BlockLSM

Yasmine Vaziri

July 2024

1 BlockLSM

Ethereum is a popular blockchain system used in many fields like medicine, economics, IoT, software engineering, and digital assets. For Ethereum to work correctly, every full node (which helps run the network) must store all transaction data on their local disk. As the network grows, the data size has become very large, reaching about 800 GB. Therefore, Ethereum's data storage system needs to be optimized to handle this large amount of data efficiently.

To reduce the data load on blockchain systems, some methods remove parts of the data or compress it. However, these methods can cause information loss and make data queries more complicated. They also often ignore how blockchain data interacts with the storage system underneath it.

Ethereum currently uses an LSM-tree (Log-Structured Merge-tree) based key-value storage engine, which is good for writing data. When full nodes sync data, they get the latest blocks from the network, convert the data into key-value items, and store them. This approach, however, loses some important information about the order of blocks, causing unnecessary data input/output operations.

The paper proposes a new method called Ether-aware LSM-tree based KV store (Block-LSM), which keeps the block structure in mind while storing data, to improve efficiency.

Three main challenges are addressed:

1. How to coordinate Ethereum's structure with the storage system for better efficiency?

2. How to transfer Ethereum's data structure to the storage system with minimal extra work?

3. How to ensure the queries are correct and efficient?

The solution involves:

1. Analyzing Ethereum's performance and finding that syncing data causes high I/O operations due to the LSM-tree structure.

2. Introducing a prefix-based hashing approach that uses block numbers to order data, making it easier to store and access based on the time data was added.

3. Grouping multiple blocks together to reduce the overhead of adding prefixes and maintaining memory buffers to keep queries correct and efficient.

2 Ethereum

Ethereum is a blockchain system that stores all transaction records in a chain of blocks. New transactions are grouped into blocks through a mining process and added to the end of the chain. When a full node (a computer that helps run the Ethereum network) joins, it must download and synchronize all transaction blocks from the main network before it can start offering services like data transfers and transaction verifications.

For a full node to function it needs to download new blocks from the network, must process all transactions in these blocks to generate additional data called metadata (like Receipts, Block number, Block hash, etc.) and state data (like account information). This data is crucial for quick and efficient data retrieval and for managing account information.

All this data (blocks, metadata, and state data) is converted into key-value (KV) pairs using hash functions and then stored in a database. For example, to convert a piece of metadata called ”TxLookupEntry” into a KV pair, the key is created by combining a prefix and a transaction hash, and the value is the block number where the transaction is found.

There are two main types of lookups in Ethereum:

1. Transaction Query:
    - Each transaction has a unique hash value.
    - Users can request transaction details using this hash.
    - The query process involves finding the block number, calculating the block hash, and then retrieving the transaction from the block.
2. Account Query:
    - Account data is stored in a special structure called Merkle Patricia Trie (MPT).
    - To get account information, the system follows a path from the root to the leaf of the MPT.

3 LSM-tree

Ethereum uses an LSM-tree (Log-Structured Merge-tree) for its storage, a popular method for managing key-value stores. The LSM-tree has two main parts:

1. Memory Component:
    - This part handles small and random key-value pairs by organizing them into larger, sequential batches for efficient writing.
    - It uses two in-memory sorted structures called memTable and immuTable to store the data in order.

2. Disk Component:
    - This part is divided into multiple levels, each larger than the previous.
    - Data is stored in files called Sort String Tables (SSTs).

Here's how data moves through the LSM-tree:

Data first goes into the memTable. When the memTable is full, it becomes an immuTable, which is then converted into an SST file and written to disk. This process is called flushing. When a disk level (like LN) reaches its size limit, it merges and sorts overlapping data from SST files to the next level (LN+1). This process, called compaction, ensures no overlap in key ranges within a level, keeping the data sorted.

4 Performance Analysis and Experiments

The authors conducted experiments to test the efficiency of Ethereum's LSM-tree based key-value (KV) storage engine.

They synchronized 1.6M, 2.3M, 3.4M, and 4.6M transaction blocks, which correspond to data sizes of about 10GB, 20GB, 40GB, and 80GB. They measured execution times and the number of operations in different parts of the storage system, including: Memory part ("Mem") , Logging part for crash consistency ("Log") ,Compaction part ("Comp") and Other parts ("Other")

5 Findings

Compaction Operations : As the amount of synchronized data increases, the number of compaction operations rises sharply. The proportion of time spent on compaction also increases significantly. Although Ethereum writes data sequentially, the storage engine's performance suffers due to the high overhead of compaction operations, which consume a lot of disk bandwidth and slow down data writing.

5.1 Role of Compaction in LSM-tree

- Removing invalid data (like outdated state KV items).
- Ensuring strict order of data for efficient querying.

Ethereum's append-only nature means that once data is written, it's never changed. Updates to account balances generate new KV pairs rather than modifying existing ones, avoiding update operations in the storage engine.

Problem with Key Range Overlaps:

The compaction process is burdened by the overlapping key ranges of KV items. When Ethereum data is converted to KV items, the sequence information is lost due to hashing, causing keys to be sorted through costly compaction.

Proposed Solution:

To preserve the sequence information, the paper proposes using block-number-related prefixes in the hashing process. This approach aims to eliminate the need for frequent and resource-intensive compaction operations by keeping the original order of Ethereum data.

6 Block-LSM

Block-LSM is designed to align the structure of Ethereum data with its underlying storage system. This involves three main components: Prefix-based hashing , Block group-based prefix and Attribute-oriented memory buffers

These components ensure that data insertion is efficient and maintains the sequence of Ethereum transactions, reducing the need for compaction operations that slow down the system.

6.1 Prefix-based Hashing

To preserve the order of Ethereum blocks when converting data into key-value (KV) pairs.

it uses the block number as a prefix for each KV entry instead of using timestamps, which can be irregular. Next, all data from one block (transactions, metadata, state data) is grouped together using the block number as a prefix. - This keeps related KV items together, maintaining the sequence and reducing the need for compaction.

-

6.1.1 Special Handling for TxLookupEntry

Since the block number isn't known in advance during a transaction query, the transformation for TxLookupEntry doesn't use the prefix-based hashing. This ensures query efficiency.

SSTable Alignment:

Adjusts the size of SSTable files to match the size of KV items, ensuring all data from the same block is stored together. This avoids key range overlaps and reduces compaction needs.

6.2 Block Group-based Prefix

To reduce the space overhead caused by long prefixes.

it groups multiple blocks together and use a group number as a prefix. This reduces the number of unique prefixes needed. Properly setting the memory buffer capacity ensures that all data with the same prefix is stored in the same SSTable file.

for example, instead of using four bytes for individual block numbers, groups of 100 blocks can use three bytes for the group number, saving space.

6.3 Attribute-oriented Memory Buffers

To manage special metadata (like TxLookupEntry) separately from other data.

it uses different memory buffers for special metadata and other KV items. This keeps the KV items for TxLookupEntry separate from other data, avoiding key range overlaps and reducing unnecessary compaction.

Even though TxLookupEntry items are randomly distributed, their smaller size means that the resulting compactions have minimal impact on overall performance.

7 Evaluation

7.1 Experiment Setup

- Developed a prototype of Block-LSM based on Ethereum v1.92, modifying 6300 lines of code.
- Uses LevelDB for the key-value (KV) storage engine.
- Tested on a machine with an Intel i7-10875 CPU, 16 GB RAM, and a 1TB NVMe SSD, running Ubuntu 18.04.
- Default settings: 2MB SST file size and 200MB BlockCache.
- Group size for testing: 100 blocks.

7.2 Workloads

Synchronization: Used Geth to sync transaction blocks from the Ethereum main chain to local disk, then replayed them on Block-LSM.

Data sets: Four groups of blocks with sizes of 10GB, 20GB, 40GB, and 80GB.

Read Workloads: Generated synthetic read workloads using three distributions (Skewed Latest Zipfian, Scrambled Zipfian, Normal Random) based on 4.6M Ethereum blocks.

7.3 Data Synchronization Performance

- Throughput:
  - Block-LSM consistently outperforms the original Ethereum in throughput (KV insertions per second), especially as data size increases.
  - Improvement ranges from 23.01 percent to 182.75 percent, due to reduced compaction operations.
- Compaction:

  - Block-LSM significantly reduces the number of compaction operations compared to Ethereum.
  - Even with larger data sizes, Block-LSM has far fewer compactions, improving efficiency.

- Write Amplification:
  - Block-LSM has a lower write amplification coefficient (WA Coef) than Ethereum, meaning less redundant data writing.
  - WA Coef for Block-LSM is around 2, while for Ethereum it ranges from 5.12 to 9.24.

- Execution Time:
  - Block-LSM spends less time on compaction and memory table operations, making data synchronization faster and more efficient.

7.4 Analysis

Compared various configurations:
Ethereum (Eth): Original setup.
Prefix (P): Only block number related prefix hashing.
Prefix+Align (PA): Adds SSTable alignment.
Prefix+Align+Group (PAG): Adds block group-based prefix.
Block-LSM: Full setup with all optimizations.
Results:
Block-LSM achieves the highest throughput.
Prefix-based hashing significantly reduces compaction operations.
SSTable alignment further reduces compaction but to a lesser extent.
Block group-based prefix helps save space without negative impacts.

7.5 Read Performance

Measured as Lookup Operations per Second (LOPS).

Block-LSM shows similar read performance to Ethereum across different workloads, with less than 3 percent difference.

Due to the prefix clustering data from the same block, read performance remains stable.

7.6 Overhead

Computation Overhead:

Block-LSM introduces additional computation for prefix hashing and alignment.

Despite this, overall CPU usage is slightly lower than Ethereum because Block-LSM reduces the frequency of compaction operations.

Space Overhead:

Block-LSM requires slightly more disk space due to longer keys from prefix hashing.

Space overhead ranges from 1.82 percent to 2.76 percent, which is acceptable given the performance benefits.