/10.1145/3033273

[24] Chen Luo and Michael J. Carey. 2019. LSM-Based Storage Techniques: A Survey. *The VLDB Journal* 29, 1 (jul 2019), 393–418. https://doi.org/10.1007/s00778-019-00555-y

[25] Cong T. Nguyen, Dinh Thai Hoang, Diep N. Nguyen, Dusit Niyato, Huynh Tuong Nguyen, and Eryk Dutkiewicz. 2019. Proof-of-Stake Consensus Mechanisms for Future Blockchain Networks: Fundamentals, Applications and Opportunities. *IEEE Access* 7 (2019), 85727–85745. https://doi.org/10.1109/ACCESS.2019.2925010

[26] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.* 33, 4 (jun 1996), 351–385. https://doi.org/10.1007/s002360050048

[27] Soujanya Ponnapalli, Aashaka Shah, Amy Tai, Souvik Banerjee, Vijay Chidambaram, Dahlia Malkhi, and Michael Yung Chung Wei. 2020. RainBlock: Faster Transaction Processing in Public Blockchains. In *USENIX Annual Technical Conference*. https://api.semanticscholar.org/CorpusID:224473917

[28] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 497–514. https://doi.org/10.1145/3132747.3132765

[29] Pandian Raju, Soujanya Ponnapalli, Evan Kaminsky, Gilad Oved, Zachary Keener, Vijay Chidambaram, and Ittai Abraham. 2018. MLSM: Making Authenticated Storage Faster in Ethereum. In *Proceedings of the 10th USENIX Conference on Hot Topics in Storage and File Systems* (Boston, MA, USA) *(HotStorage'18)*. USENIX Association, USA, 10.

[30] Ohad Rodeh. 2008. B-Trees, Shadowing, and Clones. *ACM Trans. Storage* 3, 4, Article 2 (feb 2008), 27 pages. https://doi.org/10.1145/1326542.1326544

[31] Hemant Saxena, Lukasz Golab, Stratos Idreos, and Ihab F. Ilyas. 2023. Real-Time LSM-Trees for HTAP Workloads. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 1208–1220. https://doi.org/10.1109/ICDE55515.2023.00097

[32] SushiSwap. 2023. https://www.sushi.com/swap.

[33] Sheng Wang, Tien Tuan Anh Dinh, Qian Lin, Zhongle Xie, Meihui Zhang, Qingchao Cai, Gang Chen, Beng Chin Ooi, and Pingcheng Ruan. 2018. Forkbase: An Efficient Storage Engine for Blockchain and Forkable Applications. *Proc. VLDB Endow.* 11, 10 (jun 2018), 1137–1150. https://doi.org/10.14778/3231751.3231762

[34] Yingli Wang, Jeong Hugh Han, and Paul Beynon-Davies. 2019. Understanding blockchain technology for future supply chains: a systematic literature review and research agenda. *Supply Chain Management: An International Journal* (2019). https://api.semanticscholar.org/CorpusID:169917106

[35] Qian Wei, Bingzhe Li, Wanli Chang, Zhiping Jia, Zhaoyan Shen, and Zili Shao. 2022. A Survey of Blockchain Data Management Systems. *ACM Trans. Embed. Comput. Syst.* 21, 3, Article 25 (may 2022), 28 pages. https://doi.org/10.1145/3502741

[36] Daniel Davis Wood. 2014. ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER. https://api.semanticscholar.org/CorpusID:4836820

[37] Yibin Xu and Yangyu Huang. 2020. Segment Blockchain: A Size Reduced Storage Mechanism for Blockchain. *IEEE Access* 8 (2020), 17434–17441. https://doi.org/10.1109/ACCESS.2020.2966464

[38] Ting Yao, Jiguang Wan, Ping Huang, Xubin He, Fei Wu, and Changsheng Xie. 2017. Building Efficient Key-Value Stores via a Lightweight Compaction Tree. *ACM Trans. Storage* 13, 4, Article 29 (nov 2017), 28 pages. https://doi.org/10.1145/3139922

[39] Huijuan Zhang, Chengxin Jin, and Hejie Cui. 2018. A Method to Predict the Performance and Storage of Executing Contract for Ethereum Consortium-Blockchain. In *Blockchain – ICBC 2018: First International Conference, Held as Part of the Services Conference Federation, SCF 2018, Seattle, WA, USA, June 25-30, 2018, Proceedings* (Seattle, WA, USA). Springer-Verlag, Berlin, Heidelberg, 63–74. https://doi.org/10.1007/978-3-319-94478-4_5

[40] Peilin Zheng, Quanqing Xu, Zibin Zheng, Zhiyuan Zhou, Ying Yan, and Hui Zhang. 2022. Meepo: Multiple Execution Environments per Organization in Sharded Consortium Blockchain. *IEEE Journal on Selected Areas in Communications* 40, 12 (2022), 3562–3574. https://doi.org/10.1109/JSAC.2022.3213326

[41] Peilin Zheng, Zibin Zheng, Jiajing Wu, and Hong-Ning Dai. 2020. XBlock-ETH: Extracting and Exploring Blockchain Data From Ethereum. *IEEE Open Journal of the Computer Society* 1 (2020), 95–106. https://doi.org/10.1109/OJCS.2020.2990458