COLE$^+$: Towards Practical Column-based Learned Storage for Blockchain Systems (Technical Report)

Ce Zhang[*], Cheng Xu[*], Haibo Hu[‡], Jianliang Xu[*]

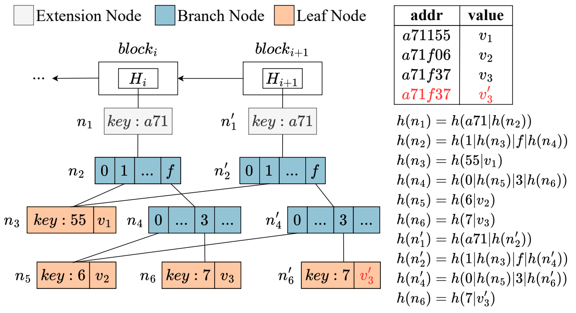
[*] *Department of Computer Science, Hong Kong Baptist University, Hong Kong SAR*
[‡] *Department of Electrical and Electronic Engineering, Hong Kong Polytechnic University, Hong Kong SAR*
{cezhang, chengxu, xujl}@comp.hkbu.edu.hk, haibo.hu@polyu.edu.hk

***Abstract*—Blockchain provides a decentralized and tamper-resistant ledger for securely recording transactions across a network of untrusted nodes. While its transparency and integrity are beneficial, the substantial storage requirements for maintaining a complete transaction history present significant challenges. For example, Ethereum nodes require around 23TB of storage, with an annual growth rate of 4TB. Prior studies have employed various strategies to mitigate the storage challenges. Notably, COLE significantly reduces storage size and improves throughput by adopting a column-based design that incorporates a learned index, effectively eliminating data duplication in the storage layer. However, this approach has limitations in supporting chain reorganization during blockchain forks and state pruning to minimize storage overhead. In this paper, we propose COLE$^+$, an enhanced storage solution designed to address these limitations. COLE$^+$ incorporates a novel rewind-supported in-memory tree structure for handling chain reorganization, leveraging content-defined chunking (CDC) to maintain a consistent hash digest for each block. For on-disk storage, a new two-level Merkle Hash Tree (MHT) structure, called prunable version tree, is developed to facilitate efficient state pruning. Both theoretical and empirical analyses show the effectiveness of COLE$^+$ and its potential for practical application in real-world blockchain systems.**

I. Introduction

Blockchain, a decentralized and append-only ledger, leverages cryptographic hash chains and distributed consensus protocols to securely record transactions across a network of untrusted nodes [1], [2]. Its inherent transparency and tamper-resistance have established it as a foundational technology for cryptocurrencies and a wide range of decentralized applications. To ensure complete and verifiable data provenance, blockchain nodes must maintain a comprehensive history of transactions and ledger states, enabling users to query historical data with strong integrity guarantees. However, this comprehensive record-keeping incurs substantial storage overhead, which grows rapidly as the blockchain expands. For example, as of October 2025, Ethereum nodes require approximately 23 TB of storage, with an annual growth rate of ∼4 TB [3].

Prior studies [4], [5] show that the excessive storage overhead stems from the underlying index, Merkle Patricia Trie (MPT) [2]. MPT retains obsolete nodes during data updates to enable data provenance via pointer chasing. For example, as shown in Figure 1, updating address $a71f37$ with value v_3' in block $i+1$ adds a new path (n_1', n_2', n_4', n_6') for the updated value while retaining the existing path (n_1, n_2, n_4, n_6) for the historical value v_3. This design allows access to any historical



addr	value
$a71155$	v_1
$a71f06$	v_2
$a71f37$	v_3
$a71f37$	v_3'

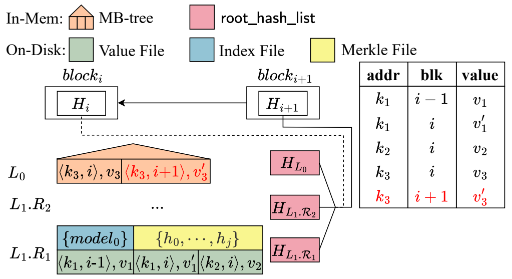
$h(n_1) = h(a71|h(n_2))$
$h(n_2) = h(1|h(n_3)|f|h(n_4))$
$h(n_3) = h(55|v_1)$
$h(n_4) = h(0|h(n_5)|3|h(n_6))$
$h(n_5) = h(6|v_2)$
$h(n_6) = h(7|v_3)$
$h(n_1') = h(a71|h(n_2'))$
$h(n_2') = h(1|h(n_3)|f|h(n_4'))$
$h(n_4') = h(0|h(n_5)|3|h(n_6'))$
$h(n_6) = h(7|v_3')$

Fig. 1. An Example of Merkle Patricia Trie

value by traversing the index nodes under a specific historical block (e.g., traversing n_1, n_2, n_4, n_6 retrieves v_3 in block i). However, it also introduces significant storage overhead due to duplication along the updated path (e.g., n_1', n_2', n_4', n_6', and n_1, n_2, n_4, n_6).

Existing work has explored various strategies to address the storage challenges posed by MPT. LETUS [4] uses delta-encoding to reduce the storage requirements of the blockchain index. It also adopts page-based, simple file storage instead of key-value databases (e.g., RocksDB [6]) to enable fine-grained I/O optimizations. Compared with MPT, LETUS achieves up to 4× storage reduction, as reported in [4]. However, its proposed index, DMM-Tree, still employs an MPT-like node duplication strategy during data updates, leaving room for further storage optimization. SlimArchive [7] opts to eliminate the Merkle-based structures and flattens the minimal blockchain state changes. However, this approach no longer supports data authentication and provenance, which are critical security features of blockchain systems.

In contrast, COLE employs a column-based design coupled with a learned index to significantly reduce storage overhead (by up to 5.8× against MPT) while maintaining efficient data access [5]. Figure 2 illustrates the overall design of COLE, where a log-structured merge-tree (LSM-tree) consisting of multiple sorted runs is used to manage blockchain states. For each on-disk sorted run, COLE uses a value file to store the ledger states as a database column, an index file to predict the states' positions in the value file via a tailored learned index, and a Merkle file to ensure the data integrity of the value file. COLE's column-based design, in which successive versions of each state are stored contiguously, enables efficient data retrieval and reduces storage size. The learned index



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'

Fig. 2. Structure of COLE [5]

leverages the inherent ordering of the data for fast data access. At the same time, maintaining the historical states within the index of the latest block eliminates MPT tree node duplication, thereby yielding substantial storage savings and enabling rapid historical state retrieval without the need to traverse prior block indexes.

Despite COLE's effectiveness in reducing storage size and enhancing system throughput, several limitations hinder its wide practical deployment:

- **Limited Chain Reorganization Support**: COLE struggles to handle chain reorganizations during blockchain forks [8]. These forks can arise from either consensus protocols (e.g., Proof of Work [1]) or software upgrades (e.g., post-Ethereum DAO attack hard fork [9]). The former, which occurs frequently, typically rewinds a few recent blocks and thus requires COLE's in-memory states to revert to a previous version. However, this is infeasible because its underlying in-memory Merkle B-tree structure depends on both the data values and their insertion order [10], [11]. Rewinding blockchain states in this manner would introduce inconsistencies in the tree index across nodes and violate the requirement for a globally agreed-upon root hash. In the latter, rarer case, reversing the index is also difficult due to modifications in on-disk runs caused by flush and recursive merge operations in the LSM-tree. As a result, COLE is limited to blockchain systems that do not fork [12], [13], [14].
- **Lack of State Pruning Support**: COLE does not support state pruning, a common technique used by blockchain full nodes to reduce storage overhead by retaining only recent state versions. Because COLE's Merkle file constructs a complete Merkle Hash Tree (MHT) over all historical state versions for the corresponding value file, pruning historical states would disrupt its ability to correctly construct the Merkle file and compute the root hash for the new compacted run during subsequent LSM-tree merge operations. This makes it impossible for full archive nodes and pruned nodes to agree on a consistent tree index and root hash for blockchain states. Consequently, all blockchain full nodes in COLE must maintain all historical states, even those that are rarely queried.
- **Uniform Indexing of State Versions**: COLE treats all state versions equally, neglecting to account for their disparate access frequencies. Since blockchain queries predominantly access the latest state values, this uniform approach introduces inefficiencies by unnecessarily expanding the search space from the current state to all historical versions during data queries.

To address the aforementioned limitations of COLE while retaining the benefits of its column-oriented design and learned index, this paper proposes COLE$^+$, a novel system that supports chain reorganization and efficient state pruning through novel storage layouts and innovative Merkle index designs.

Specifically, to enable efficient chain reorganization of recent blocks, we propose a novel in-memory *rewind-supported tree* (RS-tree). It integrates the content-defined chunking (CDC) concept [15], [16] to ensure a deterministic structure, yielding a consistent root hash for state rewinds and appends that is independent of the update order. To prevent flushing all states to disk and the possible rebuilding of on-disk runs after an LSM-tree flush, we maintain two groups of in-memory RS-tree instances along with temporary hash lists acting as checkpoints for each LSM-tree run before flushing. These hash lists allow us to retrieve the original hashes for pre-existing on-disk runs (those present before the flush) and enable computing the new root hash as if no reorganization had occurred. For rare, deeper reorganizations extending to arbitrary on-disk levels, our approach anchors the rewind point at the latest checkpoint before the common ancestor block being shared with the canonical chain. Only LSM-tree runs altered by this rewind are rebuilt, and subsequent canonical blocks are appended normally.

To facilitate efficient state pruning, we adopt a two-level MHT structure that separates the latest and historical versions of each state. Specifically, the lower level uses a specially designed prunable *version tree* to index the historical versions of each state. The upper level consists of a complete MHT that incorporates the latest value of each state along with the corresponding version tree from the lower level. This two-level design also streamlines the retrieval of the latest states by narrowing the search space. The version tree facilitates pruning through a purposefully designed CDC algorithm for index construction, which ensures a deterministic index structure even when many states have been pruned. During disk-level merge sort operations, our custom CDC enables the construction of a new merged version tree using only the left-most and right-most boundary paths from the two merging trees, thereby safely pruning all intermediate nodes between the two boundary paths.

We provide both theoretical and empirical analyses to validate the effectiveness of the proposed techniques. Experimental results show that, with pruning enabled, COLE$^+$ achieves a storage size reduction of up to $16.7\times$ and $98.1\times$ compared to COLE and MPT, respectively. Furthermore, COLE$^+$ improves throughput by up to $1.3\times$ and $3.7\times$ over COLE and MPT, respectively. These results demonstrate the potential of COLE$^+$ to significantly reduce storage requirements and enhance throughput performance while also supporting chain reorganization in practical blockchain deployments.

The remainder of the paper is organized as follows. Section II provides an overview of the COLE$^+$ designs. Section III presents the RS-tree and details its mechanism for handling chain reorganization. Section IV introduces the proposed version tree, followed by a description of COLE$^+$'s write and read operations in Section V. Section VI reports experimental results. Finally, we discuss related work in Section VII and conclude the paper in Section VIII.

II. COLE$^+$ Overview

This section presents an overview of COLE$^+$ with its novel features: chain reorganization, state pruning, and an improved storage layout. We start by giving a brief introduction to COLE's structure, followed by the key designs in COLE$^+$.

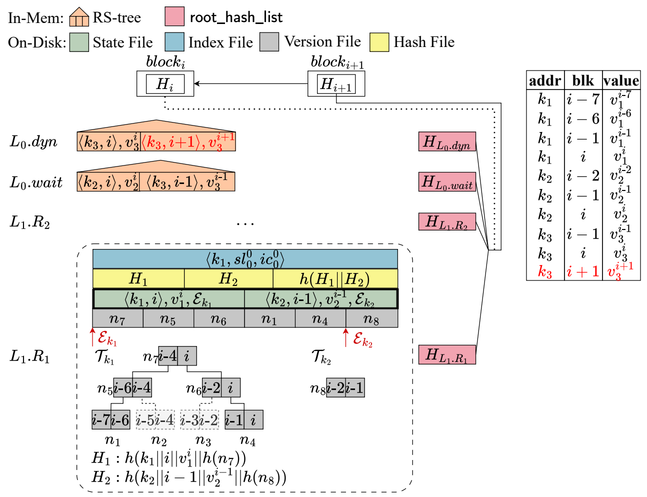
A. Preliminary: COLE

COLE employs a log-structured merge-tree (LSM-tree) to efficiently handle frequent blockchain state updates, taking advantage of its write-optimized design. The structure consists of an in-memory level and multiple on-disk levels with progressively larger capacities. When a state is updated in a new block, COLE inserts the updated state along with its version (i.e., block height) into the in-memory Merkle B-tree (MB-tree) [17] (i.e., L_0 as shown in Figure 2) using a *compound key* $\mathcal{K} = \langle addr, blk \rangle$, where $addr$ is the state address and blk is the block height. For example, in Figure 2, when block $i+1$ updates the state at address k_3, the compound pair $\langle k_3, i+1 \rangle$ together with the new value v_3' is inserted into L_0. When L_0 reaches a predefined maximum capacity, it is flushed to disk as a sorted run in the next level L_1. Merge operations may then occur recursively across subsequent levels as needed, until no level exceeds its capacity threshold. Upon block finalization, COLE computes a state digest to attest to blockchain state integrity. This digest is derived from a root_hash_list, which aggregates the root hashes of the in-memory MB-tree and the Merkle Hash Trees (MHTs) [18] associated with each on-disk run. The root_hash_list is updated once the flush and merge operations are committed.

Each on-disk level consists of multiple sorted runs, each comprising three files:

- **Value file** contains the sorted blockchain states in the form of compound key-value pairs. For instance, the first run R_1 in L_1 includes entries such as $\langle \langle k_1, i-1 \rangle, v_1 \rangle$, $\langle \langle k_1, i \rangle, v_1' \rangle$, and $\langle \langle k_2, i \rangle, v_2 \rangle$ shown in Figure 2.
- **Index file** holds a series of ϵ-bounded piecewise linear models, which help efficiently locate blockchain states within the value file. Given a model, a compound key's position in the value file is predicted as p_{pred} that satisfies $|p_{pred} - p_{real}| \le \epsilon$, where p_{real} is the key's real location. By setting ϵ to half the page capacity, at most two file pages will be accessed per model during data retrieval (i.e., the page of p_{pred} and either its preceding or succeeding page), thereby enhancing I/O efficiency.
- **Merkle file** stores a complete MHT constructed from the states in the value file for the purpose of data authentication. With the complete MHT, it is able to generate a proof to verify a given piece of data based on its position in the value file, thus facilitating efficient data provenance.

For data retrieval in COLE, the process follows a level-



addr	blk	value
k_1	$i-7$	$v_1^{i\text{-}7}$
k_1	$i-6$	$v_1^{i\text{-}6}$
k_1	$i-1$	$v_1^{i\text{-}1}$
k_1	i	v_1^i
k_2	$i-2$	$v_2^{i\text{-}2}$
k_2	$i-1$	$v_2^{i\text{-}1}$
k_2	i	v_2^i
k_3	$i-1$	$v_3^{i\text{-}1}$
k_3	i	v_3^i
k_3	$i+1$	v_3^{i+1}

Fig. 3. Structure of COLE$^+$

wise search approach. To locate the state value corresponding to address $addr_q$ with version blk_q, a search key $\mathcal{K}_q \leftarrow \langle addr_q, blk_q \rangle$ is formed. The search traverses the in-memory MB-tree and the learned models in the disk levels, and stops upon finding a key $\mathcal{K}_r \leftarrow \langle addr_r, blk_r \rangle$ such that $addr_r = addr_q$ and $blk_r \le blk_q$, at which point the corresponding value is returned. Retrieving the latest state value is similar but utilizing a special search key $\langle addr_q, max\_int \rangle$, where max_int is the maximum block height.

B. Key Designs in COLE$^+$

COLE$^+$ advances beyond COLE's limitations with several novel designs: rewind-supported trees for in-memory blockchain states, prunable version files for on-disk historical states, and an enhanced storage layer. The key designs of COLE$^+$ are detailed below.

Rewind-Supported Tree: To support efficient chain reorganization occurring commonly for recent blocks as mentioned in Section I, COLE$^+$ introduces a novel in-memory index structure, the rewind-supported tree (RS-tree). Inspired by content-defined chunking (CDC) [15], [16], the RS-tree determines node splitting points based solely on local data patterns, independent of the update sequence. This design ensures a deterministic tree structure, regardless of the removal of stale states and/or the appending of new states during chain reorganization. The in-memory layer of COLE$^+$ consists of two groups of trees: the *dynamic* group for incoming data and the *waiting* group for data ready for the flush operation. This prevents the complete flushing of in-memory data, ensuring that enough states remain available for rewinding. During index rewind, if all rewind states belong to the dynamic group, they can be directly removed from the dynamic RS-tree. If the rewind states span both groups, the current dynamic group is simply discarded, and the current waiting group is reverted back to the old dynamic group, with corresponding data being removed from its RS-tree. The old waiting group is then restored from the flushed disk run, followed by the reverting of any possible merge operations on disk. To avoid the costly disk operations in the second case, COLE$^+$ creates a temporary checkpoint by taking a snapshot of the root_hash_list before each flush operation. Although some states are flushed to disk, computing the index's root digest in the block only requires the updated in-memory RS-tree root hashes and the old snapshot hashes in the previous root_hash_list for on-disk runs.

Prunable Version File: The prunable version file is proposed for storing historical states on disk to support state pruning. It stores the historical values of each state along with their authenticated information, forming a *version tree* for each state. The version tree is a specialized MHT that uses a purposely designed CDC algorithm for node construction. Because of the locality property of CDC, node splitting points are determined by the node's content rather than the update sequence, unlike a standard complete MHT. Therefore, LSM-tree merge operations affect only the splitting points of a limited number of contiguous tree nodes bounded by the CDC algorithm. Consequently, all tree nodes outside the left-most and right-most boundary paths can be safely pruned. Importantly, regardless of pruning, the structure and corresponding root hash of the new version tree remain consistent after LSM-tree merge operations. For example, Figure 3 shows a version tree $\mathcal{T}_{k_1}$ for address k_1, where only the states' version numbers are displayed and the values are omitted. After state pruning, nodes n_2 and n_3 can be safely removed, while the nodes along the boundary paths (i.e., n_7, n_5, n_6, n_1, n_4) are retained for subsequent LSM-tree merge operations.

Improved Storage Layout: COLE$^+$ further optimizes the storage layers for the on-disk level. Like COLE, it leverages the LSM-tree maintenance strategy for better write efficiency. However, for each disk run, COLE$^+$ utilizes four improved files: a *state file*, an *index file*, a *version file*, and a *hash file*. The state file records only the *latest version* of each state in the run, along with pointers to its historical versions stored in the version file. This separation of latest and historical versions improves the efficiency of data retrieval by reducing the search space. As shown in Figure 3, the state file of run $L_1.R_1$ contains the latest version of k_1, denoted as v_1^i, and the latest version of k_2, denoted as v_2^{i-1}, along with their respective pointers, $\mathcal{E}_{k_1}, \mathcal{E}_{k_2}$. The index file contains disk-optimized learned models that serve as an index for searching values in the state file, similar to the approach used in COLE. However, as we observe that the index must read entire pages whenever accessing data from the disk, COLE$^+$ chooses to reduce the precision of model predictions and the training input data size, thereby improving the training efficiency. The version file stores the historical versions of each state using a novel design. The hash file maintains a complete MHT over the latest states from the state file, and is associated with the root hash of the corresponding version tree. To compute the index digest in the block header, the root hashes of the two RS-trees in memory and the hash file on disk are stored in the root_hash_list. In Figure 3, the hash file of run $L_1.R_1$ contains a complete MHT with two leaf hashes, H_1 and H_2, which authenticate the latest values and the version trees of addresses k_1 and k_2, respectively. The root hash is computed by hashing the concatenation of H_1 and H_2.

III. Rewind-Supported Tree

This section introduces the in-memory rewind-supported tree (RS-tree), which utilizes a CDC-based approach for node splitting. We start with a brief overview of the content-defined chunking (CDC) algorithm, then show the construction of an RS-tree, and finally explore the process of chain reorganization during a blockchain fork.

A. CDC Algorithm

A straightforward approach to node construction during tree building is splitting based on fanout. For example, in a B+-tree or a complete MHT, a node splits upon reaching its maximum fanout. However, this method suffers from a key drawback: node splitting points are sensitive to data updates. Inserting a new data entry at the beginning, for instance, shifts the entire existing data sequence, potentially invalidating almost all nodes due to altered splitting points. This non-deterministic index structure, dependent on both the update sequence and the data content, violates the requirement for a consistent root hash, especially during chain reorganization.

To make the index structure independent of the data update sequence, we adopt the content-defined chunking (CDC) method [15], [16] to determine the node splitting points based solely on local data patterns. Owing to this locality property, the index structure is determined entirely by the indexed data itself. The CDC method consists of two stages: (i) computing a *fingerprint* using a sliding window over the data content, and (ii) comparing the fingerprint against a *mask* derived from the expected chunk size to decide whether to create a cut point (i.e., finding a CDC pattern). Several rolling-hash algorithms can be used to implement the fingerprint, such as Gear Hash [19] and Rabin [20], [21].

To adapt the CDC method for COLE$^+$, we make the following essential modifications: (i) introducing a maximum chunk size f_{max}, which effectively limits the maximum fanout of each tree node, (ii) aligning cut points with the data entry size (256 bits for both state values in leaf nodes and hash values in internal nodes), and (iii) independently finding a tree node's cut point by resetting the fingerprint before examining the next one. The first modification prevents excessively large tree nodes. The second and third modifications are designed for the blockchain context. Unlike the original CDC method, which was designed for byte-stream deduplication, our proposed CDC algorithm ensures that tree node cut points are aligned with the 256-bit entry size. Moreover, cut points are determined solely by the data within the current node, rather than by data across multiple nodes, which is vital for state pruning.

Algorithm 1 shows our proposed CDC algorithm for COLE$^+$. The function `InitParams(·)` initializes a parameter object that includes a CDC mask, a current node size counter cnt, and a maximum node size f_{max}. The CDC mask is determined by the expected node size, calculated from f_{exp} multiplied by the data entry size (e.g., state key size + version size + value size). In the function `CutPoint(·)`, when the counter cnt reaches f_{max}, a cut point is returned (Lines 6 to 8).

Algorithm 1: CDC Algorithm in Tree Nodes

```
1  Function InitParams($f_{exp}, f_{max}$)
     Input: Expected fanout $f_{exp}$, maximum fanout $f_{max}$
     Output: Parameter $param_{cdc}$
2    $param_{cdc}.mask \leftarrow generate\_mask(f_{exp})$;
3    $param_{cdc}.cnt \leftarrow 0$; $param_{cdc}.f_{max} \leftarrow f_{max}$;
4    return $param_{cdc}$;
5  Function CutPoint($param_{cdc}, data$)
     Input: Parameter $param_{cdc}$, input data chunked in 256
            bits $data$
     Output: Pattern result
6    if $param_{cdc}.cnt > param_{cdc}.f_{max}$ then
7      $param_{cdc}.cnt \leftarrow 0$;
8      return CUT;
9    $h_{cdc} \leftarrow init\_cdc()$; $w \leftarrow init\_window()$;
10   slide each $|w|$ bytes in $data$
11     $slice \leftarrow$ data slice in window $|w|$;
12     $fp \leftarrow h_{cdc}.fingerprint(slice)$;
13     if $fp$ & $param_{cdc}.mask = 0$ then
14       $param_{cdc}.cnt \leftarrow 0$;
15       return CUT;
16   $param_{cdc}.cnt \leftarrow param_{cdc}.cnt + 1$;
17   return NOCUT;
```

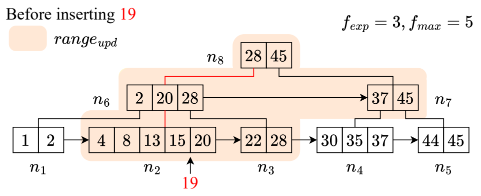


Fig. 4. RS-tree Before Inserting 19

Algorithm 2: RS-tree Maintenance (Insertion)

```
1  Function RSTreeInsert($key, value$)
     Input: Inserted key $key$, value $value$
2    $leaf \leftarrow search\_key(key)$;
3    $range_{upd} \leftarrow [leaf, \mathtt{Succ}(leaf)]$;
4    Bottom-up traverse RS-tree along $key$'s path do
5      if at leaf level then Update $\langle key, value\rangle$ in $range_{upd}$ ;
6      else
7        Replace obsolete entries in $range_{upd}$ with $e_{upd}$;
8      $nodes_{upd} \leftarrow \mathtt{CDCCreateNodes}(range_{upd})$;
9      Replace $range_{upd}$ in RS-tree with $nodes_{upd}$;
10     $e_{upd} \leftarrow \{\langle \mathcal{K}^n_{c_m}, h(n)\rangle \mid \forall n \in nodes_{upd}\}$;
11     $[head, tail] \leftarrow range_{upd}$;
12     $range_{upd} \leftarrow [\mathtt{Par}(head), \mathtt{Succ}(\mathtt{Par}(tail))]$;
13   If the root only has one entry, set its child as the new root;
```

Otherwise, the $\mathtt{CutPoint}(\cdot)$ function generates the CDC fingerprint using a sliding window over the input data to check for the CDC pattern (Lines 10 to 15). If the fingerprint shares the same least-significant bits as the mask (i.e., fp & $mask = 0$), a pattern (or cut point) is identified. Note that the CDC rolling hash will be reinitialized to clear any boundary effects before conducting the pattern check (Line 9).

B. Structure and Maintenance of RS-tree

The RS-tree resembles the structure of Merkle B-tree (MB-tree) [17], with each node identified by its hash value. A leaf node contains key-value pairs, $\{\langle \mathcal{K}_i, value_i\rangle\}_{i=1}^{m}$, and its hash is computed as $h(\mathcal{K}_1||value_1||\cdots||\mathcal{K}_m||value_m)$, where $h(\cdot)$ is a cryptographic hash function (e.g., SHA-256) and '$||$' denotes concatenation. A non-leaf node contains the search keys for locating child nodes and the hashes of those children, $\{\langle \mathcal{K}_{c_i}, h_{c_i}\rangle\}_{i=1}^{m}$. The hash of a non-leaf node is computed as $h(\mathcal{K}_{c_1}||h_{c_1}||\cdots||\mathcal{K}_{c_m}||h_{c_m})$. When a child node is updated, its hash changes, subsequently altering the parent node's hash, propagating up to the root. The root node's hash can attest to all indexed data in the leaf nodes. However, unlike the MB-tree, which splits nodes upon reaching the maximum fanout, the RS-tree splits nodes based on a CDC fingerprint matching a specific pattern or reaching the maximum fanout.

Example. *Figure 4 shows an example of an* RS-tree. *The maximum fanout is $f_{max} = 5$. For simplicity, keys are used to represent node entries. Cut points are created for different reasons: nodes n_1, n_3, n_4, and n_6 are cut due to a matching CDC pattern on their last entries, while node n_2 reaches the maximum fanout. The remaining nodes are on the right boundary path. The hash of leaf node n_2 is $h(4||8||13||15||20)$. The hash of internal node n_6 is $h(2||h_{n_1}||20||h_{n_2}||28||h_{n_3})$.*

The RS-tree insertion algorithm operates similarly to a traditional B+-tree, involving two traversals: a top-down traversal to locate the target leaf node and a bottom-up traversal to update nodes along the key's path. However, unlike traditional B+-trees, which update at most one adjacent node per level, RS-tree may update multiple consecutive nodes at a given level. These additional updates stem from the CDC method, which can introduce multiple new cut points during updates. To track these updates, we use a variable $range_{upd}$. When an entry is inserted into a node already at maximum fanout (f_{max}) or modifies a node's last entry (its cut point), successive nodes are also affected and added to $range_{upd}$. At the leaf level, this process continues until a node with fewer than f_{max} entries is encountered or the rightmost leaf is reached. Similarly, at non-leaf levels, $range_{upd}$ includes all updated child entries and extends to all necessary successive nodes.

Algorithm 2 outlines the RS-tree insertion procedure. First, it locates the target leaf node via a top-down traversal (Line 2). At the leaf level, the affected nodes $range_{upd}$ include the target leaf node and potentially its successive nodes (Line 3). Next, a bottom-up traversal is performed on the RS-tree. At each level, data entries are first updated: the $\langle key, value\rangle$ pair is inserted into the corresponding leaf node (Line 5), while non-leaf nodes update their corresponding entries (Line 7). Then, all nodes in $range_{upd}$ are processed using the CDC method, generating a set of new nodes, $nodes_{upd}$, to replace the old ones (Lines 8 to 9). To maintain the search index structure, the corresponding search keys and hashes of these new nodes are collected into an entry list, e_{upd}, which will be used to update the corresponding parent nodes (Line 10). The $range_{upd}$ for the parent level is then determined by the parent of the first node in $range_{upd}$ and the successive nodes of the parent of the last node in $range_{upd}$ (Lines 11 to 12). This bottom-up traversal continues until the root is reached. If the root contains only one entry (i.e., a single child), this child becomes the new root (Line 13).

Example. *Figures 4 and 5 illustrate the insertion of a state with key 19, assuming a maximum fanout (f_{max}) of*

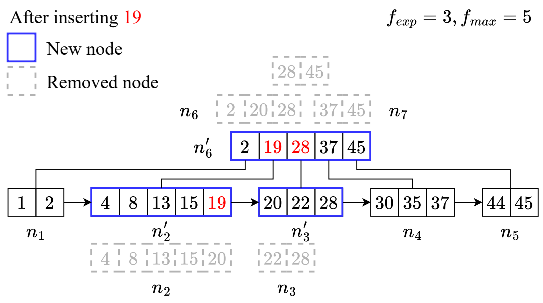


Fig. 5. RS-tree After Inserting 19

5. First, the corresponding leaf node n_2 is located for the key 19. The $range_{upd}$ at the leaf level includes both n_2 and its successive node n_3, which is included because n_2 has reached the maximum fanout. After inserting 19 into n_2, the CDC algorithm creates new nodes n_2' and n_3', replacing their old counterparts n_2 and n_3. Next, their updated entries for the parent nodes, $\{\langle 19, h_{n_2'}\rangle, \langle 28, h_{n_3'}\rangle\}$, are added to the entry list, e_{upd}. The new $range_{upd}$ for the next level is computed as $[n_6, n_7]$ accordingly. We then process the next level. $\{\langle 20, h(n_2)\rangle, \langle 28, h(n_3)\rangle\}$ in node n_6 is replaced with $\{\langle 19, h_{n_2'}\rangle, \langle 28, h_{n_3'}\rangle\}$. After that, new cut points are generated by the CDC algorithm. Here, the original n_6 and n_7 are merged into a single node n_6' due to pattern changes. The new entries e_{upd} at this level are updated as $\{\langle 45, h(n_6')\rangle\}$, and the next $range_{upd}$ becomes $[n_8, n_8]$ as n_8 has no successive node. However, since n_6' is the only node at the current level, it becomes the new root of the RS-tree*, and n_8 is discarded. In Figure 5, blue rectangles and dashed gray rectangles denote new nodes and removed nodes, respectively.*

The deletion operation is similar to the insertion, with a key difference in how $range_{upd}$ is determined. If deleting an entry results in the removal of the last entry in a node, or if the node is at f_{max} before deletion, successive nodes must be included in $range_{upd}$ to handle potential cut-point shifts.

C. Chain Reorganization in COLE$^+$

As mentioned in Section I, chain reorganizations can occur either due to the eventual consistency of consensus protocols (e.g., Proof of Work in Bitcoin [1] and Proof of Stake in Ethereum [2]) or as a result of software upgrades (e.g., post-Ethereum's DAO attack [9]). In consensus-related cases, temporary network partitions may lead to multiple concurrent chain branches before the network converges. Ultimately, only one branch becomes the canonical chain (e.g., Bitcoin's longest chain, which represents the majority of the network's computation power). During a chain reorganization, a node on a non-canonical chain first rewinds to the latest common ancestor block shared with the canonical chain. It then appends and validates the new states from the canonical chain's remaining blocks by computing their canonical index digests. For less frequent software upgrades, the process is similar but may require rewinding arbitrarily more blocks.

We first focus on frequent consensus-related chain reorganizations, which involve only recent blocks. To support efficient state rewinding, we limit it to in-memory levels by maintaining two groups of RS-trees: a dynamic group and a waiting group. Both groups ensure the index's root hash is determined solely by their content. New writes to COLE$^+$ are first inserted into the dynamic group. When the group reaches capacity, it is promoted to the waiting group, while the previous waiting group is flushed to the disk-level LSM-tree. Simultaneously, a new empty RS-tree initializes as the dynamic group. This design guarantees that state data is written to disk only after undergoing two flushes, which prevents premature flushing of all in-memory states and preserves sufficient in-memory states for future rewinds.

If all non-canonical states requiring rewinding are contained within COLE$^+$'s dynamic group, they can be directly removed from the RS-tree. Subsequently, new states from the canonical chain can be appended through normal write operations. Otherwise, if the rewind common ancestor block precedes the most recent flush operation, the current dynamic group is discarded and the current waiting group is reinstated as the dynamic group, with non-canonical states removed from its RS-tree. Note that the last flush operation also modifies the disk levels. Although the on-disk runs could be rebuilt, doing so is computationally expensive and would undermine the requirement for rapid chain reorganizations in consensus protocols. Therefore, a new design is needed to enable appending states from the canonical chain without reverting on-disk changes previously caused by level merges. COLE$^+$ creates a temporary checkpoint by snapshotting the root_hash_list before each flush operation. Although the current waiting group and all disk levels are out-of-sync with the nodes that do not undergo chain reorganization, computing the index digest (consequently validating the new block) only requires the current up-to-date dynamic group RS-tree's root hash and the previous hashes saved in the snapshotted root_hash_list for the out-of-sync waiting group and on-disk levels. Due to the inconsistency between the root_hash_list and the actual data on disk, index operations (e.g., read and write) are temporarily blocked. Once new states are appended to reach the point of the original flush operation, the waiting group and on-disk levels will catch up with the nodes without chain reorganization. After that, normal write operations are resumed.

Example. *Figure 6 shows how COLE$^+$ handles a chain reorganization. Assume each block updates four states, the in-memory capacity is 40, and s_i denotes a state. At $block_{16}$, a flush occurs. The* root_hash_list *($\{H_4, H_2, H_3, \dots\}$) is snapshotted. Then, the old dynamic group $L_0.d$ containing s_{41} to s_{60} becomes the new waiting group, while s_{21} to s_{40} in $L_0.w$ are flushed to $L_1.R_2$. Now consider a blockchain node at $block_{17}$ switching to $block_{19}'$. The node should first rewind states back to $block_{14}$ by removing states s_{57} to s_{68}, then append the new states s_{57}' to s_{76}' from $block_{15}'$ to $block_{19}'$. Since the removed states span both groups, s_{61} to s_{68} in the dynamic group $L_0.d$ are discarded entirely. The old waiting group $L_0.w$ is reverted to the dynamic group, followed by removing s_{57} to s_{60} from its* RS-tree*. At this point, the current dynamic group $L_0.d$ matches the original states of $block_{14}$ and yields the same hash value H_1. Although the waiting group and disk*

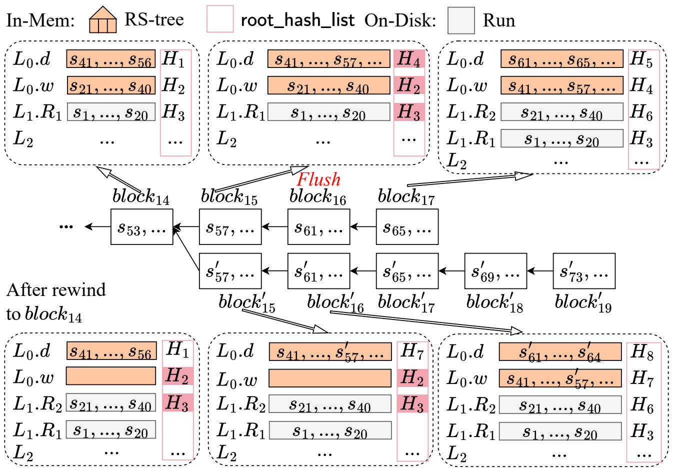


Fig. 6. Example of Chain Reorganization

runs now differ from those in $block_{14}$*, their old hash values are still available from the snapshotted* root_hash_list *(shaded in pink). Using these hash values* $\{H_2, H_3, \dots\}$ *and the up-to-date dynamic group* $L_0.d$*, the blockchain node computes the index digest for* $block'_{15}$*, even though the hashes of runs* $L_1.R_2$ *and* $L_1.R_1$ *temporarily differ from* H_2 *and* H_3*. Once both in-memory groups become full at* $block'_{16}$*, the entries in* root_hash_list *realign with both in-memory and on-disk levels. Since* $L_1.R_2$ *was flushed early, prior to* $block'_{15}$*, there is no need for an additional flush when both in-memory groups eventually fill up. The states* s'_{61} *to* s'_{76} *are then added through normal writes.*

Rare chain reorganizations, typically caused by software upgrades, necessitate extensive on-disk state rewinds. While recursive merges make rebuilding LSM-tree levels unavoidable, the checkpoint's root_hash_list minimizes the rebuilding of runs. The rewind point is defined as the most recent checkpoint immediately preceding the common ancestor block. Compared with its root_hash_list, matching runs are retained, missing runs are rebuilt from the current index states (their sorted nature facilitating consistency), and extra runs are discarded. Normal writes then resume to catch up with the canonical chain. The detailed algorithm is provided in Appendix C.

Correctness Analysis: The correctness of reorganization stems from the consistent root hashes maintained at each level. For frequent chain reorganizations, the in-memory RS-tree, leveraging CDC, ensures deterministic node cut points and index structure, thereby yielding a consistent root hash. For rare reorganizations, any changed on-disk runs are rebuilt. The sorted nature of these runs ensures deterministic index construction, thereby maintaining overall consistency. The detailed proof is provided in Appendix E.

IV. Prunable Version Tree

This section describes the version tree, which is used to store the historical versions of a state within disk runs. We first detail the design of the version tree, including its state pruning capabilities. Then, we explain how the version trees of a given state, located in a level's multiple disk runs, are merged during an LSM-tree merge operation.

A. Version Tree Structure

State pruning is a common technique used by blockchain full nodes to minimize storage overhead. Since historical versions of the state are rarely queried, blockchain full nodes can choose to retain only a few recent versions for each state. To support state pruning, in COLE$^+$, all historical versions of each state are stored as a version tree within each on-disk run. The structure of a version tree is similar to RS-tree, where the CDC method is used to cut nodes at each level. Each leaf node contains version-value pairs $\{\langle blk_i, value_i\rangle\}_{i=1}^{m}$ with $h(blk_1||vaule_1||\cdots||blk_m||value_m)$ as the node's hash. On the other hand, the non-leaf nodes contain the version number search key and corresponding hash for the child nodes, $\{\langle blk_{c_i}, h_{c_i}\rangle\}_{i=1}^{m}$ with the node's hash as $h(blk_{c_1}||h_{c_1}||\cdots||blk_{c_m}||h_{c_m})$. Similarly, the hash of the root node is used to authenticate all version states stored in the leaf nodes. However, unlike RS-tree always being a full tree, the version tree can be either a full tree or a pruned tree.

For pruned version trees, a key challenge arises: how to delete most tree nodes while still enabling the computation of the new merged version tree during subsequent LSM-tree merge operations. Thanks to the locality property of the CDC method, the node cut points at each level are fully determined by the content of the tree nodes. This ensures that both full archive and pruned blockchain nodes can agree on the same version tree structure and, consequently, the same digest for the entire COLE$^+$ index. However, sufficient information needs to be retained such that pruned blockchain nodes can still compute updated nodes during the LSM-tree merge operation. We find that preserving the boundary paths (leftmost and rightmost) during state pruning could satisfy this requirement. Since the merged version tree follows a natural chronological order, the version spaces of the two trees are guaranteed not to overlap with each other. For example, within a level containing runs $R_i, R_{i-1}, \dots, R_1$ (ordered from newest to oldest), the versions in R_i are guaranteed to be greater than those in R_j, where $i > j$. Due to this monotonic property, the merging process only requires the rightmost path of the left merged tree and the leftmost path of the right merged tree to determine node split points. All other nodes between these boundary paths can be safely discarded after pruning.

Note that while only one node per level needs to be kept along the rightmost path of the left tree, retaining only the leftmost node at each level along the leftmost path of the right tree may be insufficient. If the leftmost node of the right tree has f_{max} entries rather than following a CDC pattern, merging it with the rightmost node of the left tree might create a new node containing extra entries that do not satisfy a CDC pattern while having fewer entries than f_{max}. Therefore, at each level of the version tree, if the leftmost node has f_{max} entries, all its successive nodes are retained until one with fewer than f_{max} entries is encountered. All ancestor nodes of these boundary path nodes are likewise retained to maintain tree connectivity.

Example. *Figure 7 shows two pruned version trees,* $\mathcal{T}_l$ *and* $\mathcal{T}_r$*, of one state, assuming the maximum fanout* $f_{max} = 3$*.*

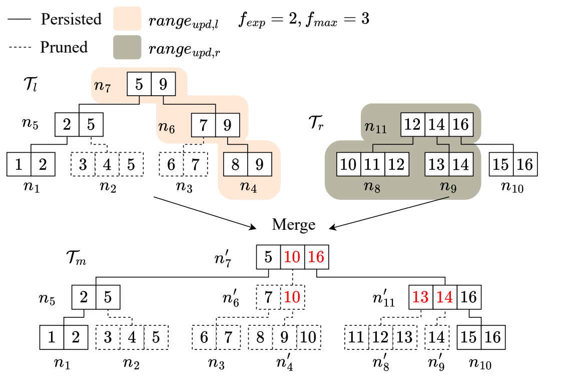


Fig. 7. Example of Version Trees

Algorithm 3: Merge Version Trees

```
1  Function MergeVersionTree($\mathcal{T}_l, \mathcal{T}_r$)
     Input: Left tree $\mathcal{T}_l$, right tree $\mathcal{T}_r$
     Output: Merged tree $\mathcal{T}_m$
2    $node_l \leftarrow \mathcal{T}_l$'s rightmost leaf node;
3    $node_r \leftarrow \mathcal{T}_r$'s leftmost leaf node;
4    $range_{upd,l} \leftarrow [node_l, node_l]$;
5    $range_{upd,r} \leftarrow [node_r, \texttt{Succ}(node_r)]$;
6    while $range_{upd,l} \neq \emptyset \vee range_{upd,r} \neq \emptyset$ do
7      Copy all nodes in the current level from $\mathcal{T}_l$, $\mathcal{T}_r$ to $\mathcal{T}_m$;
8      $range_{upd,m} \leftarrow range_{upd,l} \cup range_{upd,r}$;
9      if not at leaf level then
10       Replace obsolete entries in $range_{upd,m}$ with $e_{upd,m}$;
11     $nodes_{upd} \leftarrow \texttt{CDCCreateNodes}(range_{upd,m})$;
12     Replace $range_{upd,m}$ in $\mathcal{T}_m$ with $nodes_{upd}$;
13     $e_{upd} \leftarrow \{\langle blk^n_{c_m}, h(n)\rangle \mid \forall n \in nodes_{upd}\}$;
14     $[head_l, tail_l] \leftarrow range_{upd,l}$;
15     $[head_r, tail_r] \leftarrow range_{upd,r}$;
16     if $head_l$ is root node then $range_{upd,l} \leftarrow \emptyset$ ;
17     else $range_{upd,l} \leftarrow [\texttt{Par}(head_l), \texttt{Par}(tail_l)]$ ;
18     if $head_r$ is root node then $range_{upd,r} \leftarrow \emptyset$ ;
19     else
20       $range_{upd,r} \leftarrow [\texttt{Par}(head_r), \texttt{Succ}(\texttt{Par}(tail_r))]$;
21   If $\mathcal{T}_m$'s root only has one entry, set child as the new root;
22   return $\mathcal{T}_m$;
```

For brevity, only the version numbers of the states are shown. For $\mathcal{T}_l$, the left-most and right-most nodes at each level are retained, while $\mathcal{T}_r$ retains all the nodes. Node n_9 cannot be pruned because its predecessor, n_8, is at f_{max} capacity. There may be a situation, where after merging, versions 11 and 12 in n_8 cannot form a valid tree node on their own. Instead, a new node, n'_8, might need to be created using these extra entries along with version 13 from n_9. As such, node n_9 must be retained.

B. Merging Version Trees

Algorithm 3 details the procedure of merging two version trees, $\mathcal{T}_l$ and $\mathcal{T}_r$. It can be easily extended to support merging multiple trees by iteratively merging the resulting tree with the next one. Similar to Algorithm 2, the procedure works by bottom-up traversing both $\mathcal{T}_l$ and $\mathcal{T}_r$ along the merging boundary paths. During the traversal, we maintain two ranges, $range_{upd,l}$ and $range_{upd,r}$, to track the new cut points from the CDC method for both left and right trees. Due to the aforementioned reason, at each tree level, $range_{upd,l}$ only contains the rightmost node of the left tree (Lines 4 and 17), while $range_{upd,r}$ contains the leftmost node along with all necessary successive tree nodes for the right tree (Lines 5 and 20). For each iteration of the traversal, we first copy all nodes at the current level from both trees to the merged tree (Line 7). The update range of the merged tree $range_{upd,m}$ is computed by combining $range_{upd,l}$ and $range_{upd,r}$ (Line 8). The updated entries e_{upd} from the previous iteration are applied to this updated range, followed by creating new tree nodes using the CDC method (Lines 10 to 11). Next, the updated entries e_{upd} are computed for the new tree nodes (Line 13). Finally, we update ranges $range_{upd,l}$ and $range_{upd,r}$ for the next iteration (Lines 16 to 20). The tree traversal ends until reaching the tree roots for both $\mathcal{T}_l$ and $\mathcal{T}_r$. If the merged tree's root contains only one entry, this child becomes the new root (Line 21).

Example. *Following the example in Figure 7, the merge operation is executed in a bottom-up fashion. Starting at the leaf level, $range_{upd,l}$ and $range_{upd,r}$ are computed as $[n_4, n_4]$ and $[n_8, n_9]$, respectively. Applying the CDC method, new nodes n'_4, n'_8, and n'_9 are created. These new nodes entail entry updates e_{upd}, consisting of $\{\langle 10, h(n'_4)\rangle, \langle 13, h(n'_8)\rangle, \langle 14, h(n'_9)\rangle\}$. At the next level, $range_{upd,l}$ and $range_{upd,r}$ become $[n_6, n_6]$ and $[n_{11}, n_{11}]$, respectively. Consequently, new nodes n'_6 and n'_{11} are generated with update entries $e_{upd} = \{\langle 10, h(n'_6)\rangle, \langle 16, h(n'_{11})\rangle\}$. Finally, at the root level, $range_{upd,l}$ and $range_{upd,r}$ become $[n_7, n_7]$ and $\emptyset$, respectively. A root node n'_7 is created for the merged tree. The updated entries are highlighted in red. The merged tree can be further pruned, retaining only nodes, n'_7, n_5, n'_{11}, n_1, and n'_{10}.*

Correctness Analysis: The correctness of version trees stems from the consistent root hash achieved between full archive nodes and pruned nodes after merge operations. The version tree's structure is determined by the content-locality property of CDC. Full nodes trivially compute the correct root hash. For pruned nodes, any merge failure would imply the presence of essential updates outside the retained boundary paths, which directly contradicts CDC's locality property. Thus, retaining the boundary paths ensures consistent root hashes across pruned nodes. The complete proof, as well as the analysis of the storage reduction, are given in Appendix F and Appendix G.

V. Write and Read Operations in COLE$^+$

This section details the write and read operations in COLE$^+$. Algorithm 4 outlines the write operation in COLE$^+$. The updated state's address $addr$ and the current block height blk form the compound key $\mathcal{K}$. First, the compound key paired with the state value $value$ is inserted into the RS-tree of the dynamic group at level $L_0.dyn$ (Lines 2 to 3). When the dynamic group reaches half of the total in-memory capacity $\frac{B}{2}$, a flush operation begins. The current root_hash_list is stored as a snapshot to facilitate potential state rewinds (Line 5). Then, the current waiting group $L_0.wait$ is flushed into an on-disk sorted run at L_1, creating four files: a state file $\mathcal{F}_S$, an index file $\mathcal{F}_I$, a version file $\mathcal{F}_V$, and a hash file $\mathcal{F}_H$ (Lines 7 to 8). Subsequently, the current dynamic group is promoted to the new waiting group, and a new empty waiting group

Algorithm 4: 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 RS-tree in $L_0.dyn$;
4      if $L_0.dyn$ contains $\frac{B}{2}$ key-value pairs then
5          Store the temporary root_hash_list;
6          if $L_0.wait$ is not empty then
7              Flush the leaf nodes in $L_0.wait$ to $L_1$ as a sorted run;
8              Generate files $\mathcal{F}_S, \mathcal{F}_I, \mathcal{F}_V, \mathcal{F}_H$ for this run;
9              $L_0.wait.clear()$;
10         Switch $L_0.dyn$ and $L_0.wait$;
11         $i \leftarrow 1$;
12     while $L_i$ contains $T$ runs do
13         Sort-merge all the runs in $L_i$ to $L_{i+1}$ as a new run;
14         Generate files $\mathcal{F}_S, \mathcal{F}_I, \mathcal{F}_V, \mathcal{F}_H$ for the new run;
15         Remove all the runs in $L_i$;
16         $i \leftarrow i + 1$;
17     Update $H_{index}$ when finalizing the current block;
```

is created for future write operations. Besides the in-memory flush operation, if any on-disk level L_i reaches its capacity (T runs), all runs in L_i are merged into a new sorted run at level L_{i+1} (Lines 12 to 16). During the process, multiple version trees for the same state address across different runs in L_i are merged using the MergeVersionTree($\cdot$) function. Finally, the index digest H_{index} for the new block is computed as h(root_hash_list), which captures the root hashes of L_0's two RS-trees and all on-disk runs (Line 17).

Next, we detail the construction of the four files for each on-disk run. As mentioned in Section II, the latest version of each state is stored in the state file, while historical versions are kept in the version file. This separation reduces the search space when querying the latest version, improving efficiency. The state file contains tuples of the form $\langle addr, blk, value, \mathcal{E}_{addr} \rangle$ for each state in the current run. The first three elements represent the compound key-value pair, while $\mathcal{E}_{addr}$ is a pointer to the corresponding version tree in the version file. Similar to COLE, a Bloom filter is built upon each $addr$ in a state file to expedite state searches. The version file stores the version tree for each state by flattening its nodes in a breadth-first search order. To authenticate both the state file and the version file, a hash file is created containing a complete Merkle Hash Tree (MHT). Each MHT leaf node is computed as $h(addr||blk||value||H_{\mathcal{T}_{addr}})$, where $value$ is the latest version in the state file and $H_{\mathcal{T}_{addr}}$ is the root hash of the corresponding version tree. The root hash of the hash file serves as the root hash of the current on-disk run and is stored in the root_hash_list for computing the final index digest.

Similar to COLE, the index file employs a hierarchy of piecewise linear models for efficiently indexing the state file. However, COLE$^+$ introduces several modifications: (i) the learned models use only the state address ($addr$) instead of the full compound key ($\mathcal{K}$) as input; (ii) model predictions return a page ID granularity rather than an exact offset in the state file; (iii) models are trained only on the first state in each page of the state file instead of all states; and (iv) the training error bound (ϵ) for the piecewise linear models is set to 1. The first

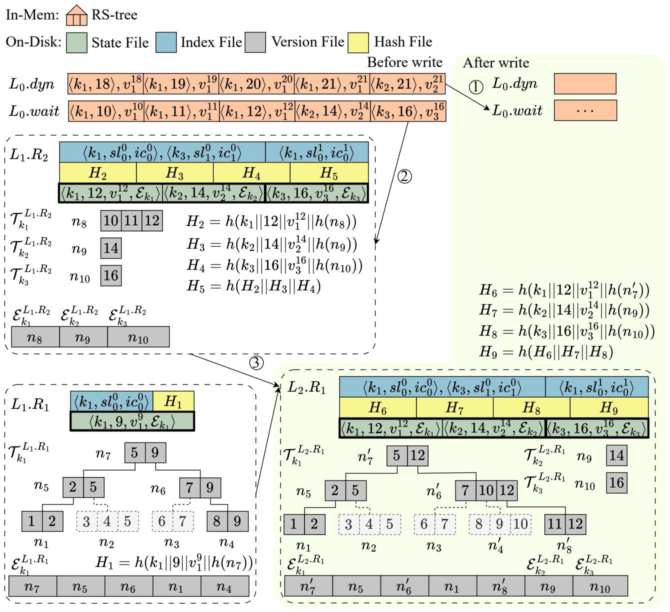

In-Mem: RS-tree
On-Disk: State File, Index File, Version File, Hash File
Before write
After write
$L_0.dyn$
$L_0.wait$
①
②
③
$L_1.R_2$
$L_1.R_1$
$L_2.R_1$
$H_2 = h(k_1||12||v_1^{12}||h(n_8))$
$H_3 = h(k_2||14||v_2^{14}||h(n_9))$
$H_4 = h(k_3||16||v_3^{16}||h(n_{10}))$
$H_5 = h(H_2||H_3||H_4)$
$H_6 = h(k_1||12||v_1^{12}||h(n_7'))$
$H_7 = h(k_2||14||v_2^{14}||h(n_9))$
$H_8 = h(k_3||16||v_3^{16}||h(n_{10}))$
$H_9 = h(H_6||H_7||H_8)$
$H_1 = h(k_1||9||v_1^9||h(n_7))$

Fig. 8. An Example of Write Operation

modification stems from COLE$^+$ relying on a separate version file for historical versions. The remaining changes are based on the observation that the index must read entire pages whenever accessing data from the disk, eliminating the need to predict high-precision offsets. These improvements decrease the size of training inputs and their precision requirements, leading to increased training efficiency.

Example. *Figure 8 illustrates the write operation in COLE$^+$. When $L_0.dyn$ reaches its capacity, ① it is promoted to $L_0.wait$. ② The previous $L_0.wait$ is then flushed to the disk as $L_1.R_2$. Note that the state file in $L_1.R_2$ only contains the latest entries for compound keys $\langle k_1, 12 \rangle$, $\langle k_2, 14 \rangle$, and $\langle k_3, 16 \rangle$. Since L_1 now contains two runs, ③ they are merged into a new run at the next level $L_2.R_1$. As before, the state file at $L_2.R_1$ retains only the latest entries. The corresponding version file is constructed by merging $\mathcal{T}_{k_1}^{L_1.R_1}$ and $\mathcal{T}_{k_1}^{L_1.R_2}$ (via Algorithm 3) while copying $\mathcal{T}_{k_2}^{L_1.R_2}$ and $\mathcal{T}_{k_3}^{L_1.R_2}$. These version trees are stored on disk consecutively, with tree nodes arranged in breadth-first search order. The hash file's leaf nodes are computed as H_6, H_7, and H_8, with the root hash being computed as $H_9 = h(H_6||H_7||H_8)$. This root hash serves as the digest for run $L_2.R_1$ by being added into root_hash_list. Finally, the index file is created by training piecewise learned models using inputs $\{\langle k_1, 0 \rangle, \langle k_3, 1 \rangle\}$, assuming a maximum of two entries per page in the state file.*

There are two types of read operations in COLE$^+$, get queries and provenance queries. Get queries retrieve only the latest value for a given state $addr_q$. Like COLE, a get query in COLE$^+$ first searches the in-memory indexes, then on-disk runs in order of recency. However, the search keys differ between these levels. In the two RS-trees, a special key, $\mathcal{K}_q \leftarrow \langle addr_q, max\_int \rangle$ is used to find the entry with the largest key $\mathcal{K}_r < \mathcal{K}_q$. If $\mathcal{K}_r.addr = addr_q$, its value is returned directly. Otherwise, the search proceeds to on-disk runs. In contrast to COLE, the search key is simply $addr_q$, because the Bloom filters and the learned models are built directly on the state addresses. If the Bloom filter indicates the non-existence of $addr_q$, the state file is skipped. Otherwise, the learned model in the index file predicts the location within the state file using a page ID. If the query entry is not in the corresponding page, either its preceding or succeeding page is retrieved and checked, limiting I/O to at most two pages per run. The search stops once the latest matching entry for $addr_q$ is found in the most recent run.

The provenance query returns historical versions of the queried state add_q within the block height range $[blk_l, blk_u]$, along with a Merkle proof rooted at the latest block to enable integrity verification. First, the in-memory RS-trees are searched using $[\langle addr_q, blk_l - 1\rangle, \langle addr_q, blk_u + 1\rangle]$, where the offsets ensure no versions are omitted from the results. During the search, the traversal path in the RS-tree is recorded as part of the Merkle proof. For on-disk runs, COLE$^+$ differs from COLE due to the separation of the latest and historical versions. First, the index file is queried with $addr$ to locate the corresponding version file offset $\mathcal{E}_{addr}$. If found, the version tree is searched using $[blk_l - 1, blk_u + 1]$, and the tree traversal path is added to the Merkle proof. Regardless of whether the query address appears in the current run, the corresponding Merkle path in the hash file is always included in the Merkle proof. The search stops once versions both earlier than blk_l and later than blk_u are encountered for $addr_q$. On the client side, verification involves recomputing the COLE$^+$ index root hash by reconstructing the Merkle tree using paths from the Merkle proof. The computed hash is then compared against the value stored in the block header.

Example. *Following the example in Figure 8, where read operations occur after the write operation completes. Consider a get query of state k_3, L_0 is first searched using the compound key $\mathcal{K}_q \leftarrow \langle k_3, max\_int\rangle$. $L_0.dyn$ returns nothing as it is empty. On the other hand, $L_0.wait$ returns entry for $\langle k_2, 21\rangle$, which does not match k_3. Next, $L_2.R_1$ is searched as L_1 is also empty. Assuming the index file predicts that the address k_3 is located at the first page of the state file in $L_2.R_1$. In this case, the neighboring page (i.e., the second page) will be examined, yielding the final query result v_3^{16}.*

Next, consider a provenance query for state address k_1 within the version range $[10, 18]$ using the non-pruned index. $[\langle k_1, 9\rangle, \langle k_1, 19\rangle]$ is used to search L_0, which returns empty and $\langle k_1, 18, v_1^{18}\rangle$ and $\langle k_1, 19, v_1^{19}\rangle$ from $L_0.dyn$ and $L_0.wait$, respectively. The corresponding Merkle path at the RS-tree *is added to the Merkle proof. For on-disk runs, the index file allows us to locate $\mathcal{T}_{k_1}^{L_2.R_1}$ via pointer $\mathcal{E}_{k_1}$. At the same time, the Merkle path of the hash file, $\{H_7, H_8\}$, is added to the Merkle proof. To search the version tree, a version range $[9, 19]$ is used. The versions corresponding to block heights 9, 10, 11, and 12 are returned as part of the results, whereas $h(n_5)$, $h(n_3)$, $h(8||v_1^8)$ are used for the Merkle proof. The client can reconstruct COLE$^+$'s index root hash to verify the soundness and completeness of the query results $\{v_1^{10}, v_1^{11}, v_1^{12}, v_1^{18}\}$.*

TABLE I
SYSTEM PARAMETERS

Parameters	**Value**
# of generated blocks	$2 \times 10^4, 6 \times 10^4, 2 \times 10^5, \mathbf{6 \times 10^5}$
Size ratio T	$2, 4, 6, 8, \mathbf{10}$
MHT fanout f	$2, \mathbf{4}, 8, 16, 32$

VI. EXPERIMENTAL EVALUATION

In this section, we compare COLE$^+$ with COLE [5] and the Merkle Patricia Trie (MPT) [2], which serves as the index for the Ethereum blockchain. MPT is typically maintained by key-value databases such as RocksDB [6]. Additionally, we evaluate the non-pruned COLE$^+$ (referred to as COLE$^+$-NP). LETUS [4] is excluded from comparison because it is not open-source and demonstrates smaller performance gains over MPT than COLE [5]. In the following subsections, we describe the system implementation and parameter settings, followed by the workloads and evaluation metrics. We then present the detailed experimental results.

A. Implementation and Parameter Settings

COLE$^+$ is implemented in the Rust programming language with 16,000 lines of code [22]. Both COLE$^+$ and COLE leverage asynchronous merge operations with multi-threading, introduced in [5], to mitigate tail latency during LSM-tree merges. Blockchain transactions are executed using the Ethereum Virtual Machine (EVM). Each block processes 100 transactions, resulting in various read and write operations. Like COLE, COLE$^+$ uses simple file-based storage. The Gear Hash [19] is employed to implement the CDC algorithm. To maximize pruning efficiency, COLE$^+$ retains only the leftmost and rightmost boundary paths in each state's version tree, while pruning all intermediate nodes between these two boundary paths by default. Table I lists the parameters, with default values highlighted in bold. The impact of parameters is evaluated in Appendix H. The size ratio, indicating the maximum number of runs in a level, is set to 10 as it yields the highest throughput and comparably low latency. The complete MHT's fanout is set to 4 as it leads to the fastest provenance queries. The experiments are conducted on a machine with an Intel i7-10710U CPU and a 1 TB SSD.

B. Workloads and Evaluation Metrics

To simulate the blockchain workload, we use the KVStore workload from BlockBench [23] to generate blockchain transactions. Each transaction corresponds to a state read/update operation derived from the YCSB benchmark [24]. Initially, $20,000$ transactions with new states are inserted into the storage as base data. Subsequently, various scenarios with different ratios of read/update operations are generated: (i) Write-Only (entirely update operations); (ii) Read-Write (half read and half update operations); and (iii) Read-Only (entirely read operations). Transactions are packed in blocks, which are committed serially using a single thread, as required by the blockchain's consensus protocol. We measure storage size and system throughput to assess overall performance. For provenance queries, a set of state addresses is randomly selected

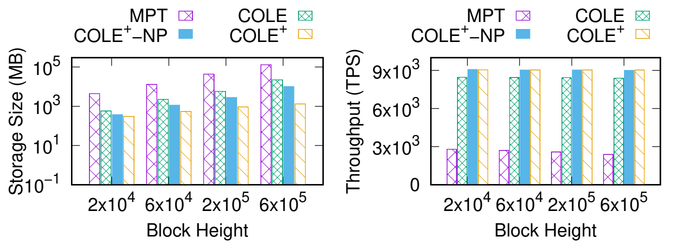


Fig. 9. Performance vs. Block Height

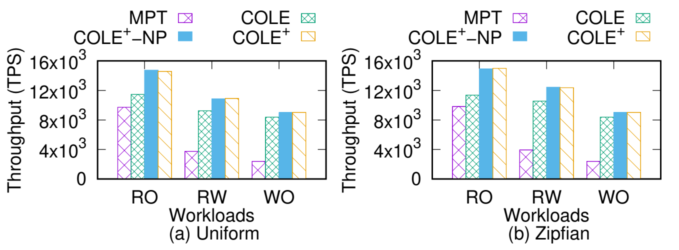


Fig. 10. Throughput vs. Workloads

from the base data, and different state version ranges extending from the latest block (e.g., $2, 4, \ldots, 128$) are generated. We measure the total CPU time for both executing queries on the blockchain node and the query user's verification time, as well as the proof size. We also evaluate block rewind latency for both frequent and rare cases as the number of rewound blocks increases. Additionally, we conduct an ablation study to assess the throughput impact of (i) the state separation layout (separating the latest states from historical ones), (ii) learned indexes for accelerating run searches, (iii) the improved CDC, and (iv) the proposed RS-tree.

C. Experimental Results

1) Overall Performance: Figure 9 compares the storage size and throughput of the evaluated indexes under the Write-Only workload. Compared to COLE, COLE$^+$-NP reduces the storage size by up to $2.2\times$ at a block height of 6×10^5. This improvement stems from redesigning the state and version files. In COLE, each state address in a run is duplicated across historical versions, whereas COLE$^+$-NP stores each address only once in the run and relocates all historical values to the version file, while preserving the column-based design. State pruning in COLE$^+$ further reduces the storage size by $1.2\times$ to $7.7\times$ compared to COLE$^+$-NP, and by $16.7\times$ compared to COLE. This substantial gain arises from the innovative prunable version tree design, which retains only the boundary paths. Although COLE already achieves a $5.8\times$ storage reduction compared to MPT, COLE$^+$ achieves up to $98.1\times$ reduction over MPT.

Regarding system throughput, COLE$^+$-NP and COLE$^+$ achieve comparable throughput. While COLE already boosts throughput by up to $3.5\times$ over MPT thanks to its efficient column-based design, COLE$^+$ delivers an even higher improvement, up to $3.7\times$ over MPT, while also enabling state rewinds and pruning — features that are critical for most practical blockchain applications.

2) Impact of Workloads: We use three workloads, Read-Only (RO), Read-Write (RW), and Write-Only (WO)

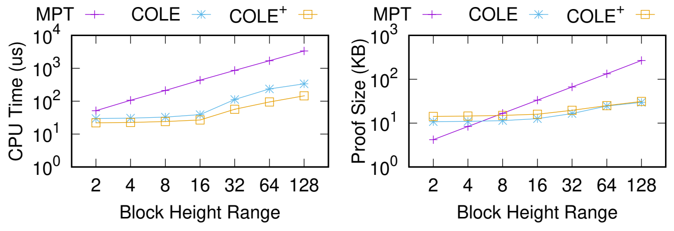


Fig. 11. Prov-Query Performance vs. Query Range

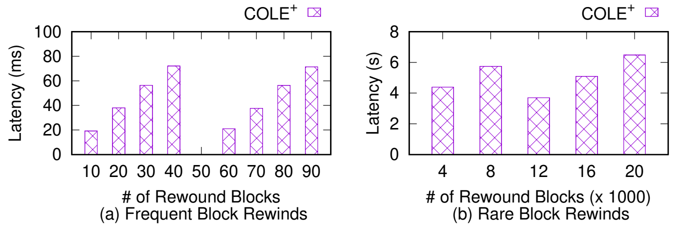


Fig. 12. Frequent and Rare Block Rewind Performance

across both Uniform and Zipfian distributions to assess their impact on system throughput. As shown in Figure 10, all systems experience reduced throughput under increasing write operations. Specifically, the throughput of MPT decreases by up to 75.5%. COLE's throughput decreases by up to 26.9%, while COLE$^+$-NP and COLE$^+$ experience comparable reductions, up to 39% and 38.1%, respectively. The LSM-tree-based maintenance approach used by COLE and COLE$^+$ generally enhances write performance.

Another interesting observation is that COLE$^+$ improves throughput over COLE under both Read-Only and Read-Write workloads. Under the Read-Only workload, COLE$^+$ achieves up to 28.7% and 31.2% higher throughput compared to COLE for the Uniform and Zipfian distributions, respectively. Under the Read-Write workload, COLE$^+$ also surpasses COLE, improving throughput by up to 17.7%. These performance gains are mainly attributed to COLE$^+$'s design that separates the latest and historical versions. Storing only the latest state in a state file reduces the search space for on-disk get queries, whereas COLE must search all historical versions.

3) Provenance Query Performance: To evaluate provenance query performance, COLE$^+$ retains boundary paths that cover sufficient historical data (e.g., the most recent 128 blocks) instead of only the left-most and right-most boundary paths. Figure 11 compares the CPU time and proof size of MPT, COLE, and COLE$^+$ for provenance queries. For MPT, both metrics increase linearly with block height due to the requirement to query each block in the range. In contrast, COLE and COLE$^+$ exhibit sublinear growth. COLE$^+$ achieves superior CPU performance by separating latest state values from historical data, which enables its index models to filter non-queried states more effectively and thereby reduce the search space. The consecutive storage of version tree nodes further optimizes I/O costs. While COLE$^+$ has a slightly larger proof size than COLE, this increase results from the inclusion of version numbers as search keys in the version trees, which are not required in COLE's complete MHT.

4) Block Rewind Performance: Figure 12 shows the latency for frequent and rare block rewinds during chain

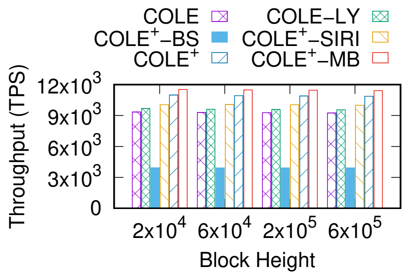


Fig. 13. Ablation Study

reorganizations, across varying numbers of rewound blocks. Block appending results, similar to the Write-Only workload, are omitted. Frequent rewinds complete efficiently in under 100 ms. The drop at 50 blocks corresponds to the complete discarding of the dynamic group and only a few remaining removals from the waiting group. Rare rewinds are several hundred times slower, but still complete within 8 seconds for up to 2×10^4 rewound blocks, which is acceptable in practice for infrequent events such as software upgrades. The drop at 12×10^3 blocks stems from reusing unchanged on-disk runs, thus eliminating the costly run reconstruction process.

5) Ablation Study: We perform an ablation study with the following variants: COLE-LY (adding latest/historical state separation to COLE), COLE$^+$-BS (replacing the learned model prediction with binary search), COLE$^+$-SIRI (excluding the proposed CDC improvement), and COLE$^+$-MB (replacing COLE$^+$'s $\mathrm{RS\text{-}tree}$ with MB-tree [17]). As shown in Figure 13, COLE-LY increases throughput over COLE by 4% under Read-Write workload. With more read operations, the new layout further boosts the gain to 14%, as shown in Appendix I. COLE$^+$ achieves 2.8× throughput compared to COLE$^+$-BS, validating the effectiveness of the learned index. COLE$^+$'s improved CDC yields a 10% increase in throughput compared to COLE$^+$-SIRI. Using $\mathrm{RS\text{-}tree}$ in COLE$^+$ incurs only a 5% performance overhead compared to MB-tree, demonstrating an acceptable trade-off to enable efficient state rewind.

VII. Related Work

In this section, we provide a brief review of related studies on learned indexes and blockchain storage management.

Learned Indexes. Kraska *et al.* introduced the concept of learned index structures by replacing search keys in an index node with a model, significantly reducing the search complexity and the size in terms of a node [25]. Since then, numerous studies have been conducted to apply this approach across various scenarios. For example, [26], [27], [28], [29] focus on making learned indexes updatable. [30] employs optimal piecewise linear models to construct indexes with theoretical worst-case bounds. Studies such as [31], [32], [33] address multidimensional data. [34] explores a hybrid construction to balance performance and memory consumption, while [35] targets memory-efficient sliding-window queries over data streams. Beyond in-memory solutions, on-disk learned indexes, as discussed in [36], [37], [38], [39], account for the unique challenges of disk access and layout. COLE [5] is the first learned storage for blockchain systems, supporting both data integrity and provenance queries. However, it fails to support chain reorganization and state pruning.

Blockchain Storage Management. Extensive studies have been proposed to optimize blockchain storage. Sharding technique is actively researched, where each node maintains only a partition of the blockchain, thereby reducing storage costs and enhancing parallelism [40], [41], [42], [43], [44], [45]. Several studies have opted for off-chain storage solutions to alleviate on-chain costs [46], [47], [48]. ForkBase [10] studied the problem of concurrent updates in a distributed network. It also employs the CDC method to efficiently identify and eliminate duplicate content across data objects in different branches to improve performance. However, the CDC approach used by ForkBase is insufficient to directly support chain reorganization and state pruning. Feng *et al.* proposed SlimArchive, a storage optimization system for Ethereum full archive nodes. It reduces storage costs by flattening minimal blockchain state changes and removing Merkle-based structures [7]. However, this storage cost reduction comes at the cost of sacrificing data authentication and provenance, which are essential security features of blockchain systems. Tian *et al.* proposed LETUS, a log-structured, trusted, universal blockchain storage system [4]. LETUS employs a novel Merkle-based structure called DMM-Tree, which uses delta-encoding to reduce storage costs. However, similar to MPT, DMM-Tree retains obsolete nodes from historical blocks to support provenance queries.

In addition to storage optimization, improving query efficiency in blockchain systems is also a promising research area. Some studies focused on verifiable queries over blockchain databases [49], [50], [51], [52]. Others explore query processing in the context of on-chain and off-chain hybrid storage [53], [54], [55], [56], [57]. Recently, FlexIM [58] has been proposed to manage and select verifiable indexes for dynamic queries in blockchain systems. Unlike these studies, COLE$^+$ and COLE focus on general-purpose blockchain storage.

VIII. Conclusion

This paper has introduced COLE$^+$, an enhanced column-based learned storage system for blockchains that addresses the limitations of existing state-of-the-art solutions like COLE. By proposing a novel rewind-supported in-memory index structure based on CDC, COLE$^+$ enables efficient chain reorganization, a critical feature missing in COLE. Furthermore, COLE$^+$ introduces a new two-level MHT structure, incorporating a prunable version tree, that facilitates efficient state pruning and significantly reduces storage overhead. Empirical evaluations demonstrate that with pruning enabled, COLE$^+$ reduces storage size by up to 16.7× and 98.1× and increases throughput up to 1.3× and 3.7× compared to COLE and MPT, respectively. These improvements, particularly the support for chain reorganization and state pruning, pave the way for wider practical adoption of COLE$^+$ by real-world blockchain systems.

IX. AI-Generated Content Acknowledgment

No AI tools were used to generate the text, figures, images, or pseudo-code in this paper.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized Business Review*, p. 21260, 2008.

[2] G. Wood. (2014) Ethereum: A secure decentralised generalised transaction ledger. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[3] "Ethereum full node sync (archive) chart," https://etherscan.io/chartsync/chainarchive, 2025.

[4] S. Tian, Z. Lu, H. Zhuo, X. Tang, P. Hong, S. Chen, D. Yang, Y. Yan, Z. Jiang, H. Zhang *et al.*, "Letus: A log-structured efficient trusted universal blockchain storage," in *Companion of the 2024 International Conference on Management of Data*, 2024, pp. 161–174.

[5] C. Zhang, C. Xu, H. Hu, and J. Xu, "COLE: A column-based learned storage for blockchain systems," in *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, 2024, pp. 329–345.

[6] S. Dong, A. Kryczka, Y. Jin, and M. Stumm, "Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications," *ACM Trans. Storage*, pp. 1–32, 2021.

[7] H. Feng, Y. Hu, Y. Kou, R. Li, J. Zhu, L. Wu, and Y. Zhou, "{SlimArchive}: A lightweight architecture for ethereum archive nodes," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024, pp. 1257–1272.

[8] (2019) Chain reorganization. [Online]. Available: https://en.bitcoin.it/wiki/Chain_Reorganization

[9] (2025) Ethereum classic and the ethereum hard fork. [Online]. Available: https://help.coinbase.com/en/coinbase/getting-started/crypto-education/eth-hard-fork

[10] S. Wang, T. T. A. Dinh, Q. Lin, Z. Xie, M. Zhang, Q. Cai, G. Chen, B. C. Ooi, and P. Ruan, "Forkbase: an efficient storage engine for blockchain and forkable applications," *PVLDB*, pp. 1137–1150, 2018.

[11] C. Yue, Z. Xie, M. Zhang, G. Chen, B. C. Ooi, S. Wang, and X. Xiao, "Analysis of indexing structures for immutable data," in *ACM SIGMOD*, 2020, pp. 925–935.

[12] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th symposium on operating systems principles*, 2017, pp. 51–68.

[13] G. Wood, "Polkadot: Vision for a heterogeneous multi-chain framework," *White Paper*, pp. 2327–4662, 2016.

[14] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich *et al.*, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the thirteenth EuroSys conference*, 2018, pp. 1–15.

[15] W. Xia, Y. Zhou, H. Jiang, D. Feng, Y. Hua, Y. Hu, Q. Liu, and Y. Zhang, "FastCDC: A fast and efficient Content-Defined chunking approach for data deduplication," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016, pp. 101–114.

[16] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *Proceedings of the eighteenth ACM symposium on operating systems principles*, 2001, pp. 174–187.

[17] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin, "Dynamic authenticated index structures for outsourced databases," in *ACM SIGMOD*, 2006, pp. 121–132.

[18] R. C. Merkle, "A certified digital signature," in *Conference on the Theory and Application of Cryptology*, 1989, pp. 218–238.

[19] W. Xia, H. Jiang, D. Feng, L. Tian, M. Fu, and Y. Zhou, "Ddelta: A deduplication-inspired fast delta compression approach," *Performance Evaluation*, vol. 79, pp. 258–272, 2014.

[20] A. Z. Broder, "Some applications of rabin's fingerprinting method," in *Sequences II: Methods in Communication, Security, and Computer Science*, 1993, pp. 143–152.

[21] M. O. Rabin, "Fingerprinting by random polynomials," *Technical report*, 1981.

[22] (2025) COLE$^+$ source code. [Online]. Available: https://github.com/hkbudb/cole_plus

[23] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "Blockbench: A framework for analyzing private blockchains," in *ACM SIGMOD*, 2017, pp. 1085–1100.

[24] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[25] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *ACM SIGMOD*, 2018, pp. 489–504.

[26] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann *et al.*, "ALEX: an updatable adaptive learned index," in *ACM SIGMOD*, 2020, pp. 969–984.

[27] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska, "Fiting-tree: A data-aware index structure," in *ACM SIGMOD*, 2019, pp. 1189–1206.

[28] J. Wu, Y. Zhang, S. Chen, J. Wang, Y. Chen, and C. Xing, "Updatable learned index with precise positions," *PVLDB*, pp. 1276–1288, 2021.

[29] T. Yu, G. Liu, A. Liu, Z. Li, and L. Zhao, "LIFOSS: a learned index scheme for streaming scenarios," *World Wide Web*, pp. 1–18, 2022.

[30] P. Ferragina and G. Vinciguerra, "The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds," *PVLDB*, pp. 1162–1175, 2020.

[31] P. Li, H. Lu, Q. Zheng, L. Yang, and G. Pan, "LISA: A learned index structure for spatial data," in *ACM SIGMOD*, 2020, pp. 2119–2133.

[32] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska, "Learning multi-dimensional indexes," in *ACM SIGMOD*, 2020, pp. 985–1000.

[33] J. Ding, V. Nathan, M. Alizadeh, and T. Kraska, "Tsunami: A learned multi-dimensional index for correlated data and skewed workloads," *PVLDB*, p. 74–86, 2020.

[34] S. Zhang, J. Qi, X. Yao, and A. Brinkmann, "Hyper: A high-performance and memory-efficient learned index via hybrid construction," *Proceedings of the ACM on Management of Data*, pp. 1–26, 2024.

[35] L. Liang, G. Yang, A. Hadian, L. A. Croquevielle, and T. Heinis, "Swix: A memory-efficient sliding window learned index," *Proceedings of the ACM on Management of Data*, pp. 1–26, 2024.

[36] J. Zhang, K. Su, and H. Zhang, "Making in-memory learned indexes efficient on disk," *Proceedings of the ACM on Management of Data*, pp. 1–26, 2024.

[37] H. Lan, Z. Bao, J. S. Culpepper, R. Borovica-Gajic, and Y. Dong, "A fully on-disk updatable learned index," in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 2024, pp. 4856–4869.

[38] H. Lan, Z. Bao, J. S. Culpepper, and R. Borovica-Gajic, "Updatable learned indexes meet disk-resident dbms-from evaluations to design choices," *Proceedings of the ACM on Management of Data*, pp. 1–22, 2023.

[39] Y. Dai, Y. Xu, A. Ganesan, R. Alagappan, B. Kroth, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "From WiscKey to bourbon: A learned index for Log-Structured merge trees," in *OSDI*, 2020, pp. 155–171.

[40] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi, "Towards scaling blockchain systems via sharding," in *Proceedings of the 2019 international conference on management of data*, 2019, pp. 123–140.

[41] M. El-Hindi, C. Binnig, A. Arasu, D. Kossmann, and R. Ramamurthy, "BlockchainDB: A shared database on blockchains," *PVLDB*, pp. 1597–1609, 2019.

[42] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: Scaling blockchain via full sharding," in *ACM CCS*, 2018, pp. 931–948.

[43] S. Gupta, S. Rahnama, J. Hellings, and M. Sadoghi, "ResilientDB: Global scale resilient blockchain fabric," *PVLDB*, p. 868–883, 2020.

[44] L. Jia, Y. Liu, K. Wang, and Y. Sun, "Estuary: A low cross-shard blockchain sharding protocol based on state splitting," *IEEE Transactions on Parallel and Distributed Systems*, pp. 405–420, 2024.

[45] Y. Xu, J. Zheng, B. Düdder, T. Slaats, and Y. Zhou, "A two-layer blockchain sharding protocol leveraging safety and liveness for enhanced performance," in *31th Annual Network and Distributed System Security Symposium, NDSS 2024*, 2024.

[46] C. Xu, C. Zhang, J. Xu, and J. Pei, "SlimChain: scaling blockchain transactions through off-chain storage and parallel processing," *PVLDB*, pp. 2314–2326, 2021.

[47] Z. Hong, S. Guo, E. Zhou, W. Chen, H. Huang, and A. Zomaya, "Gridb: Scaling blockchain database via sharding and off-chain cross-shard mechanism," *PVLDB*, pp. 1685–1698, 2023.

[48] Y. Teng, Z. Wang, Y. Gao, and W. Dong, "Timechain: A secure and decentralized off-chain storage system for iot time series data," in *The Web Conference (WWW) 2025*.

[49] H. Wang, C. Xu, C. Zhang, J. Xu, Z. Peng, and J. Pei, "vChain+: Optimizing verifiable blockchain boolean range queries," in *IEEE ICDE*, 2022, pp. 1927–1940.

[50] C. Xu, C. Zhang, and J. Xu, “vChain: Enabling verifiable boolean range queries over blockchain databases,” in *ACM SIGMOD*, 2019, pp. 141–158.

[51] Q. Liu, Y. Peng, Z. Tang, H. Jiang, J. Wu, T. Wang, T. Peng, and G. Wang, “veffchain: Enabling freshness authentication of rich queries over blockchain databases,” *IEEE Transactions on Knowledge and Data Engineering*, pp. 2285–2300, 2023.

[52] H. Wang, C. Xu, X. Chen, C. Zhang, H. Hu, S. Tian, Y. Yan, and J. Xu, “V2fs: A verifiable virtual filesystem for multi-chain query authentication,” in *IEEE ICDE*, 2024, pp. 1999–2011.

[53] C. Zhang, C. Xu, J. Xu, Y. Tang, and B. Choi, “GEM^2-Tree: A gas-efficient structure for authenticated range queries in blockchain,” in *IEEE ICDE*, 2019, pp. 842–853.

[54] C. Zhang, C. Xu, H. Wang, J. Xu, and B. Choi, “Authenticated keyword search in scalable hybrid-storage blockchains,” in *IEEE ICDE*, 2021, pp. 996–1007.

[55] Q. Liu, Y. Peng, M. Xu, H. Jiang, J. Wu, T. Wang, T. Peng, and G. Wang, “Mpv: Enabling fine-grained query authentication in hybrid-storage blockchain,” *IEEE Transactions on Knowledge and Data Engineering*, pp. 3297–3311, 2024.

[56] S. Li, Z. Zhang, J. Xiao, M. Zhang, Y. Yuan, and G. Wang, “Authenticated keyword search on large-scale graphs in hybrid-storage blockchains,” in *IEEE ICDE*, 2024, pp. 1958–1971.

[57] Q. Lin, B. Gu, and F. Nawab, “Rollstore: Hybrid onchain-offchain data indexing for blockchain applications,” *IEEE Transactions on Knowledge and Data Engineering*, 2024.

[58] B. Li, L. Lin, S. Zhang, J. Xu, J. Xiao, B. Li, and H. Jin, “Flexim: Efficient and verifiable index management in blockchain,” *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–14, 2025.

APPENDIX

A. Glossary

TABLE II
GLOSSARY

Term	Definition
LSM-tree	Log-structure-merge tree
root_hash_list	A list storing all the Merkle roots for both in-memory and on-disk levels
MB-tree	Merkle B-tree
RS-tree	Rewind-Supported tree
State File	Storing the latest versions of each state
Index File	Storing learned models for fast reads
Version File	Storing historical versions of each state
Hash File	Storing the complete Merkle Hash Tree built upon the version trees' root hash values
Checkpoint	A snapshot of the root_hash_list taken before the flush operation
CDC	Content-defined chunking
CDC Mask	A mask to control the average chunk size

B. Discussion for Redesigned CDC

In Section III, we have shown three essential modifications to the CDC design specifically for the blockchain context. The first modification involves establishing a maximum chunk size, f_{max}. This parameter limits the fanout of each tree node, which mitigates the risk of unbounded node growth. Such growth would inevitably compromise blockchain synchronization efficiency and impede the efficacy of state pruning. The second modification ensures that CDC cut points are consistently aligned with the blockchain's entry size. Blockchain data entries are characterized by their fixed size (256 bits for state and hash values) and atomic nature, meaning they cannot be subdivided. Arbitrary cut points, resulting from a lack of alignment, would violate this atomicity requirement and consequently degrade system performance. The third essential modification is to reset the CDC fingerprint at every cut point. This is paramount for COLE$^+$'s state pruning capabilities. Unlike traditional CDC algorithms that use information from preceding chunks to determine the next cut point, which is a valid approach when the entire data stream is always present, COLE$^+$ actively prunes historical states. Consequently, data prior to a cut point becomes inaccessible. To avoid dependencies on pruned data and ensure that COLE$^+$'s tree node boundaries are defined only by the contents of the current tree node, resetting the internal CDC fingerprint at each new node is a fundamental requirement.

Algorithm 5: Chain Reorganization with On-Disk Rewinds

```
1  Function Chain-Reorg($blk_{cur}, blk_{rew}, blk_{can}$)
       Input: Current block $blk_{cur}$, rewound block $blk_{rew}$, latest
              canonical block $blk_{can}$
2      $\mathcal{L}_{cur} \leftarrow$ Get root_hash_list for $blk_{cur}$;
3      $\mathcal{L}_{rew} \leftarrow$ Get most recent root_hash_list happening before
        $blk_{rew}$;
4      $n_{cur} \leftarrow |\mathcal{L}_{cur}|$; $n_{rew} \leftarrow |\mathcal{L}_{rew}|$; $idx \leftarrow 0$;
       /* Identify the unchanged runs via hash
          values                                        */
5      foreach $\langle h_{cur}, h_{rew} \rangle$ in $\mathcal{L}_{cur}.rev(), \mathcal{L}_{rew}.rev()$ do
6          if $h_{cur} \neq h_{rew}$ then break ;
7          $idx \leftarrow idx + 1$;
       /* Keep the unchanged disk runs                  */
8      for $h_{cur} \in \mathcal{L}_{cur}[n_{cur} - idx, n_{cur} - 1]$ do
9          Retain the disk run w.r.t. $h_{cur}$;
       /* Rebuild the inconsistent disk runs and
          RS-tree                                       */
10     for $h_{rew} \in \mathcal{L}_{rew}[2, n_{rew} - idx - 1].rev()$ do
11         $\langle l_{rew}, r_{rew} \rangle \leftarrow h_{rew}.meta()$;
12         if $h_{rew}$ corresponds to a disk run then
13             Rebuild $h_{rew}$'s run using $\{s_i | s_i \in h_{cur}$'s run s.t.
                $s_i.blk \in \langle l_{rew}, r_{rew} \rangle, h_{cur} \in \mathcal{L}_{rew}\}$;
14         else
15             Rebuild $h_{rew}$'s RS-tree using $\{s_i | s_i \in h_{cur}$'s run s.t.
                $s_i.blk \in \langle l_{rew}, r_{rew} \rangle, h_{cur} \in \mathcal{L}_{rew}\}$;
16     Discard runs whose hash values are not in $\mathcal{L}_{rew}$;
       /* Append the remaining blocks of the
          canonical chain                               */
17     for $blk \in$ sub-chain from $\mathcal{L}_{rew}$'s block to $blk_{can}$ do
18         Execute TXs in $blk$;
```

C. Chain Reorganization with On-Disk Rewinds

In Section III-C, our primary focus is on chain reorganization at the in-memory level. Thanks to the careful design of temporary checkpoints, only the in-memory RS-trees need to be rewound, while the on-disk runs remain unchanged, resulting in highly efficient chain reorganization. This type of reorganization occurs frequently due to the eventual consistency consensus protocols like Proof of Work in Bitcoin [1] and Proof of Stake in Ethereum [2]. However, there are cases where the state rewind extends beyond the in-memory level, such as during Ethereum's fork following the DAO attack [9], where lots of blocks are rewound. In such scenarios, simply rewinding the RS-trees is insufficient because the on-disk runs have changed due to flush and recursive merge operations in the LSM-tree. To address this challenge, additional steps are necessary. The rewind point is tied to the most recent checkpoint prior to the common ancestor block shared with the canonical chain, as runs are only updated at these checkpoints. The changed on-disk runs are rebuilt using blockchain states stored in the current version index. Note that no transaction re-execution is required for rebuilding the on-disk runs, since the necessary data already exists, albeit distributed across different on-disk runs. To further facilitate this rebuilding process, each checkpoint stores not only the root hash value of each run, but also metadata that includes the minimum and maximum block heights of the run's updated states.

Algorithm 5 shows the procedure of chain reorganization for on-disk levels. It takes three inputs: the current block blk_{cur}, the rewound block blk_{rew} (the most recent common ancestor shared by the current and canonical chains), and the latest canonical block blk_{can}. First, both the snapshot corresponding to blk_{cur} and the most recent snapshot that happens before blk_{rew} are retrieved. (Lines 2 to 3). Next, to determine which runs should be retained, the common hash values between $\mathcal{L}_{cur}$ and $\mathcal{L}_{rew}$ are identified by iterating each list from the end (Lines 5 to 7). This reverse iteration is motivated by the LSM-tree merge sequence, where newer runs are more likely to have changed than older ones. The index, idx, records the number of these matching hashes. Any disk runs associated with these common hash values remain unchanged during the state rewind. For the remaining (changed) hash values in $\mathcal{L}_{rew}$, their corresponding disk runs or $\mathrm{RS\text{-}tree}$ are rebuilt (Lines 10 to 15). Specifically, the minimum and maximum block heights of the rebuilt run or $\mathrm{RS\text{-}tree}$ (i.e., l_{rew}, r_{rew}) are obtained from the metadata. The rebuild then uses the states in the current index whose update versions fall in $\langle l_{rew}, r_{rew} \rangle$. Runs whose hash values are absent from $\mathcal{L}_{rew}$ are then discarded. Finally, the remaining blocks of the canonical chain are appended, which is similar to the chain reorganization for the in-memory level.

D. Discussion for Adversarial Issues and ACID Properties

COLE$^+$ focuses solely on the blockchain storage layer. Although blockchain systems as a whole are designed for adversarial environments, the adversarial issues are already addressed by the consensus protocol and cryptographic primitives. The storage layer of a blockchain system only needs to support basic data reads and writes while ensuring traditional ACID properties.

The atomicity is achieved by maintaining root_hash_list in an atomic manner. During the level merges, root_hash_list is updated atomically only after the completion of constructing all the files in the new level, followed by removing the old level files. This ensures data consistency because the old level files remain intact and are referenced by root_hash_list during a node crash. Concurrency control is not considered because the consensus protocol ensures the write serializability. The Merkle-based structures guarantee data integrity. For durability, COLE$^+$ utilizes checkpoints and blockchain transaction logs as Write-Ahead Logs to prevent data loss. Checkpoints are created each time the in-memory index flushes to the disk. At this moment, only the waiting group is full, while the dynamic group is empty, and all other levels have already been persisted to disk. In the event of a crash at any point, the system can revert to the last checkpoint and safely discard all files associated with unfinished flush or merge operations. To reconstruct the in-memory waiting group, the system replays transactions occurring between the last two checkpoints. Similarly, to reconstruct the dynamic group created after the checkpoint, it replays additional transactions starting from the checkpoint up to the latest block. If a crash occurs during in-memory chain reorganization, we halt rewinding the in-memory RS-tree (since its state will be lost upon restart). Instead, the system treats this as a disk-level chain reorganization. In either case, a crash forces restoration to the most recent checkpoint preceding the rewind point, from which chain reorganization is restarted.

E. Proof of Correctness for Chain Reorganization

Theorem 1 (Chain Reorganization Correctness)**.** *COLE$^+$ supports chain reorganization with both in-memory rewind and on-disk rewind, and ensures consistent index digest between nodes with and without chain reorganization.*

Proof. The index digest is computed from the hash values in the root_hash_list. To ensure its consistency, we analyze each entry in the list. For the dynamic group, it is guaranteed to have the same hash owing to the fact that $\mathrm{RS\text{-}tree}$'s hash depends completely on its stored data. For the waiting group and all on-disk runs, we prove by contradiction. Assume there is any inconsistency, it would imply there are some write operations in the waiting group or on-disk runs. This is impossible if the entire rewind happens at the dynamic group as all other LSM-tree runs remain unchanged during the time span. If the rewind occurs in the waiting group, an observed inconsistency would imply that multiple flush operations occurred during the rewind, altering the snapshot hash values in the root_hash_list. This contradicts the assumption that the rewind happens solely in the waiting group. Alternatively, if the rewind extends to on-disk levels, the LSM-tree runs are rebuilt using the blockchain states in the existing index. An inconsistency would indicate additional merges occurred between snapshots, which is impossible since snapshots are taken before each flush operation. □

F. Proof of Correctness for Version Tree

Theorem 2 (Version Tree Correctness)**.** *The proposed version tree ensures a consistent root hash between full archive nodes and pruned nodes after merge operations.*

Proof. Since the structure of the version tree is fully determined by its content, different blockchain nodes should derive the same root hash value, provided they have sufficient information to construct the merged version tree. This is trivial for full archive nodes. For pruned nodes, we prove by contradiction that retaining tree nodes along the boundary paths is sufficient to construct the merged version tree. Assume, for contradiction, that pruned nodes cannot compute certain tree nodes during merging. This implies that updates occurred outside the boundary paths, beyond the retained nodes and the bounds established by the CDC pattern. Clearly, this is impossible as it violates CDC method's locality property. □

G. Complexity Analysis of Storage Reduction through State Pruning

We finally analyze the storage reduction achieved by state pruning. Assume the number of versions for an address, denoted as $\mathcal{V}(addr)$, follows a Zipfian distribution, meaning a small number of addresses hold most historical versions. In COLE$^+$, state pruning operates on each address's version tree. Therefore, we focus on the address with the most amount of historical versions. The size of a version tree is given by $O\left(f_{exp} \times (|k| + |v|) \times \frac{\mathcal{V}(addr)}{f_{exp}-1}\right)$, where f_{exp} is the expected fanout, $|k|$ is the size of the search key (i.e., version number), and $|v|$ is the size of the entry value (state value for leaf nodes or hash value for internal nodes). The height of the version tree is $\lceil \log_{f_{exp}} \mathcal{V}(addr) \rceil$. After state pruning, only the boundary paths are retained. The storage saving from pruning is:

$$O\left(f_{exp} \times (|k| + |v|) \times \left(\frac{\mathcal{V}(addr)}{f_{exp}-1} - 2 \times \lceil \log_{f_{exp}} \mathcal{V}(addr) \rceil + 1\right)\right)$$

For a large number of historical versions ($\mathcal{V}$ is large), the difference between $\frac{\mathcal{V}(addr)}{f_{exp}-1}$ and $2 \times \lceil \log_{f_{exp}} \mathcal{V}(addr) \rceil$ becomes asymptotically significant. This demonstrates the potential storage savings of state pruning while maintaining correctness.

H. Additional Experiment: Impact of Parameters

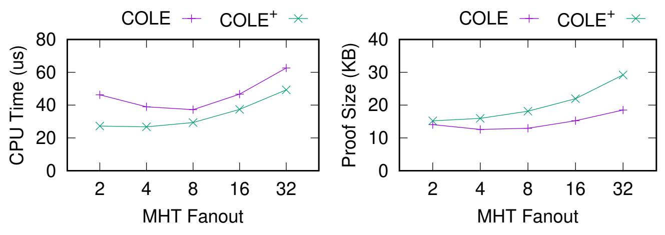


Fig. 14. Impact of MHT Fanout

To evaluate the impact of fanout on CPU time and proof size for provenance queries, we vary the fanout exponentially from 2 to 32, while maintaining a fixed block height range of 16. As shown in Figure 14, the CPU time exhibits a U-shaped trend, while the proof size of COLE$^+$ increases with larger fanouts. This occurs because larger fanouts reduce tree height, improving search efficiency, but also result in larger node sizes, increasing the proof size. The optimal fanout is 8 for COLE and 4 for COLE$^+$. We set the default fanout to 4 in subsequent experiments.

Figure 15 compares the throughput and latency of COLE and COLE$^+$ for various size ratios T under a block height of 6×10^5. The throughput increases slightly with larger size ratios, while the latency decreases for both COLE and COLE$^+$. We set the default size ratio to 10, as it yields the highest throughput and relatively low latency for COLE$^+$.

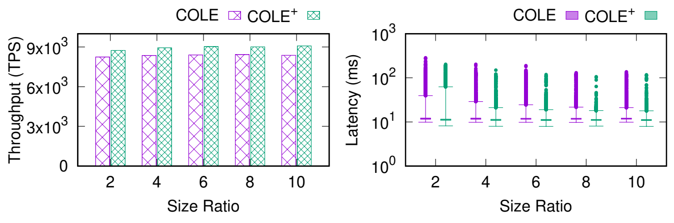


Fig. 15. Impact of Size Ratio

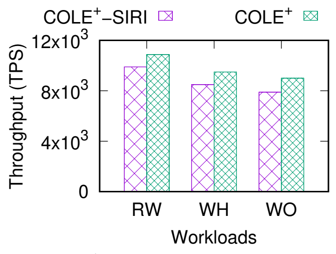


Fig. 16. COLE$^+$-SIRI Performance vs. Workloads

I. Additional Experiment: More on Ablation Study

To evaluate the effectiveness of RS-tree versus SIRI [10], we test under varying workloads: (i) Read-Write (RW, 50% update, 50% read); (ii) Write-Heavy (WH, 75% update, 25% read); and (iii) Write-Only (WO, 100% update). As shown in Figure 16, COLE$^+$'s performance advantage increases with higher write intensity, rising from 10% to 14%, demonstrating the robustness of the RS-tree. Furthermore, it is worth noting that the CDC used in SIRI cannot directly support chain reorganization and state pruning, as discussed in Appendix B.

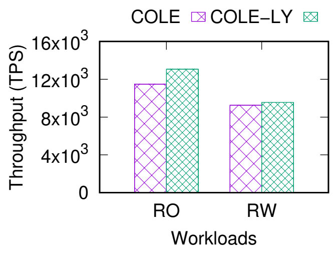


Fig. 17. COLE-LY Performance vs. Workloads

To assess the impact of the design of separating the latest and historical states, we compare baseline COLE with a variant of COLE featuring the same separated layout, denoted as COLE-LY. As shown in Figure 17, COLE-LY improves throughput by up to 14% over COLE under varying workloads. Thanks to the new layout, the search space of the latest states is reduced, thereby enhancing overall system throughput.