COLE: A Column-based Learned Storage for Blockchain Systems (Technical Report)

Ce Zhang*, Cheng Xu*, Haibo Hu†, Jianliang Xu*
*Hong Kong Baptist University †Hong Kong Polytechnic University

Abstract

Blockchain systems suffer from high storage costs as every node needs to store and maintain the entire blockchain data. After investigating Ethereum's storage, we find that the storage cost mostly comes from the index, i.e., Merkle Patricia Trie (MPT). To support provenance queries, MPT persists the index nodes during the data update, which adds too much storage overhead. To reduce the storage size, an initial idea is to leverage the emerging learned index technique, which has been shown to have a smaller index size and more efficient query performance. However, directly applying it to the blockchain storage results in even higher overhead owing to the requirement of persisting index nodes and the learned index's large node size. To tackle this, we propose COLE, a novel column-based learned storage for blockchain systems. We follow the column-based database design to contiguously store each state's historical values, which are indexed by learned models to facilitate efficient data retrieval and provenance queries. We develop a series of write-optimized strategies to realize COLE in disk environments. Extensive experiments are conducted to validate the performance of the proposed COLE system. Compared with MPT, COLE reduces the storage size by up to 94% while improving the system throughput by 1.4×-5.4×.

1 Introduction

Blockchain, as the backbone of cryptocurrencies and decentralized applications [38,52], is an immutable ledger built on a set of transactions agreed upon by untrusted nodes. It employs cryptographic hash chains and consensus protocols for data integrity. Users can retrieve historical data from blockchain nodes with integrity assurance, also known as provenance queries. However, all nodes are required to store the complete transactions and ledger states, leading to amplified storage expenses, particularly as the blockchain continues to grow. For example, the Ethereum blockchain requires about 16TB storage as of December 2023, with an annual growth of around 4TB [1]. This storage requirement may compel the resource-limited nodes to retain only the data of a few recent blocks, which restricts the ability to support data provenance. The nodes that maintain the complete data may also leave the network due to the rapidly increasing storage size, which potentially affects system security.

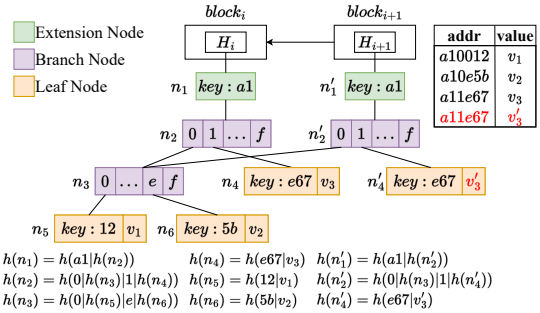

Figure 1: An Example of Merkle Patricia Trie

To tackle the storage issue, we investigate Ethereum's index, Merkle Patricia Trie (MPT), to identify the storage bottleneck. MPT combines Patricia Trie with Merkle Hash Tree (MHT) [37] to ensure data integrity. During data updates, its index nodes are persisted to support provenance queries. Figure 1 shows an example of an MPT storing three state addresses across two blocks. Each node is augmented with a digest from its content and child nodes (e.g., $h(n_1) = h(a1|h(n_2))$). The root hash secures data integrity through the collision-resistance of the cryptographic hash function and the hierarchical structure. With each new block, MPT retains obsolete nodes from the preceding block. For example, in block $i+1$, updating address $a11e67$ with v'_3 introduces new nodes n'_1, n'_2, n'_4, while old nodes n_1, n_2, n_4 endure. This setup allows historical data retrieval from any block (e.g., for address $a11e67$ in block i, value v_3 is retrieved by traversing nodes n_1, n_2, and n_4).

However, this approach adds too much storage overhead due to duplicating nodes along the update path (e.g., n_1, n_2, n_4 and n'_1, n'_2, n'_4 in Figure 1). Consequently, most storage overhead comes from the index rather than the underlying data. In a preliminary experiment with 10 million transactions under the SmallBank workload [17], we observed that the underlying data contributes only 2.8% of the total storage. Thus, a more compact index supporting data integrity and provenance queries is imperative.

Recently, a novel indexing technique, learned index [15,20, 26, 54], has emerged and shows notably smaller index size and faster query speed. The improved performance comes from the substitution of the directing keys in index nodes with a learned model. For instance, consider a key-value database with linear key distribution: $(1, v_1), (2, v_2), \cdots, (n, v_n)$. In a

traditional B+-tree with fanout f, this leads to $O(\frac{n}{f})$ nodes and $O(\log_f n)$ levels, resulting in $O(n)$ storage costs and $O(\log_f n \cdot \log_2 f)$ query times. Conversely, using a simple linear model $y = x$ enables accurate data positioning with just $O(1)$ storage and $O(1)$ query times. Although this example may not perfectly reflect real-world applications, it highlights that the learned index outperforms traditional indexes significantly when the model effectively learns the data.

In view of the advantages of the learned index, one may want to apply it to blockchain storage to improve performance. However, the current learned indexes do not support both data integrity and provenance queries required by blockchain systems. A naive approach is to combine the learned index with MHT [37] and make the index nodes persistent, as in MPT. Nonetheless, this is not feasible due to the larger node size of the learned index. The fanout of such a node is mainly dictated by data distribution. In favorable cases, only a few models are needed to index data, leading to a node fanout comparable to data magnitude. Thus, persisting learned index nodes might incur even higher storage overhead than MPT. Our evaluation in Section 8 shows that a learned index with persistent nodes is 5× to 31× larger than MPT. Furthermore, as blockchain systems require durable disk-based storage and often involve frequent data updates, the learned index should be optimized for both disk and write operations. Therefore, a blockchain-friendly learned index needs to be proposed.

In this paper, we propose COLE, a novel column-based learned storage for blockchain systems that overcomes the limitations of current learned indexes and supports provenance queries. The key challenge in adapting learned indexes to blockchains is the need for node persistence, which may lead to substantial storage overhead. COLE tackles this issue with an innovative *column-based* design, inspired by column-based databases [4, 36]. In this design, each ledger state is treated as a "column", with different versions of a state stored contiguously and indexed using *learned models* within *the latest block's* index. This enables efficient data updates as append operations with associated version numbers (i.e., state's block heights). Moreover, historical data queries no longer traverse previous block indexes, but utilize the learned index in the most recent block. The column-based design also simplifies model learning and reduces disk IOs.

To handle frequent data updates and enhance write efficiency in COLE, we propose adopting the *log-structured merge-tree* (LSM-tree) [33, 41] maintenance approach to manage the learned models. This involves inserting updates into an in-memory index before merging them into on-disk levels that grow exponentially. For each on-disk level, we design a disk-optimized learned model that can be constructed in a *streaming* way, which enables efficient data retrieval with minimal IO cost. To guarantee data integrity, we construct an m-ary complete MHT for the blockchain data in each on-disk level. The root hashes of the in-memory index and all MHTs combine to create a root digest that attests to the entire blockchain data. However, recursive merges during write operations can lead to long-tail latency in the LSM-tree approach. To alleviate this issue, we further develop a novel checkpoint-based asynchronous merge strategy to ensure the synchronization of the storage among blockchain nodes.

To summarize, this paper makes the following contributions:

- To the best of our knowledge, COLE is the first column-based learned storage that combines learned models with the column-based design to reduce storage costs for blockchain systems.
- We propose novel write-optimized and disk-optimized designs to store blockchain data, learned models, and Merkle files for realizing COLE.
- We develop a new checkpoint-based asynchronous merge strategy to address the long-tail latency problem for data writes in COLE.
- We conduct extensive experiments to evaluate COLE's performance. The results show that compared with MPT, COLE reduces storage size by up to 94% and improves system throughput by 1.4×-5.4×. Additionally, the proposed asynchronous merge decreases long-tail latency by 1-2 orders of magnitude while maintaining a comparable storage size.

The rest of the paper is organized as follows. We present some preliminaries about blockchain storage in Section 2. Section 3 gives a system overview of COLE. Section 4 designs the write operation of COLE, followed by an asynchronous merge strategy in Section 5. Section 6 describes the read operations of COLE. Section 7 presents a complexity analysis. The experimental evaluation results are shown in Section 8. Section 9 discusses the related work. Finally, we conclude our paper in Section 10.

2 Blockchain Storage Basics

In this section, we give some necessary preliminaries to introduce the proposed COLE. Blockchain is a chain of blocks that maintains a set of states and records the transactions that modify these states. To establish a consistent view of the states among mutually untrusted blockchain nodes, a consensus protocol is utilized to globally order the transactions [7, 38, 45]. The transaction's execution program is known as *smart contract*. A smart contract can store states, each of which is identified by a state address $addr$. In Ethereum [52], both the state address $addr$ and the state value $value$ are fixed-sized strings. Figure 2 shows an example of the block data structure. The header of a block consists of (i) H_{prev_blk}, the hash of the previous block; (ii) TS, the timestamp; (iii) π_{cons}, the consensus protocol related data; (iv) H_{tx}, the root digest of the transactions in the current block; (v) H_{state}, the root digest of the states. The block body includes the transactions, states, and their corresponding Merkle Hash Tree (MHTs).

MHT is a prevalent hierarchical structure to ensure data

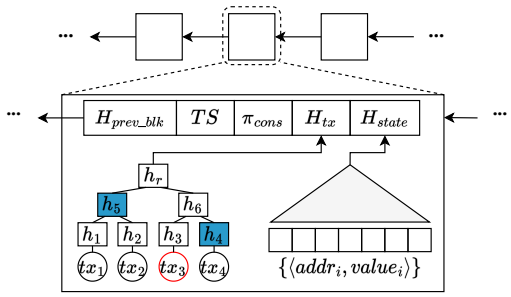

Figure 2: Block Data Structure

integrity [37]. In the context of blockchain, MHT is built for the transactions of each block and the ledger states. Figure 2 shows an example of an MHT of a block's transactions. The leaf nodes are the hash values of the transactions (e.g., $h_1 = h(tx_1)$). The internal nodes are the hash values of their child nodes (e.g., $h_5 = h(h_1||h_2)$). MHT enables the proof of existence for a given transaction. For example, to prove tx_3, the sibling hashes along the search path (i.e., h_4 and h_5, shaded in Figure 2) are returned as the proof. One can verify tx_3 by reconstructing the root hash using the proof (i.e., $h(h_5||h(h(tx_3)||h_4))$) and comparing it with the one in the block header (i.e., H_{tx}). Apart from being used in the blockchain, MHT has also been extended to database indexes to support result integrity verification for different queries. For example, MHT has been extended to Merkle B+-tree (MB-tree) by combining the Merkle structure with B+-tree, to support trustworthy queries in relational databases [29].

The blockchain storage uses an index to efficiently maintain and access the states [50, 52]. Besides the write and read operations that a normal index supports, the index of the blockchain storage should also fulfill the two requirements we mentioned before: (i) ensuring the *integrity* of the indexed blockchain states, (ii) supporting *provenance queries* that enable blockchain users to retrieve historical state values with integrity assurance. With these requirements, the index of the blockchain storage should support the following functions:

- `Put`$(addr, value)$: insert the state with the address $addr$ and the value $value$ to the current block;
- `Get`$(addr)$: return the *latest* value of the state at address $addr$ if it exists, or returns *nil* otherwise;
- `ProvQuery`$(addr, [blk_l, blk_u])$: return the provenance query results $\{value\}$ and a proof π, given the address $addr$ and the block height range $[blk_l, blk_u]$;
- `VerifyProv`$(addr, [blk_l, blk_u], \{value\}, \pi, H_{state})$: verify the provenance query results $\{value\}$ w.r.t. the address, the block height range, the proof, and H_{state}, where H_{state} is the root digest of the states.

Ethereum employs Merkle Patricia Trie (MPT) to index blockchain states. In Section 1, we have shown how MPT implements `Put`$(\cdot)$ and `ProvQuery`$(\cdot)$ using Figure 1 and the address $a11e67$. We now explain the other two functions using the same example. `Get`$(a11e67)$ finds $a11e67$'s latest value v'_3 by traversing n'_1, n'_2, n'_4 under the latest block $i+$

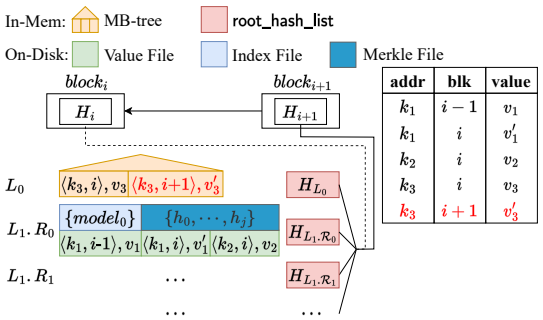

Figure 3: Overview of COLE

1. After `ProvQuery`$(a11e67, [i,i])$ gets v_3 and the proof $\pi = \{n_1, n_2, n_4, h(n_3)\}$ in block i, `VerifyProv`$(\cdot)$ is used to verify the integrity of v_3 by reconstructing the root digest using the nodes from n_4 to n_1 in π and checks whether the reconstructed one matches the public digest H_i in block i and whether the search path in π corresponds to the address $a11e67$.

3 COLE Overview

This section presents COLE, our proposed column-based learned storage for blockchain systems. We first give the design goals and then show how COLE achieves these goals.

3.1 Design Goals

We aim to achieve the following design goals for COLE:

- **Minimizing storage size.** To scale up the blockchain system, it is important to reduce the storage size by leveraging the learned index and column-based design.
- **Supporting the requirements of blockchain storage.** As blockchain storage, it should ensure data integrity and support provenance queries as mentioned in Section 2.
- **Achieving efficient writes in a disk environment.** Since blockchain is write-intensive and all data needs to be preserved on disk, the system should be write-optimized and disk-optimized to achieve better performance.

3.2 Design Overview

Figure 3 shows the overview of COLE. Following the column-based design [4, 36], we adopt an analogy between blockchain states and database columns. Each state's historical versions are contiguously stored in the index of the latest block. When a state is updated in a new block, the state and its version number (i.e., block height) are appended to the index where all of the state's historical versions are stored. For indexing historical state values, we use a *compound key* $\mathcal{K}$ in the form of $\langle addr, blk\rangle$, where blk is the block height when the value of $addr$ was updated. In Figure 3, when block $i+1$ updates the state at address k_3 (highlighted in red), a new compound key of k_3, $\mathcal{K}'_3 \leftarrow \langle k_3, i+1\rangle$, is created. Then, the updated value v'_3 indexed by $\mathcal{K}'_3$ is inserted into COLE. With the column-based design, v'_3 is stored next to k_3's old version v_3. Compared with

the MPT in Figure 1, the cumbersome node duplication along the update path (e.g., n_1, n_2, n_4 and n'_1, n'_2, n'_4) is avoided to save the storage overhead.

To mitigate the high write cost associated with learned models for indexing blockchain data in a column-based design, we propose using the LSM-tree maintenance strategy in COLE. It structures index storage into levels of exponentially increasing sizes. New data is initially added to the first level. When the level reaches its pre-defined maximum capacity, all the data in that level is merged into a sorted run in the next level. This merge operation can occur recursively until the capacity requirement is no longer violated. The first level, often highly dynamic, is typically stored in memory, while other levels reside on disk. COLE employs Merkle B+-tree (MB-tree) [29] for the first level and disk-optimized learned indexes for subsequent levels. We choose MB-tree over MPT for the in-memory level due to its better efficiency in compacting data into sorted runs and flushing them to the first on-disk level.

Each on-disk level contains a fixed number of sorted runs, each of which is associated with a value file, an index file, and a Merkle file:

- **Value file** stores blockchain states as compound key-value pairs, which are ordered by their compound keys to facilitate the learned index.
- **Index file** helps locate blockchain states in the value file during read operations. It uses a disk-optimized learned index, inspired by PGM-index [20], for efficient data retrieval with minimal IO cost.
- **Merkle file** authenticates the data stored in the value file. It is an m-ary complete MHT built on the compound key-value pairs.

Note that since the model construction and utilization require numerical data types, we convert a compound key into a big integer using the binary representation of the address and the block height. For example, given a compound key $\mathcal{K} \leftarrow \langle addr, blk \rangle$, its big integer is computed as $binary(addr) \times 2^{64} + blk$, where blk is a 64-bit value. Moreover, to ensure data integrity, root hashes of both the in-memory MB-tree and the Merkle files of each on-disk run are combined to create a root_hash_list. The root digest of states, stored in the block header, is computed from this list. This list is cached in memory to expedite root digest computation.

With this design, to retrieve the state value of address $addr_q$ at a block height blk_q, a compound key $\mathcal{K}_q \leftarrow \langle addr_q, blk_q \rangle$ is employed. The process entails a level-wise search within COLE, initiated from the first level. The MB-tree or the learned indexes in other levels are traversed. The search ceases upon encountering a compound key $\mathcal{K}_r \leftarrow \langle addr_r, blk_r \rangle$ where $addr_r = addr_q$ and $blk_r \leq blk_q$, at which point the corresponding value is returned. For retrieving the latest value of a state, the procedure remains similar but with the search key set to $\langle addr_q, max\_int \rangle$, where max_int is the maximum integer. That is, the search is stopped as long as a state value with the queried address $addr_q$ is found.

Algorithm 1: Write Algorithm

```
1  Function Put (addr, value)
       Input: State address addr, value value
2      blk ← current block height; K ← ⟨addr, blk⟩;
3      Insert ⟨K, value⟩ into the MB-tree in L_0;
4      if L_0 contains B compound key-value pairs then
5          Flush the leaf nodes in L_0 to L_1 as a sorted run;
6          Generate files F_V, F_I, F_H for this run;
7          i ← 1;
8          while L_i contains T runs do
9              Sort-merge all the runs in L_i to L_{i+1} as a new run;
10             Generate files F_V, F_I, F_H for the new run;
11             Remove all the runs in L_i;
12             i ← i + 1;
13     Update H_state when finalizing the current block;
```

4 Write Operation of COLE

We now detail the write operation of COLE. As mentioned in Section 3.2, COLE organizes the storage using an LSM-tree, which consists of an in-memory level and multiple on-disk levels. The in-memory level has a capacity of B states in the form of compound key-value pairs. Once this capacity is reached, the in-memory level is flushed to the disk as a sorted run. Similarly, when the first on-disk level reaches its capacity of T sorted runs, they are merged into a new run in the next level. This merging process continues for subsequent disk levels, with the size of each run growing exponentially with a ratio of T. That is, level i has a maximum of $B \cdot T^i$ states.

Algorithm 1 shows COLE's write operation. It starts by calculating a compound key for the state using the address and the current block height (Line 2). The compound key-value pair is inserted into the in-memory level L_0 indexed by the MB-tree (Line 3). As L_0 fills up, it is flushed to the first on-disk level L_1 as a sorted run (Line 5). The value file $\mathcal{F}_V$ is generated by scanning compound key-value pairs in the MB-tree's leaf nodes (Line 6). At the same time, the index file $\mathcal{F}_I$ and the Merkle file $\mathcal{F}_H$ are constructed in a *streaming* manner (see Section 4.1, Section 4.2 for details). When on-disk level L_i fills up (i.e., with T runs), all the runs in L_i are merge-sorted as a new run in the next level L_{i+1}, with corresponding three files generated (Lines 8 to 11). This level-merge process continues recursively until a level does not fill up. The blockchain's state root digest H_{state} is computed by hashing the concatenation of the root hash of L_0's MB-tree and root hashes of runs in other levels, stored in root_hash_list, when finalizing the current block (Line 13).

Example. *Figure 4 shows an example of the insertion of s_{10}. For clarity, we show only the states and the value files but omit the index files and Merkle files. Assume $B = 2$ and $T = 3$. The sizes of the runs in L_1 and L_2 are 2 and 6, respectively. After s_{10} is inserted into in-memory level L_0, the level is full*

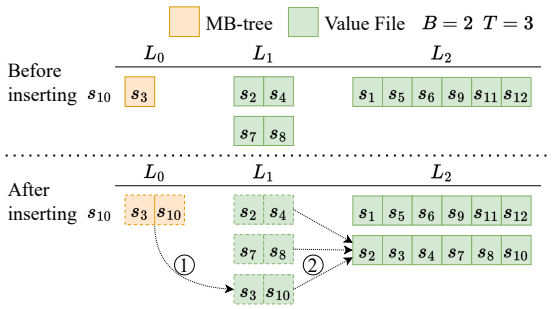

Figure 4: An Example of Write Operation

and its states are flushed to L_1 as a sorted run (step ①). This incurs L_1 reaching the maximum number of runs. Thus, all the runs in L_1 are next sort-merged as a new run, placed in L_2 (step ②). Finally, L_0 and L_1 are empty and L_2 has two runs, each of which contains six states.

A common optimization technique to speed up read operations is to integrate a Bloom filter into the in-memory MB-tree and each run in the on-disk levels. We incorporate the Bloom filter into COLE with careful consideration. First, they should be built upon the addresses of the underlying states rather than their compound keys to facilitate read operations. Second, since the Bloom filters may produce false positives, if they indicate that an address exists, we further resort to the normal read process of the corresponding MB-tree or the disk run to ensure the search correctness. We will elaborate on their usage during the read operation in Section 6. Moreover, the Bloom filters should be incorporated alongside the root hashes of each run when computing the states' root digest. This is needed to verify the result integrity during provenance queries.

4.1 Index File Construction

An index file consists of the models that can be used to locate the positions of the states' compound keys in the value file. Inspired by PGM-index [20], we start by defining an ε-bounded piecewise linear model (or *model* for short) as follows.

Definition 1 (ε-Bounded Piecewise Linear Model)**.** *The model is a tuple of $\mathcal{M} = \langle sl, ic, k_{min}, p_{max} \rangle$, where sl and ic are the slope and intercept of the linear model, k_{min} is the first key in the model, and p_{max} is the last position of the data covered by the model.*

Given a model, one can predict a compound key $\mathcal{K}$'s position p_{real} in a file, if $\mathcal{K} \geq k_{min}$. The predicted position p_{pred} is calculated as $p_{pred} = \min(\mathcal{K} \cdot sl + ic, p_{max})$, which satisfies $|p_{pred} - p_{real}| \leq \varepsilon$. Since files are often organized into pages, we set ε as half the number of models that can fit into a single disk page to generate the models in a disk-friendly manner. As will be shown, this reduces the IO cost by ensuring that at most two pages need to be accessed per model during read operations.

Algorithm 2: Learn Models from a Stream

```
Function BuildModel(S, ε)
    Input: Input stream S, error bound ε
    Output: A stream of models {M}
    k_min ← ∅, p_max ← ∅, g_last ← ∅;
    Init an empty convex hull H;
    foreach ⟨K, p_real⟩ ← S do
        if k_min = ∅ then k_min ← K;
        Add ⟨BigNum(K), p_real⟩ to H;
        Compute the minimum parallelogram G that covers H;
        if G.height ≤ 2ε then
            p_max ← p_real, g_last ← G;
        else
            Compute slope sl and intercept ic from g_last;
            M ← ⟨sl, ic, k_min, p_max⟩;
            yield M;
            k_min ← K;
            Init a new convex hull H with ⟨BigNum(K), p_real⟩;
```

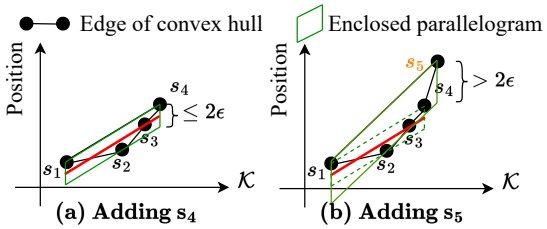

Figure 5: An Example of Model Learning

To compute models from a stream of compound keys and their corresponding positions, we treat each compound key and its position as a point's coordinates. Upon the arrival of a new compound key, we convert it into a big integer using the binary representation of the address and the block height as mentioned in Section 3.2. Next, we find the smallest convex shape containing all the existing input points, which is known as a convex hull. Note that this convex hull can be computed incrementally in a streaming fashion [40]. Then, we find the minimal parallelogram that covers the convex hull, with one side aligned to the vertical axis (i.e., the position axis). If the parallelogram's height stays under 2ε, all existing inputs can fit into a single model. In this case, we try to include the next compound key in the stream for model construction. However, if the current parallelogram fails to meet the height criteria, the slope and intercept of the central line in the parallelogram will be used to build a model that covers all existing compound keys except the current one. After this, a new model will be built, starting from the current compound key. We summarize the algorithm in Algorithm 2.

Example. *Figure 5 shows an example of model learning from a stream. Assume states s_1 to s_3 form a convex hull, with its minimal parallelogram satisfying the height criterion (i.e., below 2ε). After state s_4 is added, the parallelogram's height remains within 2ε (see Figure 5(a)), indicating that states s_1 to s_4 can be fit into one model. However, after the next state s_5 is added, the parallelogram's height exceeds 2ε (see*

Algorithm 3: Index File Construction

1 **Function** `ConstructIndexFile`$(\mathcal{S}, \varepsilon)$
  Input: Input stream $\mathcal{S}$ of compound key-position pairs
  Output: Index file $\mathcal{F}_I$
2 Create an empty index file $\mathcal{F}_I$;
3 Invoke `BuildModel`$(\mathcal{S}, \varepsilon)$ and write to $\mathcal{F}_I$;
4 $n \leftarrow$ # of pages in $\mathcal{F}_I$;
5 **while** $n > 1$ **do**
6 $\mathcal{S} \leftarrow \{\langle \mathcal{M}.k_{min}, pos\rangle \mid$ **foreach** $\langle \mathcal{M}, pos\rangle \in \mathcal{F}_I[-n:]\}$;
7 Invoke `BuildModel`$(\mathcal{S}, \varepsilon)$ and append to $\mathcal{F}_I$;
8 $n \leftarrow$ # of pages in $\mathcal{F}_I - n$;
9 **return** $\mathcal{F}_I$;

Figure 5(b)). Thus, the slope and intercept of the previous parallelogram's central line (highlighted in red) are used to build a model for s_1 to s_4, with s_5 reserved for the next model.

Algorithm 3 shows the overall procedure of index file generation. During flush or sort-merge operations in Algorithm 1, ordered compound keys and state values are generated and written streamingly into the value file. Meanwhile, another stream consisting of compound keys and their positions is created and used to generate models with Algorithm 2 (Line 3). Once the models are yielded by Algorithm 2, they are immediately written to the index file, constituting the bottom layer of the run's learned index. Then, we recursively build the upper layers of the index until the top layer can fit into a single disk page (Lines 4 to 8). Specifically, for each layer, we scan lower-layer models (denoted as $\mathcal{F}_I[-n:]$) to create a compound key stream using k_{min} in each model and their index file positions (Line 6). Similar to the bottom layer, we use Algorithm 2 on the stream to create models and instantly write them to the index file (Line 7). This results in the sequential storage of models across layers in a bottom-up manner. The index file remains valid from its construction until the next level merge operation thanks to the LSM-tree-based maintenance approach, which avoids costly model retraining.

4.2 Merkle File Construction

A Merkle file stores an m-ary complete MHT that authenticates the compound key-value pairs in the corresponding value file. The related index file's learned models are excluded from authentication, as they solely enhance query efficiency and do not affect blockchain data integrity. For the m-ary complete MHT, the bottom layer consists of hash values of every compound key-value pair in the value file. The hash values in an upper layer are recursively computed from every m hash values in the lower layer, except that the last one might be computed from less than m hash values in the lower layer.

Definition 2 (Hash Value)**.** *A hash value in the bottom layer of the MHT is computed as $h_i = h(\mathcal{K}_i \| value_i)$, where $\mathcal{K}_i, value_i$ are the corresponding compound key and value, $\|$ is the concatenation operator, and $h(\cdot)$ is a cryptographic hash function such as SHA-256. A hash value in an upper layer of the MHT is computed as $h_i = h(h_i^1 \| h_i^2 \| \cdots \| h_i^{m^*})$, where $m^* \leq m$ and h_i^j is the corresponding j-th hash in the lower layer.*

Algorithm 4: Merkle File Construction

1 **Function** `ConstructMerkleFile`$(\mathcal{S}, n, m)$
  Input: Input stream $\mathcal{S}$ of compound key-value pairs, stream size n, fanout m
  Output: Merkle file $\mathcal{F}_H$
2 $N_{nodes} \leftarrow [n, \lceil \frac{n}{m} \rceil, \lceil \frac{n}{m^2} \rceil, \cdots, 1]$, $d \leftarrow |N_{nodes}|$;
3 layer_offset$[0] \leftarrow 0$;
4 layer_offset$[i] \leftarrow \sum_0^{i-1} N_{nodes}[i-1], \forall i \in [1, d-1]$;
5 Create a merkle file $\mathcal{F}_H$ with size $\sum_{i=0}^{d-1} N_{nodes}[i]$;
6 Create a cache $\mathcal{C}$ with d number of buffers;
7 **foreach** $\langle \mathcal{K}, value\rangle \leftarrow S$ **do**
8 $h' \leftarrow h(\mathcal{K} \| value)$, append h' to $\mathcal{C}[0]$;
9 **foreach** i ***in*** 0 ***to*** $d-2$ **do**
10 **if** $|\mathcal{C}[i]| = m$ **then**
11 $h' \leftarrow h(\mathcal{C}[i])$, append h' to $\mathcal{C}[i+1]$;
12 Flush $\mathcal{C}[i]$ to $\mathcal{F}_H$ at offset layer_offset$[i]$;
13 layer_offset$[i] \leftarrow$ layer_offset$[i] + m$;
14 **else break**;
15 **foreach** i ***in*** 0 ***to*** $d-1$ **do**
16 **if** $\mathcal{C}[i]$ *is not empty* **then**
17 $h' \leftarrow h(\mathcal{C}[i])$, append h' to $\mathcal{C}[i+1]$;
18 Flush $\mathcal{C}[i]$ to $\mathcal{F}_H$ at offset layer_offset$[i]$;
19 **return** $\mathcal{F}_H$;

Similar to Algorithm 3, we streamingly generate the Merkle file. However, instead of layer-wise construction, we concurrently build all MHT layers to reduce IO costs, as shown in Algorithm 4. Note that the size of the input stream of compound key-value pairs n is known in advance since the size of a value file is determined by the level of its corresponding run. Thus, the MHT has $\lceil \log_m n \rceil + 1$ layers, containing $n, \lceil \frac{n}{m} \rceil, \lceil \frac{n}{m^2} \rceil, \cdots, 1$ hash values (Line 2). Layer offsets can also be computed (Lines 3 to 4). For concurrent construction, $\lceil \log_m n \rceil + 1$ buffers are maintained, one per layer. Upon the arrival of a new compound key-value pair, its hash value is computed and added to the bottom layer's buffer (Line 8). When a buffer fills with m hash values, an upper layer's hash value is created and added to its buffer (Line 11). Next, the buffered hash values in the current layer are flushed to the Merkle file, followed by incrementing the offset (Lines 12 to 13). This process recurs in upper layers until a layer with less than m buffered hash values is encountered. Once the input stream is fully processed, any remaining non-empty buffers will hold fewer than m hash values. If so, we'll initiate this process by taking a buffer from the lowest layer and iteratively generating hash values. Each hash value is added to the upper layer before flushing the buffer to the Merkle file (Lines 15 to 18).

Example. *Figure 6 shows an example of a 2-ary MHT with states s_1 to s_4. According to the MHT's structure, $N_{nodes} = [4, 2, 1]$ and layer_offset $= [0, 4, 6]$. Assume that s_1, s_2 are already added. In this case, $\mathcal{F}_H$ has h_1, h_2 and cache $\mathcal{C}[1]$ con-*

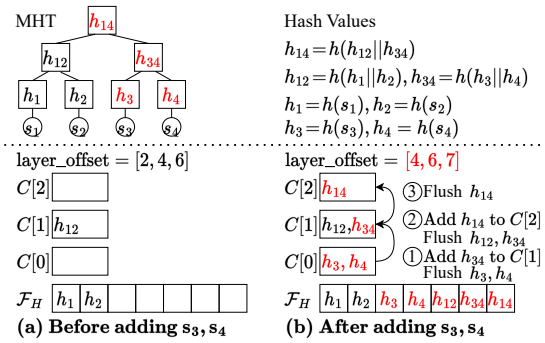

Figure 6: An Example of Merkle File Construction

tains h_{12}, where h_1, h_2 are the hash values of s_1, s_2 and h_{12} is derived from h_1, h_2 (Figure 6(a)). Meanwhile, layer_offset$[0]$ has been updated to 2. After s_3, s_4 are added, their hashes h_3, h_4 will be inserted to cache $C[0]$, resulting in $C[0]$ having 2 hash values. Thus, h_{34} derived from h_3, h_4 will be added into cache $C[1]$ and h_3, h_4 are then flushed to $\mathcal{F}_H$ at offset 2 (step ①). Since $C[1]$ also has 2 hash values so the derived h_{14} is added to cache $C[2]$ and h_{12}, h_{34} are flushed to $\mathcal{F}_H$ at offset layer_offset$[1] = 4$ (step ②). Finally, h_{14} in $C[2]$ is flushed to $\mathcal{F}_H$ at offset layer_offset$[2] = 6$ (step ③).

4.3 Discussions

As discussed earlier, COLE adopts the LSM-tree-based maintenance approach to optimize data writes and disk operations under the column-based design. However, it also comes with some tradeoffs. The presence of multiple levels can impact read performance, as retrieving a state requires traversing multiple levels until a satisfactory result is found. Additionally, the merge operation complicates the process of state rewind, as data cannot be deleted in-place. Therefore, COLE does not support blockchain forking and is designed to work with blockchains that do not fork [5, 22, 53].

We next discuss the ACID properties in COLE. COLE achieves atomicity by maintaining root_hash_list in an atomic manner. During the level merge process, root_hash_list is updated atomically only after constructing all three files in the new level, followed by removing the old level files. This ensures data consistency as the old level files remain intact and are referenced by root_hash_list even during a node crash. Concurrency control is not required due to the write-serializability guarantee of the consensus protocol. Data integrity is ensured using Merkle-based structures for each level. For durability, COLE uses transaction logs as the Write Ahead Log since they are agreed upon by the consensus protocol. In case of a crash, COLE recovers by replaying transactions since the last checkpoint. A checkpoint is created when the in-memory MB-tree is flushed to the first disk level and cleared. At this time point, all the data in the system is safely stored on the disk. After a crash, COLE reverts to the last checkpoint, discards all the files in the unfinished merge levels, and starts fresh with an empty in-memory MB-tree. It then replays all unprocessed transactions and restarts the aborted level merges.

5 Write with Asynchronous Merge

Algorithm 1 may trigger recursive merge operations during some writes (e.g., steps ① and ② in Figure 4). As a result, it can introduce long-tail latency and cause all future operations to stall. This issue is known as *write stall*, which leads to periodic drops in application throughput to near-zero levels and dramatic fluctuations in system performance. A common solution is to make the merge operations asynchronous by moving them to separate threads. However, the existing asynchronous merge solution is not suitable for blockchain applications. Since different nodes in the blockchain network could have drastically different computation capabilities, the storage structure will become out-of-sync among nodes when applying asynchronous merges. This will result in different H_{state}'s and break the requirement of the blockchain protocol.

To address these challenges, we design a novel asynchronous merge algorithm for COLE, which ensures the synchronization of the storage across blockchain nodes. The algorithm introduces two checkpoints, *start* and *commit*, within the asynchronous merge process for each on-disk level. By synchronizing the checkpoints, we ensure consistent blockchain storage and thus H_{state} agreed by the network. To further minimize the possibility of long-tail latency due to delays at the commit checkpoint, we propose to make the interval between the start checkpoint and the commit checkpoint proportional to the size of the run. This ensures that the majority of the nodes in the network can complete the merge operation before reaching the commit checkpoint.

To realize our idea, we propose to have each level of COLE contain two groups of runs as shown in Figure 7. Each group's design is identical to the one discussed in Section 4. Specifically, the in-memory level now contains two groups of MB-tree, each with a capacity of B states. Similarly, each on-disk level contains two groups of up to T sorted runs. Level i can hold a maximum of $2 \cdot B \cdot T^i$ states. The two groups in each level have two mutually exclusive roles, namely *writing* and *merging*. The writing group accepts newly created runs from the upper level. On the other hand, the merging group generates a new run from its own data and adds to the writing group of the next level in an asynchronous fashion.

Algorithm 5 shows the write operation in COLE with asynchronous merge. First, new state values are inserted into the current writing group of in-memory level L_0 (Lines 2 to 4). The levels in COLE are then traversed from smaller to larger. When a level is full, we commit the previous merge operation in the current level and start a new merge operation in a new thread. To accommodate slow nodes in the network, we check if the previous merging thread of the current level exists and is still in progress, and wait for it to finish if necessary (Line 9).

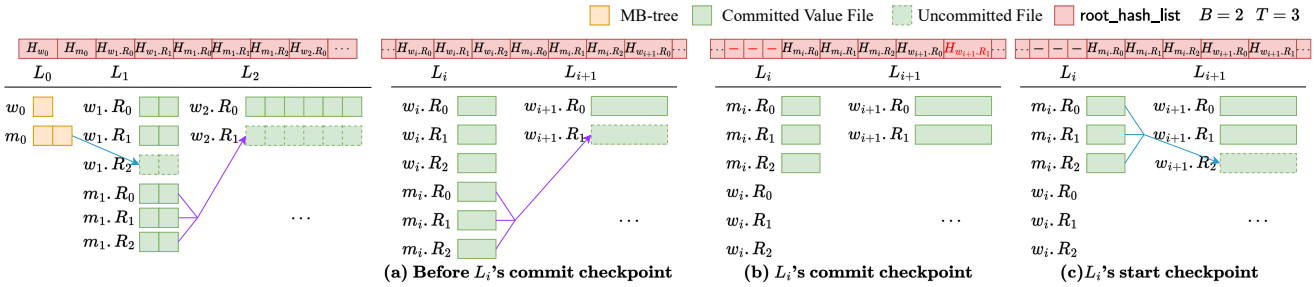

Figure 7: Asynchronous Merge

Figure 8: An Example of Asynchronous Merge

Algorithm 5: Write Algorithm with Asynchronous Merge

```
1  Function Put(addr, value)
     Input: State address addr, value value
2    blk ← current block height; K ← ⟨addr, blk⟩;
3    w_0 ← Get L_0's writing group;
4    Insert ⟨K, value⟩ into the MB-tree of w_0;
5    i ← 0;
6    while w_i becomes full do
7      m_i ← Get L_i's merging group;
8      if m_i.merge_thread exists then
9        Wait for m_i.merge_thread to finish;
10       Add the root hash of the generated run from
           m_i.merge_thread to root_hash_list;
11       Remove the root hashes of the runs in m_i from
           root_hash_list;
12       Remove all the runs in m_i;
13     Switch m_i and w_i;
14     m_i.merge_thread ← start thread do
15       if i = 0 then
16         Flush the leaf nodes in m_i to L_{i+1}'s writing group a
             sorted run;
17         Generate files F_V, F_I, F_H for the new run;
18       else
19         Sort-merge all the runs in m_i to L_{i+1}'s writing
             group a new run;
20         Generate files F_V, F_I, F_H for the new run;
21     i ← i + 1;
22   Update H_state when finalizing the current block;
```

The previous merge operation is committed by adding the root hash of the newly generated run to root_hash_list (Line 10), while obsolete run hashes are removed from root_hash_list (Line 11) and the obsolete runs in the merging group are also removed (Line 12). The above procedure ensures the commit checkpoint occurs simultaneously across nodes in the network, which is essential to synchronize the blockchain states and the corresponding root digest. Following this, the roles between the two groups in the current level are switched (Line 13). This means that future write operations will be directed to the vacated space of the new writing group, whereas the merge operation will be performed on the new merging group, which is now full. The latter starts a new merge thread, whose procedure is similar to that of Algorithm 1 (Lines 14 to 20). Lastly, when finalizing the current block, H_{state} is updated using stored root hashes in root_hash_list (Line 22).

Example. *Figure 8 shows an example of the asynchronous merge from level L_i to L_{i+1}, where $T = 3$. The uncommitted files are denoted by dashed boxes. Figure 8(a) shows COLE's structure before L_i's commit checkpoint, when L_i's writing group w_i becomes full. In case m_i's merging thread (denoted by the purple arrow) is not yet finished, we wait for it to finish. Then, during L_i's commit checkpoint, $w_{i+1}.R_1$'s root hash is added to root_hash_list and all runs in m_i (i.e., $m_i.R_0, m_i.R_1, m_i.R_2$) are removed (Figure 8(b)). Next, m_i and w_i's roles are switched. Finally, a new thread will be started (denoted by the blue arrow) to merge all runs in m_i to L_{i+1}'s writing group as the third run $w_{i+1}.R_2$ (Figure 8(c)).*

Soundness Analysis. Next, we show our proposed asynchronous merge operation is sound. Specifically, the following two requirements are satisfied.

- The blockchain states' root digest H_{state} is always synchronized among nodes in the blockchain network regardless of how long the underlying merge operation takes.
- The interval between the start checkpoint and the commit checkpoint for each level is proportional to the size of the runs to be merged.

The first requirement ensures blockchain states are solely determined by the current committed states and are independent of individual node performance variations. The second requirement minimizes the likelihood of nodes waiting for merge operations of longer runs. We now prove that our algorithm complies with the requirements.

Proof Sketch. It is trivial to show that the first requirement is satisfied as the update of root_hash_list (hence H_{state}) occurs outside the asynchronous merge thread, making the update of H_{state} fully synchronous and deterministic. For the second requirement, the interval between the start checkpoint and the commit checkpoint in any level equals the time taken to fill up the writing group in the same level. Since the latter contains those runs to be merged in this level, the interval is proportional to the size of the runs. □

6 Read Operations of COLE

In this section, we discuss the read operations of COLE, including the get query and the provenance query with its veri-

Algorithm 6: Get Query

1 **Function** `Get` $(addr)$
  Input: State address $addr$
  Output: State latest value $value$
2 $\mathcal{K}_q \leftarrow \langle addr, max\_int \rangle$;
3 **foreach** g ***in*** $\{L_0$*'s writing group,* L_0*'s merging group*$\}$ **do**
4 $\langle K', state' \rangle \leftarrow$ `SearchMBTree`$(g, \mathcal{K}_q)$;
5 **if** $\mathcal{K}'.addr = addr$ **then return** $state'$;
6 **foreach** *level* i ***in*** $\{1, 2, \dots\}$ **do**
7 $RS \leftarrow \{R_{i,j} \mid R_{i,j} \in L_i$'s writing group $\wedge$ committed$\}$;
8 $RS \leftarrow RS + \{R_{i,j} \mid R_{i,j} \in L_i$'s merging group$\}$;
9 **foreach** $R_{i,j}$ ***in*** RS **do**
10 $\langle \langle K', state' \rangle, pos' \rangle \leftarrow$ `SearchRun`$(R_{i,j}, \mathcal{K}_q)$;
11 **if** $\mathcal{K}'.addr = addr$ **then return** $state'$;
12 **return** *nil*;

fication function. We assume that COLE is implemented with the asynchronous merge.

6.1 Get Query

Algorithm 6 shows the get query process. As mentioned in Section 3.2, getting a state's latest value requires a special compound key $\mathcal{K}_q = \langle addr_q, max\_int \rangle$. Owing to the temporal order of COLE's levels, we perform the search from smaller levels to larger levels, until a satisfied state value is found. This involves searching both the writing and merging groups' MB-trees in the in-memory level L_0 as both of them are committed (Lines 3 to 5). Then, in each on-disk level, a search is performed in the committed writing group's runs, followed by the merging group's runs (Lines 6 to 11). Note that the runs in the same group will be searched in the order of their freshness. For the example in Figure 7, we search the MB-trees in w_0 and m_0, followed by the runs in the order of $w_1.R_1$, $w_1.R_0$, $m_1.R_2$, $m_1.R_1$, $m_1.R_0$, $w_2.R_0$, $\cdots$, while the uncommitted $w_1.R_2, w_2.R_1$ are skipped. The search halts once the satisfied state is found.

To search an on-disk run, we use Algorithm 7. First, if the queried address $addr_q$ is not in the run's bloom filter $\mathcal{B}$, the run is skipped (Line 2). Otherwise, models in the index file $\mathcal{F}_I$ are used to find $\mathcal{K}_q$. The search starts from the top layer of models, stored on the last page of $\mathcal{F}_I$. The model covering $\mathcal{K}_q$ is found by binary searching k_{min} of each model in this page (Line 4). Then, a recursive query on models in subsequent layers is conducted from top to bottom (Lines 5 to 7). Upon reaching the bottom layer, the corresponding model is used to locate the state value in the value file $\mathcal{F}_V$ (Line 8).

Function `QueryModel`$(\cdot)$ in Algorithm 7 shows the procedure of using a learned model $\mathcal{M}$ to locate the queried compound key $\mathcal{K}_q$. If the model covers $\mathcal{K}_q$, it predicts the position pos_{pred} of the queried data (Line 12). With the error bound of the model 2ε equaling the page size, the predicted page id is computed as $pos_{pred}/2\varepsilon$ (Line 13). The corresponding page $\mathcal{P}$ is fetched and the first and last models are checked whether they cover $\mathcal{K}_q$. If not, the adjacent page is fetched as

Algorithm 7: Search a Run

1 **Function** `SearchRun` $(\mathcal{F}_I, \mathcal{F}_V, \mathcal{B}, \mathcal{K}_q)$
  Input: Index file $\mathcal{F}_I$, value file $\mathcal{F}_V$, bloom filter $\mathcal{B}$,
  compound key $\mathcal{K}_q = \langle addr_q, blk_q \rangle$
  Output: Queried state s and its position pos
2 **if** $addr_q \notin \mathcal{B}$ **then return**;
3 $\mathcal{K}_q \leftarrow BigNum(\mathcal{K}_q)$;
4 $\mathcal{P} \leftarrow \mathcal{F}_I$'s last page; $\mathcal{M} \leftarrow$ `BinarySearch`$(\mathcal{P}, \mathcal{K}_q)$;
5 $\langle \mathcal{M}, pos \rangle \leftarrow$ `QueryModel`$(\mathcal{M}, \mathcal{F}_I, \mathcal{K}_q)$;
6 **while** pos *is* ***not*** *pointing to the bottom models* **do**
7 $\langle \mathcal{M}, pos \rangle \leftarrow$ `QueryModel`$(\mathcal{M}, \mathcal{F}_I, \mathcal{K}_q)$;
8 **return** `QueryModel`$(\mathcal{M}, \mathcal{F}_V, \mathcal{K}_q)$;
9 **Function** `QueryModel` $(\mathcal{M}, \mathcal{F}, \mathcal{K}_q)$
  Input: Model $\mathcal{M}$, query file $\mathcal{F}$, compound key $\mathcal{K}_q$
  Output: Queried data and its position in $\mathcal{F}$
10 $\langle sl, ic, k_{min}, p_{max} \rangle \leftarrow \mathcal{M}$;
11 **if** $\mathcal{K}_q < k_{min}$ **then return**;
12 $pos_{pred} \leftarrow \min(\mathcal{K}_q \cdot sl + ic, p_{max})$;
13 $page_{pred} \leftarrow pos_{pred}/2\varepsilon$;
14 $\mathcal{P} \leftarrow \mathcal{F}$'s page at $page_{pred}$;
15 **if** $\mathcal{K}_q < \mathcal{P}[0].k$ **then**
16 $\mathcal{P} \leftarrow \mathcal{F}$'s page at $page_{pred} - 1$;
17 **else if** $\mathcal{K}_q > \mathcal{P}[-1].k$ **then**
18 $\mathcal{P} \leftarrow \mathcal{F}$'s page at $page_{pred} + 1$;
19 **return** `BinarySearch`$(\mathcal{P}, \mathcal{K}_q)$;

$\mathcal{P}$ (Lines 15 to 18). This process involves at most two pages for prediction, hence minimizing IO. Finally, a binary search in $\mathcal{P}$ locates the queried data (Line 19).

6.2 Provenance Query

A provenance query resembles a get query but with notable distinctions. Unlike a get query, a provenance query involves a range search based on the queried block height range. This entails computing two boundary compound keys, $\mathcal{K}_l = \langle addr, blk_l - 1 \rangle$ and $\mathcal{K}_u = \langle addr, blk_u + 1 \rangle$, with offsets adjusted by one to prevent the omission of valid results. Moreover, a provenance query provides Merkle proofs to authenticate the results.

Specifically, during the search of MB-trees in L_0, in addition to retrieving satisfactory results, Merkle paths are included in the proof using a similar approach mentioned in Section 2. For the runs of the on-disk levels, we search in the same order as those described in Algorithm 6. $\mathcal{K}_l$ is used as the search key when applying the learned models to find the first query result in each run. Then, the value file is scanned sequentially until a state beyond $\mathcal{K}_u$ is reached.[1] Afterwards, a Merkle proof is computed upon the first and last results' positions pos_l, pos_u of each run. Since the states in the value file and their hash values in the Merkle file share the same position, the Merkle paths of the hash values at pos_l and pos_u are used as the Merkle proof. To compute the Merkle path,

[1] For simplicity, we assume that $addr$ is in the bloom filter $\mathcal{B}$. If not, $\mathcal{B}$ is also added as the proof to prove that $addr$ is not in the run.

Cost	MPT	COLE	COLE w/ async-merge
Storage size	$O(n \cdot d_{MPT})$	$O(n)$	
Write IO cost	$O(d_{MPT})$	$O(d_{COLE})$	
Write tail latency	$O(1)$	$O(n)$	$O(1)$
Write memory footprint	$O(1)$	$O(T + m \cdot d_{COLE})$	$O(T \cdot d_{COLE} + m \cdot d_{COLE}^2)$
Get query IO cost	$O(d_{MPT})$	$O(T \cdot d_{COLE} \cdot C_{model})$	
Prov-query IO cost	$O(d_{MPT})$	$O(T \cdot d_{COLE} \cdot C_{model} + m \cdot d_{COLE}^2)$	
Prov-query proof size	$O(d_{MPT})$	$O(m \cdot d_{COLE}^2)$	

Table 1: Complexity Comparison

we traverse the MHT in the Merkle file from bottom to top. Note that given a hash value's position pos at layer i, we can directly compute its parent hash value's position in the Merkle file as $\lfloor (pos - \sum_0^{i-1} \lceil \frac{n}{m^i} \rceil)/m \rfloor + \sum_0^i \lceil \frac{n}{m^i} \rceil$. Due to the space limitation, the detailed procedure of the provenance query is given in Appendix B .

On the user's side, the verification algorithm works as follows: (1) use each MB-tree's results and their corresponding Merkle proof to reconstruct the MB-tree's root hash; (2) use each searched run's results and their corresponding Merkle proof to reconstruct the run's root hash; (3) use the reconstructed root hashes to reconstruct the states' root digest and compare it with the published one, H_{state}, in the block header; (4) check the boundary results of each searched run against the compound key range $[\mathcal{K}_l, \mathcal{K}_u]$ to ensure no missing results. If all these checks pass, the results are verified.

7 Complexity Analysis

In this section, we analyze the complexity in terms of storage, memory footprint, and IO cost. To ease the analysis, we assume n as the total historical values, T as the level size ratio, B as the in-memory level's capacity, and m as COLE's MHT fanout. Table 1 shows the comparison of MPT, COLE, and COLE with the asynchronous merge.

We first analyze the storage size. Since MPT duplicates the nodes of the update path for each insertion, its storage has a size of $O(n \cdot d_{MPT})$, where d_{MPT} is the height of the MPT. COLE completely removes the node duplication, thus achieving an $O(n)$ storage size.

Next, we analyze the write IO cost. MPT takes $O(d_{MPT})$ to write the nodes in the update path, while COLE takes $O(d_{COLE})$ for the worst case when all levels are merged, where d_{COLE} is the number of levels in COLE. Similar to the traditional LSM-tree's write cost [13], the level merge in COLE takes an amortized $O(1)$ IO cost to write the value file, the index file, and the Merkle file. The number of levels d_{COLE} is $\lceil \log_T(\frac{n}{B} \cdot \frac{T-1}{T}) \rceil$, which is logarithmic to n. Note that normally $d_{COLE} < d_{MPT}$ since d_{MPT} depends on the data's key size, which can be large (e.g., when having a 256-bit key, maximum d_{MPT} is 64 under hexadecimal base while COLE has only a few levels following the LSM-tree).

Regarding the write tail latency, MPT has a constant cost since there is no write stall during data writes. On the other hand, COLE may experience the write stall in the worst case, which requires waiting for the merge of all levels and results in the reading and writing of $O(n)$ states. The asynchronous merge algorithm removes the write stall by merging the levels in background threads and reduces the tail latency to $O(1)$.

As for the write memory footprint, MPT has a constant cost since the update nodes are computed on the fly and can be removed from the memory after being flushed to the disk. For COLE, we consider the case of merging the largest level as this is the worst case. The sort-merge takes $O(T)$ memory and the model construction takes constant memory [40]. Constructing the Merkle file takes $O(m \cdot d_{COLE})$ since there are logarithmic layers of cache buffers and each buffer contains m hash values. To sum up, COLE takes $O(T + m \cdot d_{COLE})$ memory during a write operation. For COLE with the asynchronous merge, the worst case is that each level has a merging thread, thus requiring d_{COLE} times of memory compared with the synchronous merge, i.e., $O(T \cdot d_{COLE} + m \cdot d_{COLE}^2)$.

We finally analyze the read operations' costs, including the get query IO cost, the provenance query IO cost, and the proof size of the provenance query. MPT's costs are all linear to the MPT's height, $O(d_{MPT})$. For COLE, T runs in each level should be queried, where we assume that each run takes C_{model} to locate the state. Therefore, the cost of the get query is $O(T \cdot d_{COLE} \cdot C_{model})$. To generate the Merkle proof during the provenance query, an additional $O(m \cdot d_{COLE}^2)$ is required since there are multiple layers of MHT in all levels and $O(m)$ hash values are retrieved for each MHT's layer. The proof size is $O(m \cdot d_{COLE}^2)$ for a similar reason.

8 Evaluation

In this section, we first describe the experiment setup, including comparing baselines, implementation, parameter settings, workloads, and evaluation metrics. Then, we present the experiment results.

8.1 Experiment Setup

8.1.1 Baselines

We compare COLE with the following baselines:

- MPT: It is used by Ethereum to index the blockchain storage. The structure is made persistent as mentioned in Section 1.
- LIPP: It applies LIPP [54], the state-of-the-art learned index supporting *in-place* data writes, to the blockchain storage without our column-based design. LIPP retains the node persistence strategy to support provenance queries.
- Column-based Merkle Index (CMI): It uses the column-based design with traditional Merkle indexes rather than the learned index. It adopts a two-level structure. The upper index is a non-persistent MPT whose key is the state address and the value is the root hash of the lower index. The lower index follows the column-based design, using an MB-tree to store the state's historical values in a

Parameters	Value
# of generated blocks	$10^2, 10^3, 10^4, \mathbf{10^5}$
Size ratio T	$2, \mathbf{4}, 6, 8, 10, 12$
COLE's MHT fanout m	$2, \mathbf{4}, 8, 16, 32, 64$

Table 2: System Parameters

contiguous fashion [29].

8.1.2 Implementation and Parameter Setting

We implement COLE and the baselines in Rust programming language. The source code is available at https://github.com/hkbudb/cole. We use the Rust Ethereum Virtual Machine (EVM) to execute transactions, simulating blockchain data updates and reads [2]. Transactions are packed into blocks, each containing 100 transactions. Ten smart contracts are initially deployed and repeatedly invoked with transactions. Big number operations mentioned in Section 3.2 are implemented using the *rug* library [3]. Baselines utilize RocksDB [18] as the underlying storage, while COLE uses simple files for data storage as enabled by our design.

We set $\varepsilon = 23$ based on the page size (4KB) and the compound key-pair size (88 bytes). By default, the size ratio T and the MHT fanout m of COLE are set to 4. Following the default configuration of RocksDB, its memory budget is set to 64MB. The in-memory capacity B is set to the number of states that can fit within the same memory budget. Table 2 shows all the parameters where the default settings are highlighted in bold font. All experiments are run on a machine equipped with an Intel i7-10710U CPU, 16GB RAM, and Samsung SSD 256GB.

8.1.3 Workloads and Evaluation Metrics

The experiment evaluation includes two parts: the overall performance of transaction executions and the performance of provenance queries. For the first part, SmallBank and KVStore from Blockbench [17] are used as macro benchmarks to generate the transaction workload. SmallBank simulates the account transfers while KVStore uses YCSB [9] for read/write tests. YCSB involves a loading phase where base data is generated and stored, followed by a running phase for read/update operations. A transaction that reads/updates data is denoted as a read/write transaction. We set 10^5 transactions as the base data and vary read/update ratios to simulate different scenarios: (i) Read-Write with equal read/write transactions; (ii) Read-Only with only read transactions; and (iii) Write-Only with all write transactions. The overall performance is evaluated in terms of the average transaction throughput, the tail latency, and the storage size.

To evaluate provenance queries, we use KVStore to simulate the workload including frequent data updates. We initially write 100 states as the base data and then continuously generate write transactions to update the base data's states. For each query, we randomly select a key from the base data and vary the block height range (e.g., $2, 4, \cdots, 128$), which follows [44]'s setting. The evaluation metrics include (i) CPU time of the query executed on the blockchain node and verified by the query user and (ii) proof size.

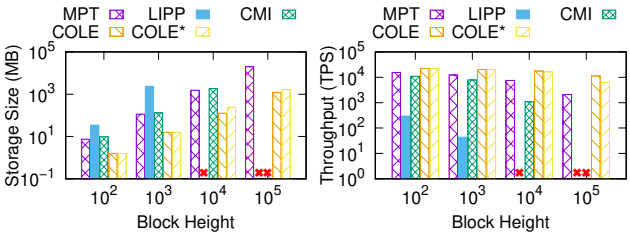

Figure 9: Performance vs. Block Height (SmallBank)

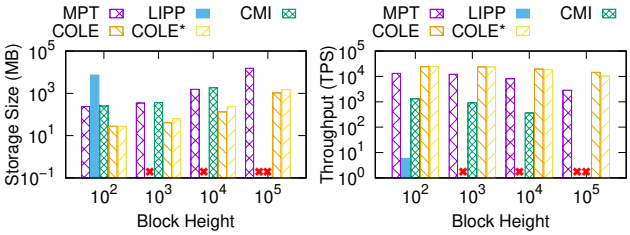

Figure 10: Performance vs. Block Height (KVStore)

8.2 Experimental Results

8.2.1 Overall Performance

Figures 9 and 10 show the storage size and throughput of COLE and all baselines under the SmallBank and KVStore workloads, respectively. We denote COLE with the asynchronous merge as COLE*.

We make several interesting observations. First, COLE significantly reduces the storage size compared to MPT as the blockchain grows. For example, at a block height of 10^5, the storage size decreases by 94% and 93% for SmallBank and KVStore, respectively. This is due to COLE's elimination of the need to persist internal data structures via the column-based design, and its use of storage-efficient learned models for indexing. Moreover, COLE outperforms MPT in throughput, achieving a 1.4×-5.4× improvement, thanks to its learned index. COLE* performs slightly worse than COLE due to the overhead of the asynchronous merge.

Second, using the learned index without the column-based design (LIPP) even increases the blockchain storage. At a block height of 10^2, the storage size of LIPP already exceeds MPT's by 5× (for SmallBank) and 31× (for KVStore). This happens because the learned index often generates larger index nodes that must be persisted with each new block, leading to increased storage and significant IO operations. Consequently, LIPP's throughput is significantly worse than MPT. We are not able to report the results of LIPP for the block height above 10^3 for SmallBank and 10^2 for KVStore as the experiment could not be finished within 24 hours.

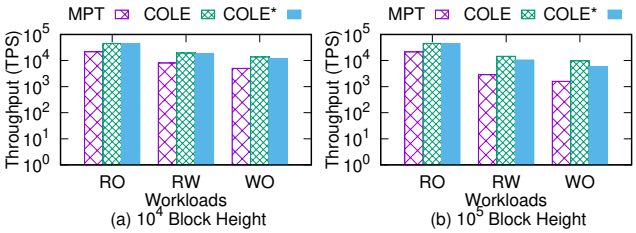

Figure 11: Throughput vs. Workloads (KVStore)

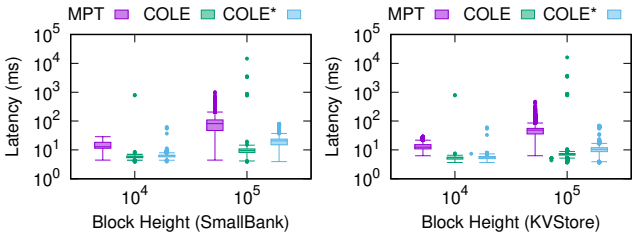

Figure 12: Latency Box Plot

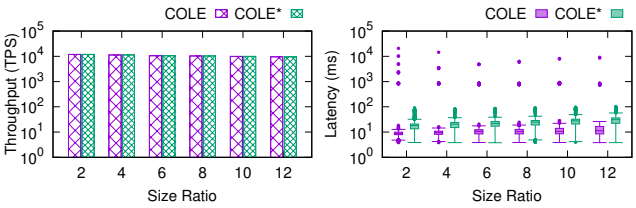

Figure 13: Impact of Size Ratio

Third, extending MPT with the column-based design (CMI) does not significantly change the storage size. The additional storage of the lower-level MB-tree and the use of the RocksDB backend largely negate the benefit of removing node persistence. Additionally, refreshing Merkle hashes of all nodes in the index update path, which entails both read and write IOs, further impacts performance. Consequently, the throughput of CMI is 7× and 22× worse than MPT for SmallBank and KVStore, respectively, at a block height of 10^4. The experiments of CMI cannot scale beyond a block height of 10^4.

Overall, with a unique combination of the learned index, column-based design, and write-optimized strategies, COLE and COLE* not only achieve the smallest storage requirement but also gain the highest system throughput.

8.2.2 Impact of Workloads

We use KVStore to evaluate the impact of different workloads, namely Read-Only (RO), Read-Write (RW), and Write-Only (OW), in terms of the system throughput. As shown in Figure 11, the throughputs of all systems decrease with more write operations in the workload. The performance of MPT degrades by up to 93% while that of COLE and COLE* degrades by up to 87%. This shows that the LSM-tree-based maintenance approach helps optimize the write operation. We omit LIPP and CMI in Figure 11 since they cannot scale beyond a block height of 10^3 and 10^4, respectively.

8.2.3 Tail Latency

To assess the effect of the asynchronous merge, Figure 12 shows the box plot of the latency of SmallBank and KVStore workloads at block heights of 10^4 and 10^5. The tail latency is depicted as the maximum outlier. As the blockchain grows, COLE* decreases the tail latency by 1-2 orders of magnitude for both workloads. This shows that the asynchronous merge strategy will become more effective when the system scales up for real-world applications. Owing to the asynchronous merge overhead, COLE* incurs slightly higher median latency than COLE, but it still outperforms MPT.

8.2.4 Impact of Size Ratio

Figure 13 shows the system throughput and latency box plot under 10^5 block height using the SmallBank benchmark with varying size ratio T. As the size ratio increases, the throughput remains stable, while the tail latency shows a U shape. We observe that $T = 6$ and $T = 4$ are the best settings for COLE and COLE*, respectively, with the lowest tail latency. Meanwhile, with an increasing size ratio, the median latency of both COLE and COLE* increases.

8.2.5 Provenance Query Performance

We now evaluate the performance of provenance queries by querying historical state values of a random address within the latest q blocks. With the current block height fixed at 10^5, we vary q from 2 to 128. LIPP and CMI are omitted here since they cannot scale at 10^5 block height. Figure 14 shows that MPT's CPU time and proof size grow linearly with q while those of COLE and COLE* grow only sublinearly. This is because MPT requires to query each block inside the queried range. In contrast, COLE and COLE*'s column-based design often locates query results within contiguous storage of each run, hence reducing the number of index traversals during the query and shrinking the proof size by sharing ancestor nodes in the Merkle path. COLE and COLE*'s proof sizes surpass that of MPT when the query range is small due to limited sharing capabilities within a small query range.

9 Related Work

In this section, we briefly review the related works on learned indexes and blockchain storage management.

9.1 Learned Index

Learned index has been extensively studied in recent years. The original learned index [26] only supports static data while PGM-index [20], Fiting-tree [21], ALEX [15], LIPP [54], and LIFOSS [61] support dynamic data using different strategies. All these works are designed and optimized for in-

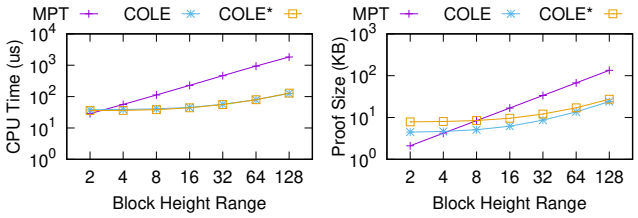

Figure 14: Prov-Query Performance vs. Query Range

memory databases. Bourbon [10] uses the PGM-based models to speed up the lookup in the WiscKey system, which is a persistent key-value store. [27] investigates how existing dynamic learned indexes perform on-disk and shows the design choices. Some other learned indexes are proposed for more complex application scenarios like spatial data [23, 32, 47], multi-dimensional data [16, 39], and variable-length string data [51]. Moreover, [30, 34] consider designing learned indexes for concurrent systems. [64] proposes a persistent learned index that is specifically designed for the NVM-only architecture with concurrency control. More recently, [31] designs a scalable RDMA-oriented learned key-value store for disaggregated memory systems. Nevertheless, existing works cannot be directly applied to blockchain storage since they do not take into account disk-optimized storage, data integrity, and provenance queries simultaneously.

9.2 Blockchain Storage Management

Pioneering blockchain systems, such as Bitcoin [38] and Ethereum [52], use MPT and store it using simple key-value storage like RocksDB [18], which implements the LSM-tree structure. While many works propose to optimize the generic LSM-tree for high throughput and low latency [12–14, 46, 60], and some propose orthogonal designs that could potentially be incorporated into COLE, they are not specifically designed to meet the unique integrity and provenance requirements of blockchain systems. On the other hand, a large body of research has been carried out to study alternative solutions to reduce blockchain storage overhead. Several studies [11, 19, 24, 25, 62] consider using *sharding* techniques to horizontally partition the blockchain storage and each partition is maintained by a subset of nodes, thus reducing the overall storage overhead. Distributed data storage [43, 59] or moving on-chain states to off-chain nodes [6, 8, 48, 56, 58] has also been proposed to reduce each blockchain node's storage overhead. Besides, ForkBase [50] proposes to optimize blockchain storage by deduplicating multi-versioned data and supporting efficient fork operations. [28] employs a vector commitment protocol and multi-level authenticated trees to reduce I/O costs for blockchain storage. To the best of our knowledge, COLE is the first work that targets the index itself to address the blockchain storage overhead.

Another related topic is to support efficient queries in blockchain systems. LineageChain [44] focuses on provenance queries in the blockchain. Verifiable boolean range queries are studied in vChain and vChain+ [49, 55], where accumulator-based authenticated data structures are designed. GEM^2-tree [63] explores query processing in the context of on-chain/off-chain hybrid storage. FalconDB [42] combines the blockchain and the collaborative database to support SQL queries with a strong security guarantee. [57] studies the authenticated spatial and keyword queries in blockchain databases. iQuery [35] supports intelligent blockchain analytical queries and guarantees the trustworthiness of query results by using multiple service providers. While all these works focus on proposing additional data structures to process specific queries, COLE focuses on improving the performance of the general blockchain storage system.

10 Conclusion

In this paper, we have designed COLE, a novel column-based learned storage for blockchain systems. COLE follows the column-based database design to contiguously store each state's historical values using an LSM-tree approach. Within each run of the LSM-tree, a disk-optimized learned index has been designed to facilitate efficient data retrieval and provenance queries. Moreover, a streaming algorithm has been proposed to construct Merkle files that are used to ensure blockchain data integrity. In addition, a new checkpoint-based asynchronous merge strategy has been proposed to tackle the long-tail latency issue for data writes in COLE. Extensive experiments show that, compared with the existing systems, the proposed COLE system reduces the storage size by up to 94% and improves the system throughput by 1.4×-5.4×. Additionally, the proposed asynchronous merge decreases the long-tail latency by 1-2 orders of magnitude while maintaining a comparable storage size.

For future work, we plan to extend COLE to support blockchain systems that undergo forking, where the states of a forked block can be rewound. We will investigate efficient strategies to remove the rewound states from storage. Furthermore, since the column-based design stores blockchain states contiguously, compression techniques can be applied to take advantage of similarities between adjacent data. We will study how to incorporate compression strategies into the learned index.

Acknowledgments

This work is supported by Hong Kong RGC Grants (Project No. 12200022, 12201520, C2004-21GF). Jianliang Xu is the corresponding author.

References

[1] Ethereum full node sync (archive) chart. https://etherscan.io/chartsync/chainarchive, 2023.

[2] Ethereum virtual machine. https://github.com/rust-blockchain/evm, 2023.

[3] rug library. https://docs.rs/rug, 2023.

[4] Daniel J Abadi, Peter A Boncz, and Stavros Harizopoulos. Column-oriented database systems. *Proceedings of the VLDB Endowment*, pages 1664–1665, 2009.

[5] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15, 2018.

[6] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In *Annual International Cryptology Conference*, pages 561–586, 2019.

[7] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, pages 398–461, 2002.

[8] Alexander Chepurnoy, Charalampos Papamanthou, Shravan Srinivasan, and Yupeng Zhang. Edrax: A cryptocurrency with stateless transaction validation. *Cryptology ePrint Archive*, 2018.

[9] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[10] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. From WiscKey to bourbon: A learned index for Log-Structured merge trees. In *OSDI*, pages 155–171, 2020.

[11] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 international conference on management of data*, pages 123–140, 2019.

[12] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *ACM SIGMOD*, pages 79–94, New York, NY, USA, 2017.

[13] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *ACM SIGMOD*, pages 505–520, 2018.

[14] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. Spooky: granulating lsm-tree compactions correctly. *PVLDB*, pages 3071–3084, 2022.

[15] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. ALEX: an updatable adaptive learned index. In *ACM SIGMOD*, pages 969–984, 2020.

[16] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. *PVLDB*, page 74–86, 2020.

[17] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. Blockbench: A framework for analyzing private blockchains. In *ACM SIGMOD*, pages 1085–1100, 2017.

[18] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Trans. Storage*, pages 1–32, 2021.

[19] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. BlockchainDB: A shared database on blockchains. *PVLDB*, pages 1597–1609, 2019.

[20] Paolo Ferragina and Giorgio Vinciguerra. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *PVLDB*, pages 1162–1175, 2020.

[21] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. Fiting-tree: A data-aware index structure. In *ACM SIGMOD*, pages 1189–1206, 2019.

[22] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th symposium on operating systems principles*, pages 51–68, 2017.

[23] Tu Gu, Kaiyu Feng, Gao Cong, Cheng Long, Zheng Wang, and Sheng Wang. The rlr-tree: A reinforcement learning based r-tree for spatial data. In *ACM SIGMOD*, 2023.

[24] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. ResilientDB: Global scale resilient blockchain fabric. *PVLDB*, page 868–883, 2020.

[25] Zicong Hong, Song Guo, Enyuan Zhou, Wuhui Chen, Huawei Huang, and Albert Zomaya. Gridb: Scaling blockchain database via sharding and off-chain cross-shard mechanism. *PVLDB*, pages 1685–1698, 2023.

[26] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *ACM SIGMOD*, pages 489–504, 2018.

[27] Hai Lan, Zhifeng Bao, J Shane Culpepper, and Renata Borovica-Gajic. Updatable learned indexes meet disk-resident dbms-from evaluations to design choices. *Proceedings of the ACM on Management of Data*, pages 1–22, 2023.

[28] Chenxing Li, Sidi Mohamed Beillahi, Guang Yang, Ming Wu, Wei Xu, and Fan Long. {LVMT}: An efficient authenticated storage for blockchain. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 135–153, 2023.

[29] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. Dynamic authenticated index structures for outsourced databases. In *ACM SIGMOD*, pages 121–132, 2006.

[30] Pengfei Li, Yu Hua, Jingnan Jia, and Pengfei Zuo. FINEdex: a fine-grained learned index scheme for scalable and concurrent memory systems. *PVLDB*, pages 321–334, 2021.

[31] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. ROLEX: A scalable RDMA-oriented learned Key-Value store for disaggregated memory systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 99–114, 2023.

[32] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. LISA: A learned index structure for spatial data. In *ACM SIGMOD*, pages 2119–2133, 2020.

[33] Yinan Li, Bingsheng He, Qiong Luo, and Ke Yi. Tree indexing on flash disks. In *IEEE ICDE*, pages 1303–1306, 2009.

[34] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. Apex: a high-performance learned index on persistent memory. *PVLDB*, pages 597–610, 2021.

[35] Lingling Lu, Zhenyu Wen, Ye Yuan, Binru Dai, Peng Qian, Changting Lin, Qinming He, Zhenguang Liu, Jianhai Chen, and Rajiv Ranjan. Iquery: A trustworthy and scalable blockchain analytics platform. *IEEE Transactions on Dependable and Secure Computing*, 2022.

[36] Raghav Mehra, Nirmal Lodhi, and Ram Babu. Column based nosql database, scope and future. *International Journal of Research and Analytical Reviews*, pages 105–113, 2015.

[37] Ralph C Merkle. A certified digital signature. In *Conference on the Theory and Application of Cryptology*, pages 218–238, 1989.

[38] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

[39] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. Learning multi-dimensional indexes. In *ACM SIGMOD*, pages 985–1000, 2020.

[40] Joseph O'Rourke. An on-line algorithm for fitting straight lines between data ranges. *Communications of the ACM*, pages 574–578, 1981.

[41] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, pages 351–385, 1996.

[42] Yanqing Peng, Min Du, Feifei Li, Raymond Cheng, and Dawn Song. Falcondb: Blockchain-based collaborative database. In *ACM SIGMOD*, pages 637–652, 2020.

[43] Xiaodong Qi, Zhao Zhang, Cheqing Jin, and Aoying Zhou. Bft-store: Storage partition for permissioned blockchain via erasure coding. In *IEEE ICDE*, pages 1926–1929, 2020.

[44] Pingcheng Ruan, Gang Chen, Tien Tuan Anh Dinh, Qian Lin, Beng Chin Ooi, and Meihui Zhang. Fine-grained, secure and efficient data provenance on blockchain systems. *PVLDB*, pages 975–988, 2019.

[45] Fahad Saleh. Blockchain without waste: Proof-of-stake. *SSRN Electronic Journal*, 2018.

[46] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. Constructing and analyzing the lsm compaction design space. *PVLDB*, pages 2216–2229, 2021.

[47] Yufan Sheng, Xin Cao, Yixiang Fang, Kaiqi Zhao, Jianzhong Qi, Gao Cong, and Wenjie Zhang. Wisk: A workload-aware learned index for spatial keyword queries. *ACM SIGMOD*, pages 1–27, 2023.

[48] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. Aggregatable subvector commitments for stateless cryptocurrencies. In *International Conference on Security and Cryptography for Networks*, pages 45–64, 2020.

[49] Haixin Wang, Cheng Xu, Ce Zhang, Jianliang Xu, Zhe Peng, and Jian Pei. vChain+: Optimizing verifiable blockchain boolean range queries. In *IEEE ICDE*, pages 1927–1940, 2022.

[50] Sheng Wang, Tien Tuan Anh Dinh, Qian Lin, Zhongle Xie, Meihui Zhang, Qingchao Cai, Gang Chen, Beng Chin Ooi, and Pingcheng Ruan. Forkbase: an efficient storage engine for blockchain and forkable applications. *PVLDB*, pages 1137–1150, 2018.

[51] Youyun Wang, Chuzhe Tang, Zhaoguo Wang, and Haibo Chen. SIndex: a scalable learned index for string keys. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 17–24, 2020.

[52] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014.

[53] Gavin Wood. Polkadot: Vision for a heterogeneous multi-chain framework. *White Paper*, pages 2327–4662, 2016.

[54] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. Updatable learned index with precise positions. *PVLDB*, pages 1276–1288, 2021.

[55] Cheng Xu, Ce Zhang, and Jianliang Xu. vChain: Enabling verifiable boolean range queries over blockchain databases. In *ACM SIGMOD*, pages 141–158, 2019.

[56] Cheng Xu, Ce Zhang, Jianliang Xu, and Jian Pei. SlimChain: scaling blockchain transactions through off-chain storage and parallel processing. *PVLDB*, pages 2314–2326, 2021.

[57] Hao Xu, Bin Xiao, Xiulong Liu, Li Wang, Shan Jiang, Weilian Xue, Jianrong Wang, and Keqiu Li. Empowering authenticated and efficient queries for stk transaction-based blockchains. *IEEE Transactions on Computers*, 2023.

[58] Zihuan Xu and Lei Chen. L2chain: Towards high-performance, confidential and secure layer-2 blockchain solution for decentralized applications. *PVLDB*, pages 986–999, 2022.

[59] Zihuan Xu, Siyuan Han, and Lei Chen. Cub, a consensus unit-based storage scheme for blockchain system. In *IEEE ICDE*, pages 173–184, 2018.

[60] Jinghuan Yu, Sam H Noh, Young-ri Choi, and Chun Jason Xue. ADOC: Automatically harmonizing dataflow between components in Log-StructuredKey-Value stores for improved performance. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 65–80, 2023.

[61] Tong Yu, Guanfeng Liu, An Liu, Zhixu Li, and Lei Zhao. LIFOSS: a learned index scheme for streaming scenarios. *World Wide Web*, pages 1–18, 2022.

[62] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *ACM CCS*, pages 931–948, 2018.

[63] Ce Zhang, Cheng Xu, Jianliang Xu, Yuzhe Tang, and Byron Choi. GEM^2-Tree: A gas-efficient structure for authenticated range queries in blockchain. In *IEEE ICDE*, pages 842–853, 2019.

[64] Zhou Zhang, Zhaole Chu, Peiquan Jin, Yongping Luo, Xike Xie, Shouhong Wan, Yun Luo, Xufei Wu, Peng Zou, Chunyang Zheng, et al. PLIN: a persistent learned index for non-volatile memory with high performance and instant recovery. *PVLDB*, pages 243–255, 2022.

A Artifact Appendix

Abstract

This artifact includes the source code of the proposed system COLE, along with the source code of other baseline systems used for comparison. Additionally, the artifact provides the procedure for generating the dataset under the YCSB benchmark, which is used for evaluating the performance.

Scope

The artifact is an academic proof-of-concept prototype and has not undergone thorough code review. It should be noted that the implementation is not suitable for production use.

Contents

The artifact consists of the following essential directories:

- *cole-index*: This directory contains the implementation of COLE.
- *cole-star*: This directory corresponds to the implementation of the asynchronous version of COLE.
- *patricia-trie*: The MPT implementation can be found in this directory.
- *lipp*: The implementation of LIPP with node persistence is located in this directory.
- *non-learn-cmi*: This directory contains the implementation of CMI, as mentioned in Section 8.
- *exp*: The evaluation backend for all systems is included in this directory.

Hosting

The artifact is hosted on a GitHub repository with the master branch and the latest commit version.

Requirements

The artifact has been evaluated on Ubuntu 20.04 LTS. Please keep in mind that the scripts provided in the README file for installing dependencies may differ for other platforms.

B Appendix

Provenance Query Algorithm

Algorithm 8 shows the procedure of the provenance query. First, we compute two boundary compound keys $\mathcal{K}_l = \langle addr, blk_l - 1\rangle$, $\mathcal{K}_u = \langle addr, blk_u + 1\rangle$ (Line 2). The offsets by one are needed to ensure that no valid results will be missing. Then, similar to the get query, we traverse both MB-trees in L_0 to find the results in the query range (Line 4). At the same time, the corresponding MB-tree paths are added to π as the Merkle proof (Line 5). If we find a state whose block height is smaller than blk_l, we stop the search since all states in the following levels must be even older than blk_l (Lines 6 to 8). Otherwise, we continue to search the on-disk runs in the same order as those described in Algorithm 6. We use $\mathcal{K}_l$ as the search key when applying the learned models to find the first query result in each run (Line 13). Afterwards, we sequentially scan the value file until the state is outside of the query range based on $\mathcal{K}_u$ (Lines 14 to 17). A Merkle proof is computed accordingly based on the position of the first and the last results in the value file of this run. This proof is added to π (Line 18). Similar to the in-memory level, we apply an early stop when we find a state's block height is smaller than blk_l (Line 19). Finally, the root hashes of the unsearched runs in root_hash_list are added to π (Line 20).

Algorithm 8: Provenance Query

```
1  Function ProvQuery(addr, [blk_l, blk_u])
       Input: State address addr, block height range [blk_l, blk_u]
       Output: Result set R, proof π
2      K_l ← ⟨addr, blk_l − 1⟩; K_u ← ⟨addr, blk_u + 1⟩;
3      foreach g in {L_0's writing group, L_0's merging group} do
4          ⟨R', π'⟩ ← SearchMBTree(g, [K_l, K_u]);
5          R.add(R'); π.add(π');
6          if min({r.blk | r ∈ R'}) < blk_l then
7              π.add(remaining of root_hash_list);
8              return ⟨R, π⟩;
9      foreach level i in {1, 2, ...} do
10         RS ← {R_{i,j} | R_{i,j} ∈
             L_i's writing group ∧ R_{i,j} is committed};
11         RS ← RS + {R_{i,j} | R_{i,j} ∈ L_i's merging group};
12         foreach R_{i,j} in RS do
13             ⟨⟨K', state'⟩, pos_l⟩ ← SearchRun(R_{i,j}, K_l);
14             pos_u ← pos_l;
15             while F_V[pos_u].k ≤ K_u do
16                 R.add(F_V[pos_u]);
17                 pos_u ← pos_u + 1;
18             π.add(MHT proof w.r.t. pos_l to pos_u);
19             if K'.blk < blk_l then
20                 π.add(remaining of root_hash_list);
21                 return ⟨R, π⟩;
22     return ⟨R, π⟩;
```

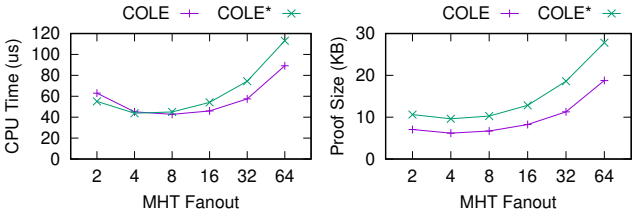

Figure 15: Impact of COLE's MHT Fanout

Impact of COLE'S MHT Fanout

Figure 15 shows the CPU time and proof size under 10^5 block height and $q = 16$ when varying COLE's MHT fanout m. We observe a U-shaped trend for both the CPU time and proof

size with the increasing fanout. The reason is that as the fanout increases, the MHT height decreases, resulting in shorter CPU time and smaller proof size. However, the size of each node of MHT increases, which may lead to longer CPU time and larger proof size (as shown in Table 1). We find that setting $m = 4$ yields the best results for both COLE and COLE*.