COLE

Yasmine Vaziri

July 2024

1 Abstract

Blockchain systems like Ethereum have high storage costs because every node needs to keep the entire blockchain data. Most of this storage cost comes from an indexing structure called the Merkle Patricia Trie (MPT), which keeps track of the data. When data is updated, MPT creates many extra copies of index nodes to ensure data can be traced back (provenance queries), which adds a lot of storage overhead.

A new idea is to use a "learned index," which is smaller and faster. However, using it directly with blockchain systems creates even more overhead due to the need to save index nodes and the large size of learned index nodes.

To solve this, the authors propose a new system called COLE. COLE stores each piece of data's history in a column format and uses learned models to index and retrieve data efficiently. It also includes strategies to optimize writing data to disk. Experiments show COLE reduces storage size by up to 94 percent and increases system performance by 1.4 to 5.4 times compared to MPT.

2 Introduction

Blockchains, used in cryptocurrencies and decentralized apps, are ledgers of transactions verified by multiple untrusted nodes. They must keep complete transaction and state data to ensure data integrity and support provenance queries (tracing data history), leading to very high storage needs. For example, Ethereum needs about 16TB of storage, growing by about 4TB annually.

The main storage issue comes from the MPT index, which keeps many old nodes after data updates. In a test, the actual transaction data was only 2.8 percent of the total storage, with the rest being the index. Therefore, a smaller, more efficient index is needed.

The "learned index" method, which uses models to predict data locations, is promising but currently doesn't support blockchain needs for data integrity and provenance. It also results in large node sizes, making it impractical.

2.1 COLE

The paper proposes COLE, a new way to store blockchain data using a column-based design. Each state is stored in columns with different versions, indexed by learned models, which makes updates efficient and reduces storage size. The system uses a log-structured merge-tree (LSM-tree) approach for handling updates, which involves storing data in memory first and then merging it to disk.

To ensure data integrity, COLE uses Merkle Hash Trees (MHT) at each storage level, combining them into a root digest that verifies all blockchain data. A new checkpoint-based asynchronous merge strategy is used to manage data writes without causing delays.

2.2 Contributions

1. COLE is the first column-based learned storage for blockchain, reducing storage costs.
2. It includes new designs optimized for writing data to disks and maintaining data integrity.
3. A novel strategy reduces delays in data writes.
4. Experiments show COLE significantly reduces storage size and improves system performance compared to traditional MPT.

In essence, COLE offers a more storage-efficient and faster alternative for managing blockchain data, ensuring it remains reliable and easy to trace.

3 Blockchain Storage Basics

3.1 defenitions

Blockchain is a chain of blocks that records transactions and maintains states.

A consensus protocol orders transactions to create a consistent view of states across different nodes.

Smart contracts execute these transactions and can store state information, each identified by a unique address.

3.2 Block Structure

Each block has a header and a body. Header includes:

H_prev_blk: Hash of the previous block.

TS: Timestamp.

$\Phi cons$: Consensus protocol data.

Htx: Root hash of current block's transactions.

$Hstate$: Root hash of the states.

$Body$ includes: Transactions, states, and their Merkle Hash Trees (MHTs).

3.3 Merkle Hash Tree (MHT)

MHT ensures data integrity by creating a tree of hash values.

Leaf nodes: Hashes of transactions (e.g., $h_1 = h(tx_1)$)

Internal nodes: Hashes of their child nodes (e.g., $h_5 = h(h_1||h_2)$)

To verify a transaction, a proof (sibling hashes along the path) is provided. Reconstructing the root hash with this proof confirms the transaction's existence.

3.4 Indexing in Blockchain

Blockchain uses an index to manage and access states efficiently. Index must Ensure integrity of blockchain states. and Support provenance queries to retrieve historical state values.

3.5 Key Functions for Blockchain Index

1. $Put(addr, value)$: Insert a state with address $addr$ and value $value$ into the current block.
2. $Get(addr)$: Return the latest value of the state at $addr$, or nil if it doesn't exist.
3. $ProvQuery(addr, [blkl, blku])$: Return historical values and proof for $addr$ within block range $[blkl, blku]$.
4. $VerifyProv(addr, [blkl, blku], value, \Phi, Hstate)$: Verify the historical values using the proof, address, block range, and root hash $Hstate$.

3.6 Ethereum's Merkle Patricia Trie (MPT)

MPT is used to index states in Ethereum. Example:

- $Get(a11e67)$: Find the latest value of $a11e67$ by traversing the nodes in the latest block.
- $ProvQuery(a11e67, [i, i])$: Retrieve value v_3 and proof for $a11e67$ in block i.
- $VerifyProv(.)$: Verify the integrity of v_3 using the proof, ensuring it matches the public digest and corresponds to $a11e67$.

4 COLE Overview

This section explains COLE, a new storage system designed to improve how blockchain systems store data. It starts with the goals and then explains how COLE works.

Design Goals:

1. Minimizing Storage Size:

   Reduce storage size using a learned index and a column-based design.

2. Supporting Blockchain Storage Requirements:

   Ensure data integrity. and support provenance queries (retrieving historical state values).

3. Efficient Disk Writes: Optimize write operations because blockchain systems are write-intensive and data needs to be stored on disk.

COLE uses a column-based design to store blockchain states. Here's how it works:

- Storing States: Each state and its historical versions are stored together. When a state is updated, the new value and its block height (version number) are added to the index.
- Indexing: Uses a "compound key" which combines the state address and the block height. This way, the updated value is stored next to the old value, avoiding the extra storage used in traditional methods.
- Reducing Write Costs: COLE uses a log-structured merge-tree (LSM-tree) to manage index storage:
  - LSM-tree: Organizes data into levels of increasing sizes.
  - First Level:Data starts here and is stored in memory.
  - When the first level is full, it merges data into the next level on disk.
  - This merging continues until the levels are not full.
- Files in Each Level:
  - Value File: Stores state data as compound key-value pairs.
  - Index File: Helps locate data during reads, using a disk-optimized learned index.
  - Merkle File: Ensures data integrity using Merkle Hash Trees (MHT).
- Converting Keys: Compound keys are converted into big integers for efficient processing. This conversion uses the binary representation of the address and block height.
- Data Integrity: Root hashes of all levels are combined to create a root digest. This digest is stored in the block header and cached in memory for fast access.
- Retrieving Data: To find a state's value, COLE searches through the levels starting from the first. The search stops when it finds a matching compound key. For the latest state value, it searches using the maximum possible block height.

5 Write Operation

Write Process:

1. Calculate a compound key using the state address and block height.
2. Insert the compound key-value pair into the in-memory level (first level).
3. When the in-memory level is full, it flushes to the first on-disk level as a sorted run.
4. This merging continues through subsequent levels when each level becomes full.

Example:
If $s10$ is inserted and fills the in-memory level, the data is moved to the first on-disk level.
If the on-disk level is also full, it merges into the next level.
This continues until all levels are balanced.

Optimization with Bloom Filters:
Bloom filters speed up read operations. They are built using state addresses to indicate if data might exist. If a Bloom filter shows a state might exist, the system performs a normal read to confirm.

6 Merkle File Construction

A Merkle file stores a special tree called an m-ary complete Merkle Hash Tree (MHT) that authenticates the data stored in a corresponding value file. This tree structure ensures data integrity by creating a hierarchical hash structure.

6.1 What is an MHT?

The bottom layer of an MHT contains hash values of each data pair (compound key-value pair). Each upper layer's hash is computed from multiple (m) hashes of the layer below it. If the last group has fewer than m hashes, it uses all available hashes.

6.2 Hash Value Definition

Bottom Layer: Each hash value hi is created by hashing the combination of a compound key Ki and its corresponding value $valuei$ using a cryptographic hash function like SHA-256.

Upper Layers: Each hash value hi is created by hashing the concatenation of m hash values from the lower layer.

6.3 Construction Process

Instead of building the MHT layer by layer, COLE builds all layers concurrently to reduce input/output (IO) costs.

Steps in the Algorithm:

1. Setup:
   - - Determine the number of layers and nodes in each layer based on the size of the value file.
   - Compute the starting points (offsets) for each layer.
   - Initialize a Merkle file with the calculated size and create buffers for each layer.
2. Processing the Input Stream:
   - For each compound key-value pair:
   - Compute its hash and add it to the bottom layer's buffer.
   - If a buffer fills up with m hashes, create an upper layer hash and add it to the next layer's buffer.
   - Flush the filled buffer to the Merkle file and update the offset.
3. Final Steps:

   Once all data is processed, handle any remaining buffers that are not full by recursively creating and flushing upper layer hashes until all data is in the Merkle file.

Example:
Imagine we have states $s1$ to $s4$ and we use a 2-ary (binary) MHT.

1. Initial State: Add $s1$ and $s2$, compute their hashes $(h1, h2)$, and derive a hash $h12$ from $h1$ and $h2$. $h1$ and $h2$ are stored in the Merkle file, and 'h12' is kept in a buffer for the next layer.
2. Adding More Data: Add $s3$ and $s4$, compute their hashes $(h3, h4)$, and derive a hash $h34$ from $h3$ and $h4$.

   $h3$ and $h4$ are stored in the Merkle file, and $h34$ is added to the buffer for the next layer. Combine $h12$ and $h34$ to get $h14$, which is stored in the Merkle file.

The Merkle file in COLE authenticates blockchain data by creating a hierarchical hash structure (MHT). It is built efficiently by processing all layers concurrently, reducing IO costs, and ensuring data integrity. This process uses buffers for each layer and constructs the MHT by hashing and flushing data systematically to the Merkle file.

7 COLE's Approach and Tradeoffs

LSM-tree-based Maintenance: COLE uses the LSM-tree (Log-Structured Merge-Tree) approach to optimize data writes and disk operations.

Tradeoff: Multiple levels in the LSM-tree can slow down read performance because searching for a state may require checking multiple levels. Additionally, merging data complicates undoing changes, so COLE doesn't support blockchain forking (creating different paths in the blockchain).

ACID Properties:

Atomicity: COLE maintains atomic updates by handling the 'root$_h ash_l ist$'$atomically. During merges, this li$

Consistency: Data consistency is maintained since updates happen atomically and the system uses a consensus protocol for concurrency control.

Integrity: -Data integrity is ensured with Merkle-based structures (MHT) for each level.

Durability: COLE uses transaction logs (Write Ahead Log) to ensure data is durable. After a crash, COLE recovers by replaying transactions from the last checkpoint, which is created when the in-memory MB-tree is flushed to disk.

Handling Long-Tail Latency:

Problem: During data writes, recursive merges can cause long delays (write stalls), leading to drops in system performance.

Solution:

Move merge operations to separate threads (asynchronous merges), but this can cause inconsistencies in blockchain nodes with different processing speeds.

COLE's Asynchronous Merge Algorithm:

1. Introduces two checkpoints (start and commit) for each merge.
2. Ensures consistent storage across all nodes by synchronizing checkpoints.
3. Makes the interval between start and commit proportional to the data size, allowing most nodes to complete merges before reaching the commit checkpoint.

7.1 Designing for Asynchronous Merges

Structure:

Each level in COLE has two groups: one for writing and one for merging.

Writing Group: Accepts new data.

Merging Group:Merges its data and moves it to the next level.

8 Write Operation with Asynchronous Merge

Algorithm Steps:

1. Insert new state values into the writing group of the in-memory level.
2. When a level is full, commit the previous merge and start a new merge in a separate thread.

3. Wait for the previous merge thread to finish if it's still running.

4. Update the $root_{hashlist}$ with the new root hash and remove obsolete hashes and runs.

5. Switch roles between the writing and merging groups.

Example:

1. Before Commit Checkpoint: Writing group is full and waiting for the merging thread to finish.

2. During Commit Checkpoint: New root hash is added, and old runs are removed.

3. After Switching Roles: A new merge thread starts, and future writes go to the new writing group.

8.1 soundness Analysis

Ensuring Synchronization: The $root_{hashlist}$ update (and hence $Hstate$) is done outside the asynchronous merge thread, making it fully synchronous and consistent.

Proportional Interval:

The interval between start and commit checkpoints is proportional to the size of the data being merged, reducing the chance of delays.

This ensures that COLE maintains consistent blockchain states across nodes and minimizes delays caused by merge operations.

9 Read Operations of COLE

This section explains how COLE handles read operations, specifically the "get" query for the latest state value and the "provenance" query for historical state values and their verification.

9.1 Get Query

The "get" query retrieves the latest value of a state. COLE performs this search starting from the smallest level and moves to larger levels until the value is found.

Algorithm Steps:

1. Prepare Compound Key:

   Create a special compound key Kq using the state address and the maximum integer (max_int) to ensure the latest value is found.

2. Search in In-Memory Level ($L0$):

   Search both the writing and merging groups in the in-memory level ($L0$). These groups hold the most recent data (Lines 3 to 5).

3. Search in On-Disk Levels:

   For each on-disk level, search the committed runs in the writing group first, then the merging group (Lines 6 to 11). This ensures that the freshest data is found first.

4. Skip Uncommitted Runs:

   Skip any runs that have not been committed to avoid reading incomplete data.

5. Return Value: The search stops and returns the value as soon as a matching state is found.

9.2 Detailed Search in On-Disk Run (Algorithm 7):

1. Check Bloom Filter: If the queried address is not in the bloom filter of the run, skip this run to save time (Line 2).

2. Use Index File: Use the index file (FI) to locate the compound key Kq.

   Start from the top layer of models in the index file and find the model covering Kq using binary search (Line 4).

   Perform a recursive query on models in subsequent layers from top to bottom (Lines 5 to 7).

3. Locate State Value: Use the final model to locate the state value in the value file (FV) (Line 8).

9.3 Function QueryModel

1. Predict Position: If the model covers Kq, predict the position $pos_pred$of the data. Compute the predicted page ID using the error bound (2ϵ) and fetch the corresponding page.

2. Binary Search: Perform a binary search in the fetched page to locate the exact data (Line 19).

10 Provenance Query

A provenance query retrieves historical state values within a specified block height range and provides proofs to verify the results.

1. Prepare Compound Keys: Compute two boundary compound keys Kl and Ku for the queried address and block height range. These keys are adjusted to ensure no valid results are omitted.

2. Search in In-Memory Level ($L0$): Similar to the get query, search both the writing and merging groups. Additionally, include Merkle paths in the proof for verification.

3. Search in On-Disk Levels:

   Use Kl as the search key to find the first result in each run.

   Scan the value file sequentially until a state beyond Ku is reached.

   Compute Merkle proofs for the first and last results' positions in each run.

4. Compute Merkle Path:

   Traverse the Merkle Hash Tree (MHT) from bottom to top to compute the Merkle path for each position.

11 Verification

1. Reconstruct Root Hashes: Use the results and their Merkle proofs to reconstruct the MB-tree and run root hashes.

2. Verify Root Digest: Use the reconstructed root hashes to verify the state's root digest against the published one ('Hstate') in the block header.

3. Check Boundary Results: Ensure the boundary results fall within the compound key range $[Kl, Ku]$ to confirm no results are missing.

12 Complexity Analysis

In this section, we analyze the complexity of COLE in terms of storage, memory usage, and IO (Input/Output) costs. We compare COLE with MPT (Merkle Patricia Trie) and COLE with asynchronous merge.

Assumptions:

- n: Total number of historical values.
- T: Level size ratio.
- B: Capacity of the in-memory level.
- m: COLE's Merkle Hash Tree (MHT) fanout.

12.1 Storage Size

1. MPT Storage: MPT duplicates nodes for each insertion. and the storage size is $O(n * dMPT)$ where dMPT is the height of the MPT.

2. COLE Storage: COLE eliminates node duplication. and the storage size is $O(n)$.

12.2 Write IO Cost

1. MPT Write Cost: $O(dMPT)$ for writing nodes in the update path.
2. COLE Write Cost: $O(dCOLE)$ in the worst case when all levels are merged.

   Amortized cost for level merge: $O(1)$

   Number of levels: $dCOLE = logT(Bn * T/(T-1))$.

   dCOLE is logarithmic to n and typically smaller than dMPT.

12.3 Write Tail Latency

1. MPT Latency: Constant cost due to no write stall.
2. COLE Latency: Worst case: Requires merging all levels, leading to $O(n)$ operations.

   Asynchronous merge: Removes write stall, reducing tail latency to $O(1)$.

12.4 Write Memory Footprint

1. MPT Memory: Constant cost since update nodes are computed on the fly and removed after being written to disk.
2. COLE Memory:

   Merging the largest level: $O(T)$memory.

   Model construction: Constant memory.

   Constructing Merkle file: $O(m * dCOLE)$.

   Total memory during write: $O(T + m * dCOLE)$.
3. COLE with Asynchronous Merge:

   Each level can have a merging thread.

   Memory requirement: $O(T * dCOLE + m * dCOLE^2)$.

12.5 Read Operations Costs

1. Get Query IO Cost:

   MPT: $O(dMPT)$.

   COLE: $O(T * dCOLE * Cmodel)$.
2. Provenance Query IO Cost: MPT:$O(dMPT)$

   COLE: $O(m * dCOLE^2)$for generating Merkle proof.
3. Proof Size for Provenance Query:

   MPT: $O(dMPT)$

   COLE:$O(m * dCOLE^2)$

13 Summary

Storage Size: COLE achieves linear storage size $O(n)$ compared to MPT's $O(n * dMPT)$

Write IO Cost: COLE's write cost is logarithmic to n and typically smaller than MPT.

Write Tail Latency: Asynchronous merge in COLE significantly reduces tail latency.

Write Memory Footprint: COLE requires more memory during writes, especially with asynchronous merge.

Read Operations: COLE's read costs depend on the number of levels and the efficiency of the learned index.

This complexity analysis demonstrates the advantages of COLE in terms of storage efficiency and optimized IO operations, while also highlighting the trade-offs in memory usage and potential write latencies.