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

Your N. Here
*Your Institution*

Second Name
*Second Institution*

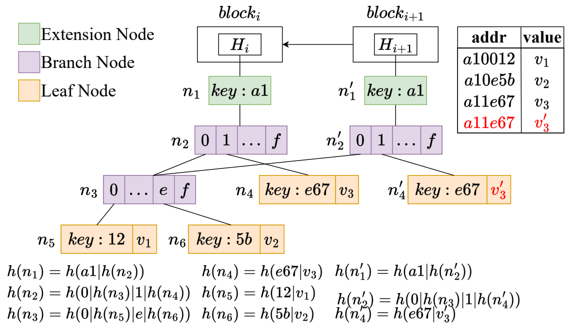
## Abstract

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), that is used to guarantee data integrity and support provenance queries. To reduce the index storage overhead, 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 blockchain's persistence requirement and the learned index's large node size. To address these challenges, 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, backbone of cryptocurrencies and decentralized applications [33, 46], is an immutable ledger formed by consensus among distrustful 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 must store the complete blockchain data, leading to amplified storage expenses, particularly as the blockchain continues to expand. For example, Ethereum's blockchain requires about 14TB storage as of April 2023 [1]. This sizable requirement could hinder participation of resource-limited nodes, potentially affecting system security.

To tackle the storage concerns, we investigate Ethereum's index, Merkle Patricia Trie (MPT), to identify the storage bottleneck. MPT combines Patricia Trie with Merkle Hash Tree (MHT) [32] for data integrity and persists index nodes during data updates to facilitate provenance queries. As shown in Figure 1, MPT includes 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



| addr | value |
|---|---|
| $a10012$ | $v_1$ |
| $a10e5b$ | $v_2$ |
| $a11e67$ | $v_3$ |
| $a11e67$ | $v'_3$ |

Figure 1: An Example of Merkle Patricia Trie

secures data integrity through cryptographic hash function collision-resistance and 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 $i$, one traverses nodes $n_1, n_2, n_4$ to get value $v_3$).

However, this approach significantly increases 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 originates from the index rather than the underlying data. In a preliminary experiment with 10 million data updates, we noted the underlying data contributes merely 4.2% of total storage. Thus, a more compact index supporting data integrity and provenance queries is imperative.

Recently, a novel indexing technique, learned index [14, 19, 24, 48], has emerged, demonstrating notably reduced index size and faster query times compared to traditional indexes. This enhancement is achieved by substituting directing keys in index nodes with a learned model, which not only incurs lower storage costs but also computes the position of searched data more efficiently. 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 data patterns.

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 involves integrating the learned index with MHT [32] and persisting index nodes, akin to MPT. Nonetheless, this isn't feasible due to the larger learned index node size. 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, resulting in large node size. 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$\times$ to 31$\times$ 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. Nevertheless, as highlighted in [25], directly applying existing learned indexes to disk storage may yield poorer write-intensive performance compared to traditional B+-tree. 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 addresses the limitations of current learned indexes and supports provenance queries. The key challenge in adapting learned indexes to blockchains is persistent node requirement, leading to substantial storage overhead. COLE tackles this issue with an innovative *column-based* design, inspired by column-based databases [4, 31]. Each 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 block heights (state version numbers). Moreover, historical data queries bypass traversing previous block indexes, instead utilizing 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 adopt the *log-structured merge-tree* (LSM-tree) [29, 36] 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, enabling 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 validates the entire blockchain data. However, recursive merges during write operations can lead to long-tail latency in the LSM-tree approach. To mitigate this, we introduce a novel checkpoint-based asynchronous merge strategy, ensuring storage synchronization 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$\times$-5.4$\times$. 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 present the preliminary information essential for introducing COLE. Blockchain is a chain of blocks holding states and recording transactions that modify these states. The transaction's execution logic is termed a *smart contract*. These contracts manage states identified by a state address $addr$. In Ethereum [46], both $addr$ and the state value $value$ are fixed-sized strings. Figure 2 illustrates the block's structure. A block's header includes: (i) $H_{prev\_blk}$, the previous block's hash; (ii) $TS$, the timestamp; (iii) $\pi_{cons}$, consensus protocol related data; (iv) $H_{tx}$, transaction root digest; and (v) $H_{state}$, state root digest. The block body includes transactions, states, and corresponding Merkle hash trees (MHTs).

MHT, a prevalent hierarchical structure, upholds data integrity [32]. As shown in Figure 2, leaf nodes are hash values of indexed data (e.g., $h_1 = h(tx_1)$), while internal nodes are hash values of their child nodes (e.g., $h_5 = h(h_1||h_2)$). MHT can validate the existence of a data record. For example, to prove $tx_3$, $h_4$ and $h_5$ (shaded in Figure 2), which are the sibling hashes of the search path, 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 block header's version (i.e., $H_{tx}$). Apart from being used in the blockchain, MHT has been extended to database indexes to support result integrity verification for different

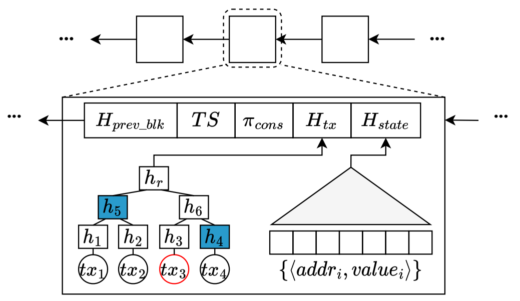


Figure 2: Block Data Structure

queries. For example, Merkle B+-tree (MB-tree) combines MHT and B+-tree structures, ensuring secure queries in relational databases [26].

Blockchain storage uses an index for efficient state maintenance and access [44, 46]. Apart from standard index's operations (i.e., read and write), the blockchain index should fulfill two requirements we mentioned before: (i) ensuring the *integrity* of the blockchain states, (ii) supporting *provenance queries* that ensure the integrity of historical state queries. With these requirements, the blockchain index should support the following functions:

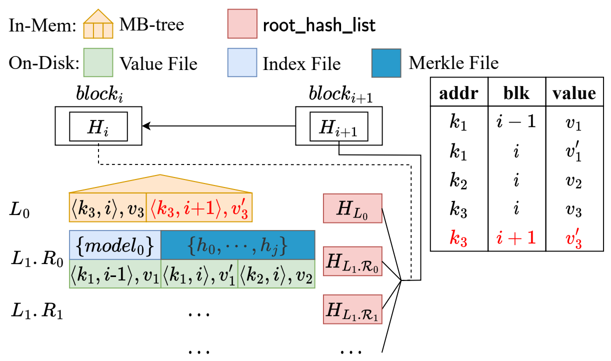
- `Put`$(addr, value)$: insert the state with the address *addr* and the value *value* to the current block;
- `Get`$(addr)$: return the *latest* value of the state at address *addr* if it exists, or returns *nil* otherwise;
- `ProvQuery`$(addr, [blk_l, blk_u])$: return the provenance query results $\{value\}$ and a proof $\pi$, 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.

Additionally, we focus on non-forking blockchain systems [5, 20, 47], which means that the blockchain storage does not need to support the rewind function.

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 $i+1$. 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 $\pi$ and checks whether the reconstructed one matches the public digest $H_i$ in block $i$ and whether the search path in $\pi$ 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.



| addr | blk | value |
|---|---|---|
| $k_1$ | $i-1$ | $v_1$ |
| $k_1$ | $i$ | $v_1'$ |
| $k_2$ | $i$ | $v_2$ |
| $k_3$ | $i$ | $v_3$ |
| $k_3$ | $i+1$ | $v_3'$ |

Figure 3: Overview of COLE

### 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 for achieving better performance.

### 3.2 Design Overview

Figure 3 shows the overview of COLE. Following the column-based design [4, 31], 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, its value and corresponding version (block height) are appended to the index housing all historical versions. 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 $\mathcal{K}_3' \leftarrow \langle k_3, i+1\rangle$, is created. This facilitates the insertion of updated value $v_3'$ into COLE, positioned adjacent to the prior version $v_3$ of $k_3$. Compared with the MPT in Figure 1, the column-based design circumvents the node duplication along the update path (e.g., $n_1, n_2, n_4$ and $n_1', n_2', n_4'$) and saves 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 exponential increasing sizes. New data is initially added to the first level. If the level reaches its predefined capacity, the data is merged into a sorted run in the next level. The merging can recur until the capacity requirement is met. The first level, often highly dynamic, is typically stored in memory, while other levels reside on disk. COLE employs Merkle B+-tree (MB-tree) [26] 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 initial on-disk level.

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

- **Value file** stores blockchain states as compound key-value pairs, 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 [19], 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.

To ensure blockchain 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 $max\_int$ as the maximum integer. That is, the search is stopped as long as a state value with the queried address $addr_q$ is found.

## 4 Write Operation of COLE

We now detail the write operation of COLE. As mentioned in Section 3.2, COLE organizes the storage by using LSM-tree, which consists of one 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. The first on-disk level contains up to $T$ sorted runs with the size of each run being $B$. Each subsequent on-disk level also contains up to $T$ sorted runs, but the size of each run grows exponentially with a ratio of $T$. That is, the maximum capacity of level $i$ is $B \cdot T^i$.

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

**Algorithm 1:** Write Algorithm

```
1  Function Put (addr, value)
       Input: State address $addr$, value $value$
2      $blk \leftarrow$ current block height; $\mathcal{K} \leftarrow \langle addr, blk \rangle$;
3      Insert $\langle \mathcal{K}, value \rangle$ 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 $\mathcal{F}_V, \mathcal{F}_I, \mathcal{F}_H$ for this run;
7          $i \leftarrow 1$;
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 $\mathcal{F}_V, \mathcal{F}_I, \mathcal{F}_H$ for the new run;
11             Remove all the runs in $L_i$;
12             $i \leftarrow i + 1$;
13     Update $H_{state}$ when finalizing the current block;
```

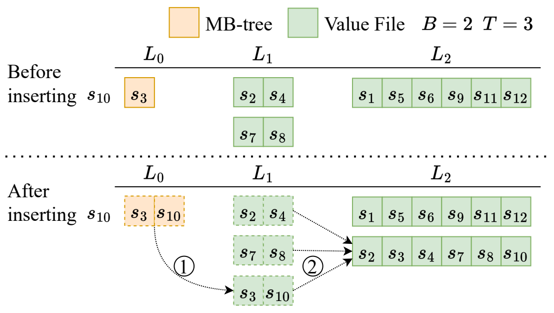


Figure 4: An Example of Write Operation

the MB-tree (Line 3). As $L_0$ fills up, it's 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 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 are constructed in a *streaming* manner (see Section 4.1, Section 4.2 for details). When on-disk level $L_i$ fills up (with $T$ runs), $L_i$'s all the runs are merge-sorted as a new run in the next level $L_{i+1}$, with corresponding three files generated (Lines 8 to 12). 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 $T = 3$, run sizes of $L_1$ and $L_2$ are 2 and 6 respectively. After $s_{10}$ is inserted into in-memory level $L_0$, the level is full 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.*

To speed up read operations, embedding Bloom filters in both the in-memory MB-tree and on-disk run levels is a common optimization. However, several aspects worth noting when implementing Bloom filters in COLE. Specifically, Bloom filters should be built upon the addresses of the underlying states rather than their compound keys to facilitate the read operation (see Section 6 for details). Moreover, to ensure provenance query's integrity, Bloom filters should be integrated alongside the root hashes of each run when computing the states' root digest.

## 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 [19], 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 $|p_{pred} - p_{real}| \leq \varepsilon$. To generate the models in a disk-friendly manner, we set ε as half the number of models that can fit into a single disk page. 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.

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 arrival of a new compound key, we convert it into a large number through binary concatenation of its components (i.e., $BigNum(\mathcal{K}) = BigNum(addr, blk)$). Next, we find the smallest convex hull encompassing all current input points, which can be calculated incrementally in a streaming manner [35]. Then, we find the minimal parallelogram that covers the convex hull, with one side aligned to the vertical axis (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.

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).

**Algorithm 2:** Learn Models from a Stream

```
1  Function BuildModel(S, ε)
       Input: Input stream $\mathcal{S}$, error bound ε
       Output: A stream of models $\{\mathcal{M}\}$
2      $k_{min} \leftarrow \emptyset$, $p_{max} \leftarrow \emptyset$, $g_{last} \leftarrow \emptyset$;
3      Init an empty convex hull $\mathcal{H}$;
4      foreach $\langle \mathcal{K}, p_{real} \rangle \leftarrow \mathcal{S}$ do
5          if $k_{min} = \emptyset$ then $k_{min} \leftarrow \mathcal{K}$;
6          Add $\langle BigNum(\mathcal{K}), p_{real} \rangle$ to $\mathcal{H}$;
7          Compute the minimum parallelogram $\mathcal{G}$ that covers $\mathcal{H}$;
8          if $\mathcal{G}.height \leq 2\varepsilon$ then
9              $p_{max} \leftarrow p_{real}, g_{last} \leftarrow \mathcal{G}$;
10         else
11             Compute slope $sl$ and intercept $ic$ from $g_{last}$;
12             $\mathcal{M} \leftarrow \langle sl, ic, k_{min}, p_{max} \rangle$;
13             yield $\mathcal{M}$;
14             $k_{min} \leftarrow \mathcal{K}$;
15             Init a new convex hull $\mathcal{H}$ with $\langle BigNum(\mathcal{K}), p_{real} \rangle$;
```

**Algorithm 3:** Index File Construction

```
1  Function ConstructIndexFile(S, ε)
       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($\mathcal{S}$, ε) 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($\mathcal{S}$, ε) and append to $\mathcal{F}_I$;
8          $n \leftarrow$ # of pages in $\mathcal{F}_I - n$;
9      return $\mathcal{F}_I$;
```

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 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 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 sequential storage of models across layers in a bottom-up manner.

## 4.2 Merkle File Construction

A Merkle file contains an $m$-ary complete MHT that authenticates 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

**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 $\langle \mathcal{K}, value \rangle \leftarrow \mathcal{S}$ 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 $\mathcal{F}_H$;
```

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.*

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 size of 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 through 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

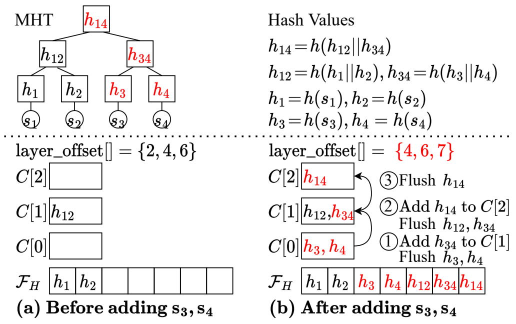

MHT
$h_{14}$
$h_{12}$
$h_{34}$
$h_1$
$h_2$
$h_3$
$h_4$
$s_1$
$s_2$
$s_3$
$s_4$
Hash Values
$h_{14} = h(h_{12} || h_{34})$
$h_{12} = h(h_1 || h_2), h_{34} = h(h_3 || h_4)$
$h_1 = h(s_1), h_2 = h(s_2)$
$h_3 = h(s_3), h_4 = h(s_4)$
layer_offset[] = {2, 4, 6}
$C[2]$
$C[1]$ $h_{12}$
$C[0]$
$\mathcal{F}_H$ $h_1$ $h_2$
(a) Before adding $s_3, s_4$
layer_offset[] = {4, 6, 7}
$C[2]$ $h_{14}$
$C[1]$ $h_{12}, h_{34}$
$C[0]$ $h_3, h_4$
③Flush $h_{14}$
②Add $h_{14}$ to $C[2]$ Flush $h_{12}, h_{34}$
①Add $h_{34}$ to $C[1]$ Flush $h_3, h_4$
$\mathcal{F}_H$ $h_1$ $h_2$ $h_3$ $h_4$ $h_{12}$ $h_{34}$ $h_{14}$
(b) After adding $s_3, s_4$

Figure 5: An Example of Merkle File Construction

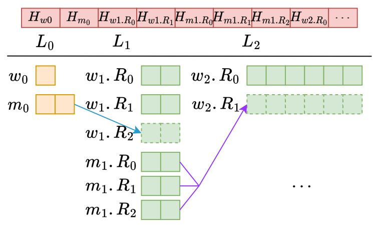
a buffer from the lowest layer and iteratively generating hash values. Each hash value is added to the upper layer before flushing the buffer to the Merkle file (Lines 15 to 18).

**Example.** *Figure 5 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$[] = \{0, 4, 6\}$. Initially, $s_1, s_2$ are added, resulting in $\mathcal{F}_H$ storing $h_1, h_2$ (hash of $s_1, s_2$) and cache $C[1]$ containing $h_{12}$ derived from $h_1, h_2$ (Figure 5(a)). After $s_3, s_4$ are added, their hashes $h_3, h_4$ are inserted into cache $C[0]$, containing 2 hash values afterwards. 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). Then, $C[1]$ contains 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).*

## 5 Write with Asynchronous Merge

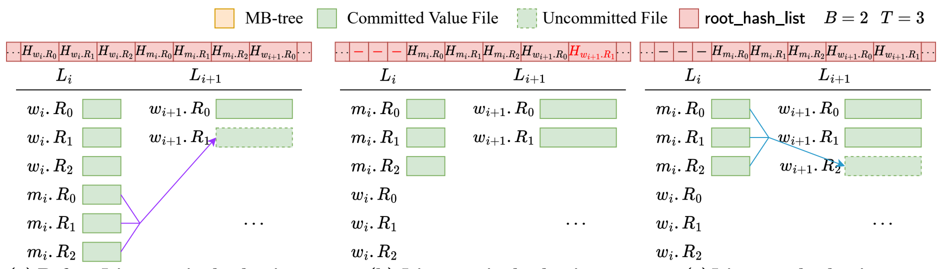
Algorithm 1 may trigger recursive merge operations during writes (e.g., steps ① and ② in Figure 4). This can lead to long-tail latency (a.k.a *write stall*) which causes significant performance fluctuations. A common solution is to make the merge operations asynchronous by moving them to separate threads. However, this is not suitable for blockchain applications due to the potential discrepancy in computational capabilities among different blockchain nodes. Asynchronous merges can cause storage structures to fall out of sync across nodes, resulting in divergent $H_{state}$ values and violating the blockchain protocol requirement.

To address these challenges, we design a novel asynchronous merge algorithm for COLE, which maintains synchronization among various 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 aim to proportionally extend the interval between the start checkpoint and the commit checkpoint based on the size of the run, making majority nodes complete the merge operation before reaching the commit checkpoint.



$H_{w0}$ $H_{m_0}$ $H_{w1.R_0}$ $H_{w1.R_1}$ $H_{m1.R_0}$ $H_{m1.R_1}$ $H_{m1.R_2}$ $H_{w2.R_0}$ ···
$L_0$
$L_1$
$L_2$
$w_0$
$m_0$
$w_1.R_0$
$w_1.R_1$
$w_1.R_2$
$m_1.R_0$
$m_1.R_1$
$m_1.R_2$
$w_2.R_0$
$w_2.R_1$
···

Figure 6: Asynchronous Merge



MB-tree
Committed Value File
Uncommitted File
root_hash_list
$B=2$ $T=3$
$H_{w_i.R_0}$ $H_{w_i.R_1}$ $H_{w_i.R_2}$ $H_{m_i.R_0}$ $H_{m_i.R_1}$ $H_{m_i.R_2}$ $H_{w_{i+1}.R_0}$
$L_i$
$L_{i+1}$
$w_i.R_0$
$w_i.R_1$
$w_i.R_2$
$m_i.R_0$
$m_i.R_1$
$m_i.R_2$
$w_{i+1}.R_0$
$w_{i+1}.R_1$
$w_{i+1}.R_2$
$H_{m_i.R_0}$ $H_{m_i.R_1}$ $H_{m_i.R_2}$ $H_{w_{i+1}.R_0}$ $H_{w_{i+1}.R_1}$
···

(a) Before $L_i$'s commit checkpoint (b) $L_i$'s commit checkpoint (c) $L_i$'s start checkpoint

Figure 7: An Example of Asynchronous Merge

**Algorithm 5:** Write Algorithm with Asynchronous Merge

```
1  Function Put (addr, value)
       Input: State address addr, value value
2      $blk \leftarrow$ current block height; $\mathcal{K} \leftarrow \langle addr, blk \rangle$;
3      $w_0 \leftarrow$ Get $L_0$'s writing group;
4      Insert $\langle \mathcal{K}, value \rangle$ into the MB-tree of $w_0$;
5      $i \leftarrow 0$;
6      while $w_i$ becomes full do
7          $m_i \leftarrow$ 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 $\leftarrow$ 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 $\mathcal{F}_V, \mathcal{F}_I, \mathcal{F}_H$ 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 $\mathcal{F}_V, \mathcal{F}_I, \mathcal{F}_H$ for the new run;
21         $i \leftarrow i + 1$;
22     Update $H_{state}$ when finalizing the current block;
```

To realize our idea, we propose to have each level of COLE contain two groups of runs as shown in Figure 6. The design of each group 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. The maximum capability of level $i$ is $2 \cdot B \cdot T^i$. 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). Levels are then traversed in COLE from top to bottom. When a level is full, the previous merge operation is committed in the current level, and a new merge operation begins in a new thread. Here, if an ongoing merging thread exists, it will be identified and waited for to finish, thereby accommodating slower network nodes (Line 9). The prior 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 removed (Line 12). This procedure ensures synchronized commit checkpoints among network nodes, which is crucial for coherent blockchain states and root digests. Roles of the two groups in the current level are then swapped (Line 13), directing future write operations to the new writing group and merging operations to the full merging group. 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 7 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 7(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 7(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 7(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 $H_{state}$ remains synchronized among blockchain nodes, 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 based on current committed states, independent of individual node performance variations. The second requirement mini-

**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          $\langle \mathcal{K}', state' \rangle \leftarrow$ 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 $\wedge R_{i,j}$ is 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             $\langle\langle \mathcal{K}', state' \rangle, pos' \rangle \leftarrow$ SearchRun($R_{i,j}$, $\mathcal{K}_q$);
11             if $\mathcal{K}'.addr = addr$ then return $state'$;
12     return nil;
```

mizes the likelihood of nodes waiting for merge operations of longer runs. We now prove that our algorithm complies with the requirements.

*Proof Sketch.* 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. 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 verification 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 level, the search starts from the lowest level and progresses upwards until finding the satisfied value. 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). For the example in Figure 6, 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

**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($\mathcal{P}$, $\mathcal{K}_q$);
```

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 in 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 $2\varepsilon$ 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 $\mathcal{P}$ (Lines 15 to 18). This process involves at most two pages for prediction, 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 point search, 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. More-

| 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

over, 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. Specifically, $\mathcal{K}_l$ 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 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, the MHT in the Merkle file is traversed from bottom to up. Note that given a hash value's position $pos$ at layer $i$, its parent hash value's position in the Merkle file is computed directly 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 A .

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 cost. MPT has $O(n \cdot d_{MPT})$ storage due to the node duplication for each insertion, where $d_{MPT}$ is the MPT's height. In contrast, COLE removes the node duplication, resulting in $O(n)$ storage. As for the write IO cost, MPT requires $O(d_{MPT})$ writes for nodes along the update path, while COLE takes $O(d_{COLE})$ for the worst case when all levels are merged ($d_{COLE}$ is the number of levels). COLE's level merge resembles traditional LSM-tree writes, with an amortized $O(1)$ IO cost for the value, index, and Merkle files. Here, $d_{COLE}$ equals $\lceil \log_T(\frac{n}{B} \cdot \frac{T-1}{T}) \rceil$, 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 256-bit key, maximum $d_{MPT}$ is 64 under hexadecimal base while COLE has only a few levels following the LSM-tree).

For the write tail latency, MPT has no write stall, hence taking $O(1)$. In contrast, COLE may have a write stall in the worst case, resulting in reading and writing of $O(n)$ states for the worse case when merging all levels. The asynchronous merge removes the write stall by merging in the background, hence taking $O(1)$. For the write memory footprint, MPT takes $O(1)$ as update nodes are computed on-the-fly and can be discarded after disk flushing. For COLE, the largest level's merge, as the worse-case, requires $O(T)$ memory for sort-merge, $O(1)$ for model construction [35]. 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. 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, we assume each run takes $C_{model}$ to locate the state. Hence, the get query in COLE takes $O(T \cdot d_{COLE} \cdot C_{model})$ as each level's $T$ runs are queried. 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 and 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

[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.

| **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

Section 1.

- LIPP: It applies LIPP [48], 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 contiguous fashion [26].

### 8.1.2 Implementation and Parameter Setting

We implement COLE and the baselines in Rust programming language. 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 4.1 are implemented using the *rug* library [3]. Baselines utilize RocksDB [17] 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. The memory budget of RocksDB is set to 64MB and 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 Intel i7-10710U CPU, 16GB RAM, 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 [16] are used as macro benchmarks to generate the transaction workload. SmallBank simulates the account transfers while KVStore uses YCSB [8] 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 scenario: (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 [39]'s setting. The evaluation metrics include (i) CPU time of query and verification, the time that the query is executed on the blockchain node and verified by the querying user, and (ii) proof size.

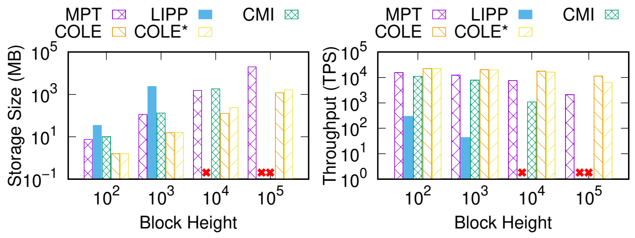


Figure 8: Performance vs. Block Height (SmallBank)

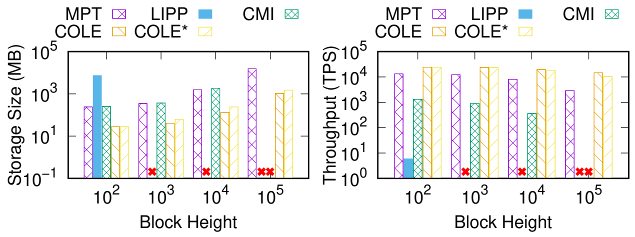


Figure 9: Performance vs. Block Height (KVStore)

## 8.2 Experimental Results

### 8.2.1 Overall Performance

Figures 8 and 9 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.

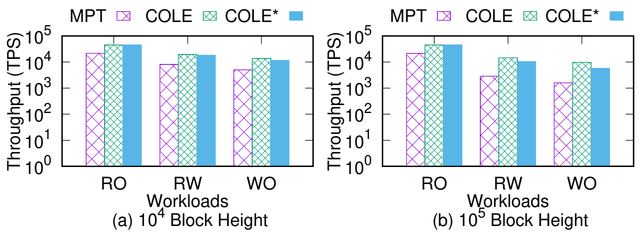


Figure 10: Throughput vs. Workloads (KVStore)

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 $5\times$ (for SmallBank) and $31\times$ (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.

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 $7\times$ and $22\times$ 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 10, 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 10 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 11 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.

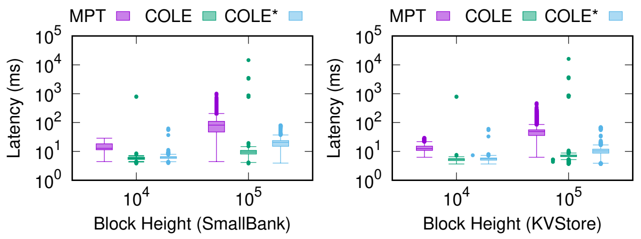


Figure 11: Latency Box plot

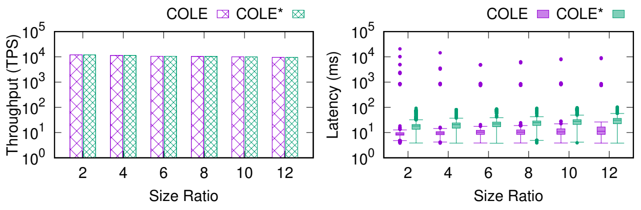


Figure 12: Impact of Size Ratio

### 8.2.4 Impact of Size Ratio

Figure 12 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 increasing size ratio, the median latency of both COLE and COLE* increases.

### 8.2.5 Provenance Query Performance

We now evaluate the provenance query performance 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 13 shows that MPT's CPU time and proof size grow linearly with $q$ while those of COLE and COLE* exhibit sublinear growth. This is because MPT requires to query each block within the specified range. Conversely, 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 reducing 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.

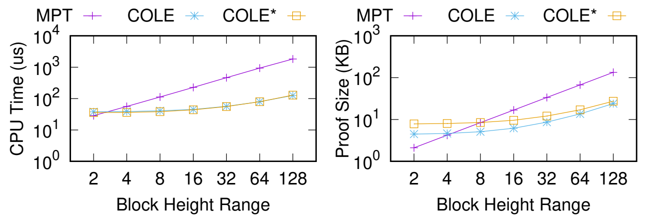


Figure 13: Prov-Query Performance vs. Query Range

## 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 [24] only supports static data while PGM-index [19], ALEX [14], LIPP [48], and LIFOSS [53] support dynamic data using different strategies. All these works are designed and optimized for in-memory databases. Bourbon [9] uses the PGM-based models to speed up the lookup in the WiscKey system, which is a persistent key-value store. [25] investigates how existing dynamic learned indexes perform on-disk and shows the design choices. There are some other learned indexes that are proposed for more complex application scenarios. LISA [28], RLR-Tree [21], Flood [34], and SIndex [45] target indexing dynamic spatial data, multi-dimensional data, and variable-length string data, respectively. XIndex [41] investigates the scenario of multi-core data storage. Tsunami [15] proposes a multi-dimensional learned index with correlated data and skewed workloads. A fine-grained learned index, FINEdex [27], is developed for scalable and concurrent memory systems. APEX [30] designs a persistent-memory-optimized learned index to support not only high performance but also concurrency and instant recovery. PLIN [56] is a persistent learned index that is specifically designed for the NVM-only architecture. 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 [33] and Ethereum [46], use MPT and store it using simple key-value storage like RocksDB [17], which implements the LSM-tree structure. While many works propose to optimize the generic LSM-tree for high throughput and low latency [11–13, 40], 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 [10, 18, 22, 23, 54] 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 [38, 52] or moving on-chain states to off-chain nodes [6, 7, 42, 50, 51] has also been proposed to reduce each blockchain node's storage overhead. Besides, ForkBase [44] proposes to optimize blockchain storage by deduplicating multi-versioned data and supporting efficient fork operations. 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 [39] focuses on provenance queries in the blockchain. Specifically, a Merkle DAG combined with a deterministic append-only skip list is embedded in the blockchain storage to speed up provenance queries. Verifiable boolean range queries are studied in vChain and vChain+ [43,49], where accumulator-based authenticated data structures are designed. GEM$^2$-tree [55] explores query processing in the context of on-chain/off-chain hybrid storage. FalconDB [37] combines the blockchain and the collaborative database to support SQL queries with a strong security guarantee. 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. Specifically, 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 $1.4\times$-$5.4\times$. 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.

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

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

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

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

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

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

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

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

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

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

[38] 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.

[39] 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.

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

[41] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. XIndex: a scalable learned index for multicore data storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 308–320, 2020.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

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

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

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

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

[50] 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.

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

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

[53] 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.

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

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

[56] 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 Appendix

### A.1 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 $\pi$ 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 $\pi$ (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 $\pi$ (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 $\pi$
2      $\mathcal{K}_l \leftarrow \langle addr, blk_l - 1\rangle$; $\mathcal{K}_u \leftarrow \langle addr, blk_u + 1\rangle$;
3      foreach $g$ in {$L_0$'s writing group, $L_0$'s merging group} do
4          $\langle R', \pi'\rangle \leftarrow$ SearchMBTree($g, [\mathcal{K}_l, \mathcal{K}_u]$);
5          $R.add(R')$; $\pi.add(\pi')$;
6          if $\min(\{r.blk | r \in R'\}) < blk_l$ then
7              $\pi.add$(remaining of root_hash_list);
8              return $\langle R, \pi\rangle$;
9      foreach level $i$ in $\{1, 2, \dots\}$ do
10         $RS \leftarrow \{R_{i,j} \mid R_{i,j} \in$ $L_i$'s writing group$\wedge R_{i,j}$ is committed$\}$;
11         $RS \leftarrow RS + \{R_{i,j} \mid R_{i,j} \in L_i$'s merging group$\}$;
12         foreach $R_{i,j}$ in $RS$ do
13             $\langle\langle \mathcal{K}', state'\rangle, pos_l\rangle \leftarrow$ SearchRun($R_{i,j}, \mathcal{K}_l$);
14             $pos_u \leftarrow pos_l$;
15             while $\mathcal{F}_V[pos_u].k \leq \mathcal{K}_u$ do
16                 $R.add(\mathcal{F}_V[pos_u])$;
17                 $pos_u \leftarrow pos_u + 1$;
18             $\pi.add$(MHT proof w.r.t. $pos_l$ to $pos_u$);
19             if $\mathcal{K}'.blk < blk_l$ then
20                 $\pi.add$(remaining of root_hash_list);
21                 return $\langle R, \pi\rangle$;
22     return $\langle R, \pi\rangle$;
```

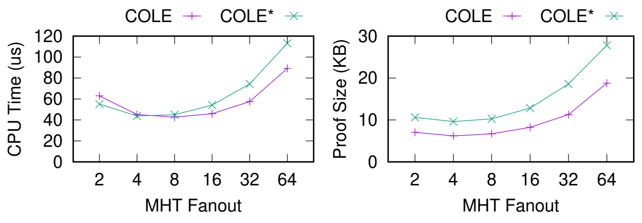


Figure 14: Impact of COLE's MHT Fanout

#### A.1.1 Impact of COLE'S MHT Fanout

Figure 14 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*.