chainkv

Yasamin Vaziri

July 2024

1 Introduction

The Log-Structure Merged tree (LSM-tree) based key-value (KV) store has been widely adopted as the storage engine for blockchain systems, such as Ethereum. In these systems, blockchain data are uniformly transformed into randomly distributed KV items for persistence. However, this transformation ignores blockchain semantics, leading to significant read/write amplification problems. As the Ethereum network scales up, the resulting massive data volume further exacerbates its storage burden.

Most existing studies focus on solutions such as sharding, data archiving, and decentralized distributed storage to alleviate this burden. However, the incompatibility between Ethereum semantics and the storage engine's characteristics has been largely ignored.

In this paper, authers present ChainKV, a new semantics-aware storage paradigm designed to improve storage management performance for the Ethereum system. ChainKV introduces several key innovations:

- **Semantic-Aware Storage Zones:** ChainKV separately stores different types of data in multiple storage zones within the KV store, reducing read/write amplification.
- **ADS Data Transformer:** A new ADS (Authenticated Data Structure) data transformer exploits data locality when persisting ADS.
- **Space Gaming Caching Policy:** A novel caching policy coordinates cache space management for two independent storage zones.
- **Node Crash Recovery:** An optional lightweight node crash recovery mechanism eliminates functional redundancy between the Ethereum protocol and the storage engine.

Experimental results indicate that ChainKV outperforms prior Ethereum systems by up to 1.99× in synchronization operations and 4.20× in query operations.

2 Background

Blockchain technology is crucial in various fields, including the Internet of Things (IoT), software engineering, and digital assets. However, public blockchains like Bitcoin and Ethereum suffer from low transaction throughput, which limits their application efficiency. This low throughput is due not only to consensus algorithms but also to the performance of the underlying storage engine.

Ethereum uses a Merkle Patricia Trie (MPT) to store account information and verify data reliability. It adopts LSM-tree based KV stores to manage historical transactions and account data. As blocks continue to be mined, read/write operations involve massive disk I/O operations, significantly degrading performance.

the analysis shows that the I/O overhead in Ethereum comes from two main aspects:

1. **Lack of Fine-Grained Data Management:** Ethereum focuses on data availability but lacks fine-grained management for data. All data are uniformly transformed into KV pairs, abandoning software semantics and reducing data manipulation efficiency.

2. **Poor Data Locality:** Persisting MPT nodes as KV pairs introduces severe read/write amplification due to the randomness of hashing, which clashes with the design of KV storage based on LSM trees.

3 ChainKV Design

To address these issues, ChainKV introduces a new storage engine with the following components:

3.1 Semantic-Aware Storage Zones

Based on the dependency and features between transactions and MPT, ChainKV separates Ethereum's non-state and state data into different storage zones. This allows independent and concurrent management, reducing I/O overhead.

3.2 Prefix MPT

ChainKV proposes Prefix MPT, an ADS data transformer that preserves data locality when persisting MPT nodes. This approach mitigates the I/O burden by improving data locality.

3.3 Space Gaming Caching Policy

A novel caching policy dynamically adjusts the cache size of the two independent storage zones to fit the upper layer access pattern, enhancing cache efficiency.

3.4 Node Crash Recovery

ChainKV includes an optional lightweight crash recovery mechanism to eliminate functional redundancy between the Ethereum protocol and the storage engine, improving overall system resilience.

4 Implementation and Evaluation

they implemented a fully functional prototype of ChainKV based on Geth, with the underlying LSM-tree storage engine based on GoLevelDB. the implementation includes approximately 6.3K lines of code in Go. Comprehensive experiments using 4.6M blocks from the real-world public Ethereum blockchain demonstrate that ChainKV achieves up to 1.99× write performance improvement and up to 4.20× read performance improvement over the original design.

5 Ethereum System

The Ethereum network functions as a fully backed-up decentralized database, with each node recording the entire transaction history as a chain of blocks. Each block contains a list of transactions and a cryptographic hash of its previous block. Ethereum nodes can be classified into two types: full nodes and light nodes. Full nodes store a complete copy of the blockchain to ensure data consistency, whereas light nodes only store block headers and lack the capability to verify data. This paper focuses on full nodes.

5.1 Workflow of Ethereum at the Data Level

When an Ethereum node receives the latest block, it processes the block's transactions as follows:

1. **Transaction Validation:** The node checks the account information of the transaction originator via the Merkle Patricia Trie (MPT) to ensure the transaction's legitimacy.

2. **Transaction Execution:** The node executes all transactions sequentially and updates the MPT to reflect the new state.

3. **State Verification:** The node compares the new MPT root with the received block to ensure data integrity.

4. **Data Persistence:** All data, including the latest block, the newly generated state information, and associated metadata, are converted into KV pairs and written to the underlying storage engine.

Ethereum data is categorized into non-state data (e.g., transactions and receipts) and state data (stored in the MPT and includes user accounts and smart contracts). Corresponding to this data composition, Ethereum supports

two main types of lookups: transaction queries and account queries. Account queries involve MPT lookups, while transaction queries trace special blocks containing the requested transactions through relevant lookup indexes. Both query types result in multiple KV pair accesses in the underlying storage engine.

5.2 Merkle Patricia Trie (MPT)

The MPT is a data structure that summarizes and verifies the existence and integrity of blockchain data. MPT nodes are categorized into three types:

- **Extension Node:** A 2-tuple $(share\_nibble, next\_node)$, where the first field is part of the hexadecimal encoding of an account address, and the second field is the hash value of its child node, which is also the key for the child node in the storage engine.
- **Branch Node:** An array of length 17, where the first 16 items correspond to hexadecimal characters (0–f) linking to child nodes, and the 17th item maintains the account balance.
- **Value Node:** A 2-tuple $(account - end, balance)$, where the first field is the last portion of the account's hexadecimal encoding, and the second field is the account balance.

5.2.1 Persistence

The MPT resides in memory, but due to its size, it must be periodically persisted into the underlying storage engine. Each MPT node is converted into a KV pair, where the value is the Recursive Length Prefix (RLP) encoding of the node's content, and the key is the hash of the value. For example, a node x is represented as $\langle Keccak(RLP(x)), RLP(x)\rangle$.

5.2.2 Query

An MPT lookup accesses all nodes along the path from the root to the leaf by matching the requested account with the nodes' fields or array indices. For instance, a query `Get(0x5a05)` accesses three nodes (x, y, and z) and returns x. Since the MPT is too large to reside entirely in memory, these lookups occur on disk, resulting in multiple KV requests. Given the maximum depth of the MPT is 64, a single lookup can trigger up to 64 KV requests.

5.3 LSM-tree based KV Store

Ethereum employs an LSM-tree based storage engine, which consists of three main components: the memory component, the disk component, and the write-ahead log (WAL).

5.3.1 Memory Component

The memory component includes two in-memory sorted skip-lists (a memtable and an immutable memtable) that store KV pairs in dictionary order.

5.3.2 Disk Component

The disk component is divided into multiple levels, with the capacity of each level increasing exponentially. Each level consists of multiple Sorted String Table (SST) files, which are composed of data blocks and meta blocks (e.g., bloom filter block, index block, and footer). All KV pairs in an SST file are sorted lexicographically by their keys.

5.3.3 Write-Ahead Log (WAL)

The WAL ensures consistency and atomicity in case of system crashes. Each KV pair is first written to the WAL file before being inserted into the memtable. The WAL is periodically deleted as data is persisted to disk. In case of a system crash, the KV store can recover inserted KV pairs from the log file.

Incoming KV pairs are initially buffered in the memtable. When the memtable is full, it is converted into an immutable memtable and written to the disk components in a flush operation. If the $L0$ level of the disk component reaches its size limit, the KV store merges all $L0$ SST files with overlapping SST files in $L1$, a process called compaction, ensuring non-overlapping key ranges in higher levels. To improve query performance, the KV store uses a Block Cache to cache recently accessed data, including data blocks, index blocks, or Bloom Filter blocks.

6 Motivation

6.1 I/O Bottleneck in Ethereum

The low throughput in previous versions of Ethereum can be attributed to two factors:

1. The Proof-of-Work (PoW) consensus defines the difficulty based on a global hash rate, limiting the block creation rate to 10–12 seconds to ensure that most miners can receive and process a block before a new one is released.

2. The number of transactions in a block is limited by the rate at which miners can process blocks.

With the transition from PoW to Proof-of-Stake (PoS), the hash rate is no longer a constraint on the block creation rate. Instead, the performance of processing blocks becomes the critical bottleneck restricting throughput.

two steps in the block processing involve heavy I/O requirements:

- **Step 1:** Verify and execute the transactions within a block. This step requires extensive MPT retrieval operations, which take up more than 70% of the total block processing time. To quantify the storage burden, they replayed over 60,000 blocks (after the height of 4.60M) in the public network and recorded the corresponding I/O statistics.
- **Step 2:** Persist all updates into the storage engine. While the LSM-tree based KV store provides good write performance, this step still accounts for 20% of the total overhead. The write performance of the storage engine is inversely proportional to the data size, making this overhead significant as data grows.

6.2 I/O Characteristics in Ethereum

The inherent characteristics of the storage engine amplify the I/O requirements from the upper transaction layer. authers identified two key observations about the incompatibility between the storage layer and the blockchain layer.

6.2.1 Observation 1: Different types of blockchain data are mixed together on disk

In Ethereum, non-state data and state data are uniformly transformed into KV pairs in the LSM-tree based KV store, and these KV items with different semantics and access patterns are managed together.

authers conducted experiments to illustrate the distribution of these data types in the storage engine. By collecting the ratio of different types of data in each SST file from about 4.6M blocks, they found that over 45% of SST files contain both non-state and state data, with most SST files dominated by non-state data and polluted by a small portion of state data

This mixture leads to performance inefficiency for two reasons:

1. **Ignoring data access patterns:** A request in Ethereum typically retrieves multiple KV pairs of the same type in succession. Mixture storage adds complexity to the query process.
2. **Magnifying search space:** Mixture storage forces queries to retrieve the entire search space, resulting in more traversed levels and compare() operations.

To illustrate the benefit of independent data management, they developed a prototype that manages non-state and state data in separate SST files.

6.2.2 Observation 2: In-memory MPT accesses show strong locality, but the same does not happen on disk

Even with separate storage of non-state and state data, querying an account remains inefficient. This inefficiency arises from the lack of semantic connection between the MPT and the storage engine.

When the MPT resides in memory, it exhibits significant spatial and temporal locality. Higher-level nodes, such as the root node, are more frequently accessed, showing high temporal locality. However, when MPT nodes are persisted to disk, the keys of KV pairs become random and discrete, conflicting with lexicographic sorting within the SST. This randomization disperses logically continuous read queries across different disk locations.

7 ChainKV Design

To solve the I/O bottleneck in Ethereum, they propose ChainKV. ChainKV mainly comprises four functional components: state separation, Prefix MPT, space gaming cache (SGC), and the lightweight recovery mechanism. Each component addresses specific aspects of the storage challenges in Ethereum.

7.1 State Separation

To isolate the management of Ethereum data based on blockchain semantics, ChainKV introduces a state separation scheme to maintain non-state data and state data individually. The storage space is divided into two parts: state zone and non-state zone, each with a memory component (State mem and Non-s mem) and a disk component (State SST zone and Non-s SST zone).

Similar to LevelDB or RocksDB, these memory buffers are organized into skip-lists to accommodate and sort KV pairs. The disk component comprises numerous SST files divided into multiple levels for persistent data storage.

- **Independent Zones:** ChainKV maintains two logically isolated abstractions, ensuring that data insertion processes are transparent to each other. Compactions only involve SST files within the same zone.
- **Efficient Query Processing:** Query requests traverse only one of the two zones based on a set of indexes, reducing the search space and read amplification.
- **Shared Version Controls:** Both zones share version controls, ensuring a consistent view of snapshots while reducing memory usage.

The state separation scheme serves three purposes:

1. Splitting the original large LSM-tree into two zones reduces read amplification and write compactions.
2. Logically isolated storage harmonizes key ranges and explores data locality advantages.
3. Isolated storage clears the way for optimizing the physical layout of state data.

7.2 Prefix MPT

To address the loss of data locality when MPT nodes are transformed into KV pairs, ChainKV introduces Prefix MPT, an optimized MPT-to-KV transformation scheme that preserves spatial and temporal locality.

- **Column Strategy:** Focuses on the continuity between parent and child nodes. The prefix of a node consists of the first field of all nodes along the path from the root to the node.
- **Row Strategy:** Explores temporal locality by assigning the same prefix to all nodes at the same depth. Nodes at the same depth are stored in the same data block.

Algorithm 1, in the main article, describes how to generate the prefix for each node. By designing the prefix values (pf_n), logically continuous MPT nodes are stored in the same data unit, significantly reducing I/O operations required for queries.

7.3 Space Gaming Cache (SGC)

To dynamically adjust the cache space based on real-time workloads, ChainKV introduces the Space Gaming Cache (SGC) strategy, inspired by Adaptive Replacement Cache (ARC).

- **Real Cache and Ghost Cache:** SGC includes a real cache for hot KV items and a virtual ghost cache that holds metadata of evicted items, hinting at potential cache hits if the real cache were larger.
- **Dynamic Adjustments:** Cache sizes are dynamically adjusted based on hits in the ghost caches, optimizing the allocation between non-state data and state data caches.

Figure 6 shows the SGC structure, which ensures efficient cache management and higher cache hit ratios by adapting to changing data access patterns.

7.4 Lightweight Node-Failure Recovery

ChainKV proposes an optional lightweight node crash recovery mechanism to replace the traditional write-ahead log (WAL), optimizing write performance without sacrificing data reliability.

- **Safe Block Tracking:** Maintains a safe block for both in-memory state memtable and Non-s memtable, which is written to disk during flush operations. The safe block indicates the progress of the latest persistent block.
- **Recovery Process:** During data recovery, the safe block is used to determine the latest synchronized block number, enabling efficient rollback and re-synchronization.

- **Performance-Security Trade-off:** Disabling WAL may increase recovery time but is a tolerable security trade-off given Ethereum's fault-tolerant nature.

8 Evaluation

This section provides an overview of the experimental setup, workloads used, evaluation results, generality discussion, and the overhead associated with the ChainKV design.

8.1 Experiment Setup

Implementation: ChainKV is prototyped based on Geth, the official Go implementation of the Ethereum protocol.

Environment: Testing is conducted on a machine with an Intel Core i7-10875 CPU @ 2.30 GHz, 16GB memory, and 1TB SN-750 NVMe SSD, running 64-bit Ubuntu 18.04 with an Ext4 file system.

Dataset: A private Ethereum environment is created to collect four datasets with different block numbers (1.6M, 2.3M, 3.4M, and 4.6M) by synchronizing historical data from the Ethereum mainnet.

8.2 Evaluation Methodology

Write Performance: Transactions are replayed to generate KV requests, and the performance of inserting these requests into the storage engine is evaluated.

Read Performance: Synthetic workloads are generated to simulate transaction queries (Tx) and account queries (Acc). The distribution of accounts is simulated using Normal Random and two Zipfian distributions.

8.3 Overall Performance

Write Performance: ChainKV significantly outperforms Geth in terms of throughput, compaction operations, and write amplification. The performance improvements range from 30.77% to 99.69%, attributed to:

- Replacement of WAL with a lightweight crash recovery mechanism.
- Separation of non-state and state data, reducing compaction operations.

Read Performance: ChainKV shows substantial improvements over Geth across all workloads. The performance improvement is highest for state data queries (Tx:Acc ratio of 0:10), with gains of 320.77%, 236.44%, and 190.13% for different distributions. For non-state data queries (Tx:Acc ratio of 10:0), improvements are 12.57%, 4.97%, and 2.52%.

8.4 Breakdown Analysis

To analyze the performance improvement of each module:

- **Geth:** Original Go-Ethereum scheme.
- **SS (State Separation):** Separates non-state and state data management.
- **SS+R:** Integrates Prefix MPT with row prefix scheme on SS.
- **SS+C:** Integrates Prefix MPT with column prefix scheme on SS.
- **ChainKV:** Full implementation with SS, column prefix scheme, lightweight crash recovery, and SGC caching policy.

Write Performance: ChainKV achieves the highest throughput improvement. State separation reduces key range overlap and compaction operations. Prefix MPT adds minimal overhead, and the lightweight recovery mechanism further improves throughput by mitigating write amplification.

Read Performance: ChainKV shows the highest throughput improvement, particularly for state data queries. The column prefix scheme performs better than the row prefix scheme due to better locality preservation.

8.5 Space Gaming Cache (SGC)

SGC improves the performance and hit rate of ChainKV over the unified LRU caching policy. Throughput improvements range from 2.72% to 6.51% for different Tx:Acc ratios.

8.6 Performance of Transaction Throughput

Experiments with real and synthetic workloads show that ChainKV greatly enhances transaction processing capabilities, achieving 4.07× and 3.89× improvements over Geth, respectively.

8.7 Generality Discussion

The techniques proposed in ChainKV are non-intrusive and applicable to other blockchain systems with fast consensus algorithms, LSM-tree based KV stores, and similar ADS. Implementations on Cosmos and Rainblock show significant performance improvements, demonstrating the generality of ChainKV.

8.8 Overhead Analysis

Space Overhead: Space overhead is minimal and acceptable, ranging from 1.83% to 18.08%.

Recovery Overhead: The recovery time ranges from 0 to 24.42 seconds, depending on the crash level. The average recovery time is about 7.59 seconds, which is acceptable.