COLE Final Report

Yasmine Vaziri

August 20, 2024

Contents

1 Introduction **3**

2 Introduction to ChainKV, COLE and BlockLSM **4**
2.1 ChainKV . . . 4
2.2 Column-based Storage . . . 4
2.3 Block-LSM: An Efficient Data Storage Method for Ethereum . . . 5
2.3.1 Block-LSM Overview . . . 5
2.3.2 Challenges and Solutions . . . 5

3 The Difference and Goal of Each Approach **5**
3.1 Data Storage Approach . . . 5
3.2 subComponents, Operation, and Recovery . . . 6
3.2.1 COLE . . . 6
3.2.2 BlockLSM . . . 7
3.2.3 Detailed Mechanisms . . . 7
3.2.4 Performance Evaluation . . . 8
3.2.5 Architecture . . . 8
3.2.6 Data Flow and Operations . . . 9
3.2.7 ChainKV . . . 9
3.3 Zone Differences between ChainKV and BlockLSM . . . 10
3.4 Compaction Reduction in ChainKV . . . 10
3.4.1 Compaction Behavior Analysis . . . 10
3.5 Compaction Reduction in BlockLSM . . . 10
3.6 Key Differences between BlockLSM and ChainKV . . . 11
3.7 Organizational Principle . . . 11
3.7.1 Compaction Strategies . . . 11
3.8 Data Locality . . . 11
3.9 Cache Management . . . 12
3.10 Recovery Mechanisms . . . 13
3.11 Performance and Storage Reduction . . . 13
3.11.1 Performance Improvement . . . 13
3.11.2 Storage Reduction . . . 14

4 COLE's Experiments and Results **14**
4.1 The Storage Consumption in SmallBank Benchmark vs. different Block Heights in Different Indexes . . . 14
4.2 Role of Scale . . . 15
4.3 The Throughputs in SmallBank Benchmark vs. different Block Heights in Different Indexes . . . 16
4.3.1 Reason for the Discrepancy Throughput . . . 18
4.3.2 Solution to Improve COLE's Throughput . . . 18
4.4 The Storage Consumption in KV store Benchmark vs. different Block Heights in Different Indexes . . . 18
4.4.1 Reasons Behind the Observed Trends . . . 20
4.4.2 Breakdown into these workloads and proof . . . 21
4.5 The Throughputs in KV store Benchmark vs. different Block Heights in Different Indexes 21
4.5.1 Analysis . . . 23
4.5.2 Improvements . . . 24
4.6 Throughput vs. Workload in different block heights . . . 24
4.7 Reasons Behind Performance Differences . . . 25
4.7.1 Improvements . . . 26
4.8 Latency in Small Bank . . . 26
4.9 Latency in KV store . . . 28
4.10 Analysis . . . 28
4.11 Latency in KV store in Read only and Half Read-Half Write . . . 30
4.11.1 Possible Future Work . . . 31

5 Problems and Possible Future Work **32**
5.1 Synchronous Merging in COLE . . . 32
5.1.1 Problem . . . 32
5.1.2 Specific Issues Observed . . . 32
5.1.3 Possible Improvements . . . 32
5.2 Asynchronous Merging in COLE* . . . 32
5.2.1 Problem . . . 32
5.2.2 Specific Issues Observed . . . 33
5.2.3 Possible Improvements . . . 33
5.3 Scalability Issues with Increasing Block Height . . . 33
5.3.1 Problem . . . 33
5.3.2 Specific Issues Observed . . . 33
5.3.3 Possible Improvements . . . 34
5.4 Latency Variability, Especially in COLE* . . . 34
5.4.1 Problem . . . 34
5.4.2 Specific Issues Observed . . . 34
5.4.3 Possible Improvements . . . 34

6 Conclusion **34**

Abstract

Blockchain technology is integral to modern distributed systems, offering secure, immutable, and transparent data management. As the complexity and scale of blockchain networks increase, the efficiency of underlying storage mechanisms becomes crucial for maintaining optimal performance. In this report, I present a comparative analysis of three prominent blockchain storage systems: COLE (A Column-based Learned Storage), ChainKV, and BlockLSM. COLE's innovative column-based design aims to optimize data retrieval and minimize storage overhead, chainKV utilizes a semantic-aware zoning strategy to improve data locality and retrieval efficiency, particularly in read-heavy environments. BlockLSM leverages a prefix-based hashing approach within an LSM tree framework to enhance data locality and reduce compaction overhead, making it well-suited for write-intensive applications.

Through a detailed examination of each system's architecture, I identify their respective strengths and weaknesses. I then replicate and extend the experiments originally conducted for COLE, testing various indexing strategies, storage sizes, and throughput conditions to assess its performance against ChainKV and BlockLSM.

Keywords— Blockchain Storage, COLE, ChainKV, BlockLSM, Column-Based Storage, Semantic Zones, Prefix Hashing, Data Locality, Compaction Techniques, Write Amplification

1 Introduction

Blockchain technology has revolutionized how distributed systems manage data, enabling secure, immutable, and transparent record-keeping. As blockchain systems grow in scale and complexity, the underlying storage mechanisms play a crucial role in ensuring efficient data retrieval, storage management, and overall system performance. Various approaches have been developed to address the unique challenges of blockchain storage, among which I compared COLE, ChainKV, and BlockLSM.

COLE (A Column-based Learned Storage) introduces a novel approach to blockchain storage by utilizing a column-based design. This design aims to optimize data retrieval times and reduce storage overhead by organizing data into columns rather than rows, which is typical in traditional databases. The COLE system emphasizes reducing redundancy and improving query performance, but it does not implement traditional compaction techniques, potentially limiting its effectiveness in handling large-scale data.

ChainKV utilizes a semantic-aware zoning strategy, which improves data locality and access speed by grouping related data together in semantic zones. This approach enhances the efficiency of data retrieval and storage management, especially in read-heavy workloads. BlockLSM, on the other hand, employs a prefix-based hashing method to manage data within a log-structured merge tree (LSM tree) framework. This method enhances data locality and reduces compaction overhead, making BlockLSM particularly effective for write-heavy blockchain applications.

Given the different approaches these systems take, a comparative analysis is essential to understand their respective strengths and weaknesses. In this work, I began by thoroughly examining the COLE system as outlined in its original proposal. I then conducted a detailed review of ChainKV and BlockLSM, focusing on how these systems address issues such as data locality, compaction efficiency, and write amplification.

To further investigate the practical implications of these systems, I conducted a series of experiments replicating the scenarios outlined in the COLE paper. These experiments were designed to assess the performance of COLE under various conditions, including different indexing strategies, storage sizes, and throughput requirements. By comparing these results with the performance characteristics of ChainKV and BlockLSM, I aim to identify specific areas where COLE may fall short and propose potential enhancements.

In the following sections, I will detail my comparative analysis, present the results of the experiments. I will also explore potential improvements to COLE that could enhance its performance.

2 Introduction to ChainKV, COLE and BlockLSM

2.1 ChainKV

ChainKV is a key-value store specifically designed for blockchain systems. It addresses the challenges of data integrity, consistency, and efficient query support in a decentralized environment. The primary goal of ChainKV is to manage blockchain states effectively, ensuring that the states are updated consistently across the network while minimizing storage overhead.

ChainKV employs a combination of Merkle Patricia Trie (MPT) and a blockchain-specific version of the Log-Structured Merge-Tree (LSM-Tree) to achieve these goals. The MPT ensures data integrity by creating a hash chain of the states, while the LSM-Tree optimizes write operations by organizing data into levels that are merged periodically. This combination allows ChainKV to efficiently handle frequent updates while maintaining the ability to retrieve historical state data.

In ChainKV, each state is identified by a unique address and stored in a structure that allows for efficient retrieval and verification. The system supports three primary operations:

- **Put(addr, value)**: Inserts or updates the state associated with a given address.
- **Get(addr)**: Retrieves the latest value associated with a given address.
- **ProvQuery(addr, [blkl, blku])**: Performs a provenance query to retrieve the historical values of a state within a specified block range.

The design of ChainKV ensures that it can scale with the increasing size of blockchain data while providing the necessary guarantees for data integrity and consistency.

2.2 Column-based Storage

Column-based storage, as implemented in the COLE system, is an innovative approach to managing blockchain data. It is designed to address the limitations of traditional storage mechanisms, particularly in terms of storage size and query efficiency. COLE stands for Column-based Learned Storage, and it leverages the concept of a columnar database to store blockchain states.

In COLE, the historical values of each blockchain state are stored contiguously, much like columns in a traditional columnar database. This design allows for efficient retrieval of historical data, as all versions of a state are stored together. To facilitate this, COLE uses a learned index model that replaces traditional tree-based indexes with models trained on the data distribution. These learned models predict the location of a state within the storage, significantly reducing the size of the index and improving query performance.

COLE is optimized for disk-based environments, which is crucial for blockchain systems where data is stored persistently. The system employs a Log-Structured Merge-Tree (LSM-Tree) strategy to manage the learned models and ensure efficient write operations. This involves organizing the storage into multiple levels, where data is initially stored in an in-memory index and then periodically merged into on-disk levels. Each on-disk level uses a disk-optimized learned model to index the data, ensuring that queries can be processed with minimal disk I/O.

The key benefits of the column-based storage approach in COLE include:

- **Reduced Storage Size**: By storing state versions contiguously and using learned indexes, COLE significantly reduces the storage overhead compared to traditional methods like the Merkle Patricia Trie.
- **Efficient Data Retrieval**: The use of learned models allows for fast queries, as the models can predict the location of data with high accuracy.
- **Support for Provenance Queries**: COLE is designed to support blockchain-specific queries that require the retrieval of historical state values, making it well-suited for applications in blockchain systems.

2.3 Block-LSM: An Efficient Data Storage Method for Ethereum

Ethereum is one of the most widely used blockchain systems across various domains, including medicine, economics, the Internet of Things (IoT), software engineering, and digital assets. For Ethereum to function correctly, every full node in the network must store all transaction data locally. However, as the Ethereum network expands, the amount of data has grown significantly, reaching approximately 800 GB. This rapid increase in data size necessitates an efficient data storage system to ensure smooth operation.

Traditional methods for managing this large data volume involve either data removal or compression. However, these methods often lead to information loss and more complex data queries. Additionally, they tend to overlook the interaction between blockchain data and the underlying storage system. Currently, Ethereum employs a Log-Structured Merge-tree (LSM-tree) based key-value (KV) storage engine, which is efficient for data writing but introduces inefficiencies during data synchronization due to loss of block order information. This results in unnecessary input/output (I/O) operations and reduced performance.

2.3.1 Block-LSM Overview

To address these challenges, the Block LSM paper introduces a novel approach called **Block-LSM**. Block-LSM is an Ether-aware LSM-tree-based key-value storage method designed to preserve the block structure of Ethereum data. The key innovation of Block-LSM lies in its consideration of the Ethereum block sequence during data storage, which significantly enhances efficiency.

2.3.2 Challenges and Solutions

The paper identifies three main challenges:

1. **Coordinating Ethereum's Structure with the Storage System:** How to align Ethereum's blockchain structure with the storage system to improve efficiency?
2. **Minimizing Extra Work:** How to transfer Ethereum's data structure to the storage system with minimal overhead?
3. **Ensuring Query Efficiency:** How to ensure that data queries remain correct and efficient?

To tackle these challenges, the authors propose several solutions:

1. **Performance Analysis:** The study begins by analyzing Ethereum's performance, revealing that the LSM-tree structure causes high I/O operations during data synchronization.
2. **Prefix-based Hashing:** Block-LSM introduces a prefix-based hashing technique that uses block numbers to maintain the order of data. This method simplifies data storage and access by keeping the sequence of Ethereum transactions intact.
3. **Block Grouping:** To reduce the overhead associated with prefixing and memory buffer maintenance, Block-LSM groups multiple blocks together. This grouping helps in maintaining query efficiency and minimizing storage overhead.

3 The Difference and Goal of Each Approach

The comparison between COLE, ChainKV, and BlockLSM highlights the strengths and weaknesses of each system. COLE is highly efficient in storage due to its column-based design, ChainKV excels in write throughput with its semantic-aware zoning, and BlockLSM performs well in both write throughput and storage efficiency due to its prefix-based hashing approach.

3.1 Data Storage Approach

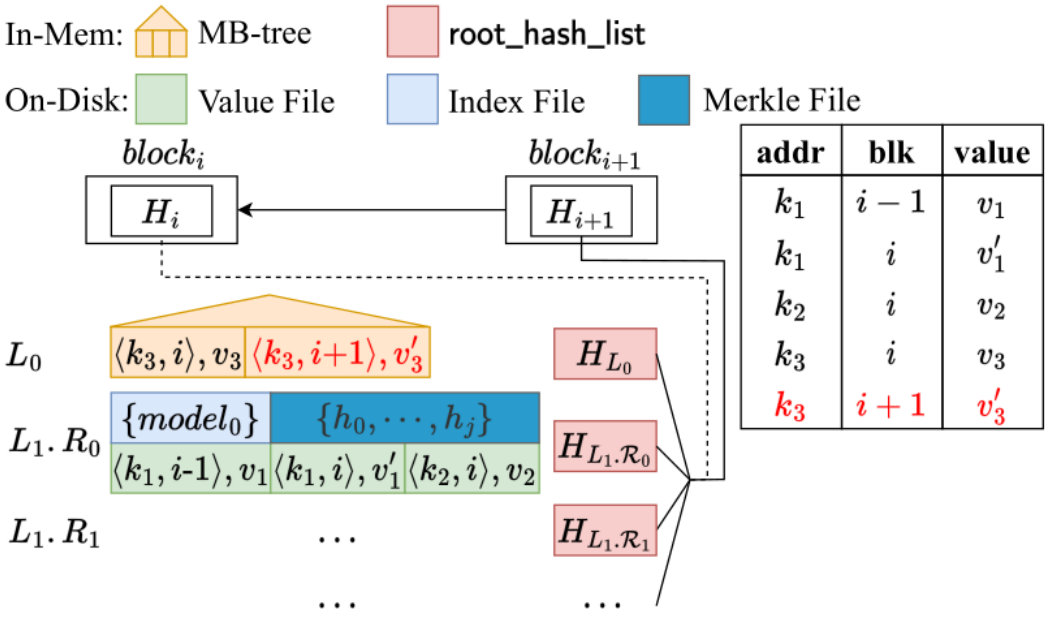
BlockLSM utilizes a **prefix-based hashing** approach to manage the storage of blockchain data efficiently. This method allows the system to maintain a structured order of data blocks, enabling quicker access and retrieval of specific data points.

ChainKV uses a **semantic-aware zone** strategy, categorizing data based on its meaning or semantic context. This method helps improve the system's ability to locate and retrieve data more efficiently.

COLE adopts a **column-based design** for storing blockchain data. This approach stores different versions of data contiguously, facilitating efficient data retrieval and reducing storage overhead.

3.2 subComponents, Operation, and Recovery

3.2.1 COLE



Components:

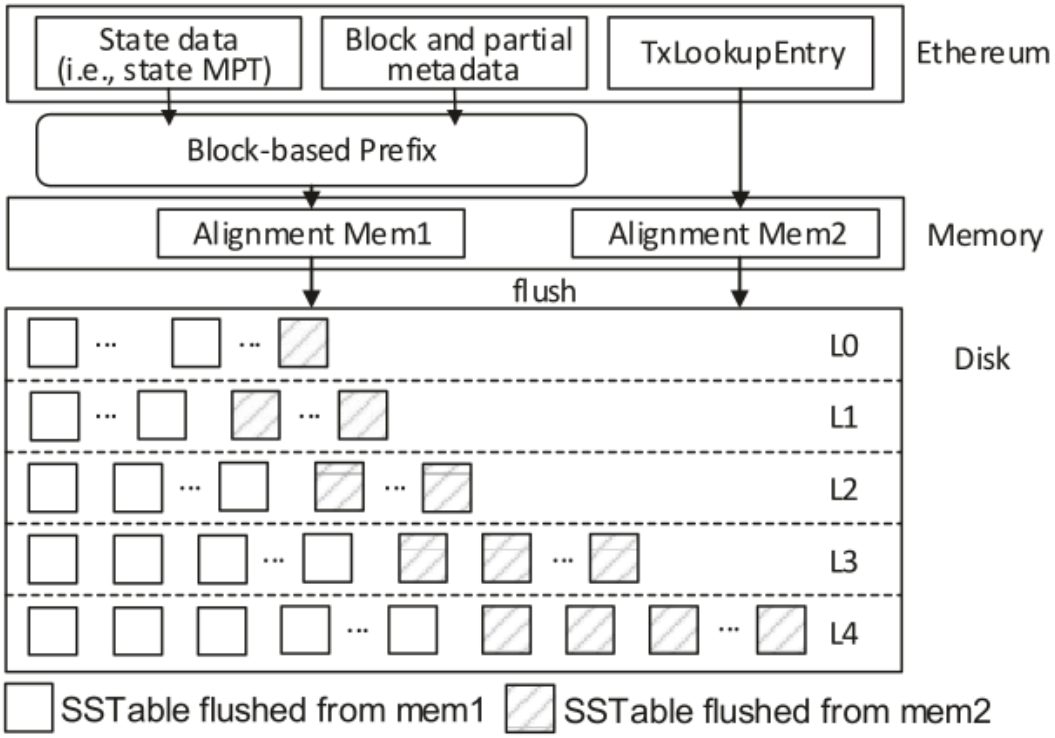
- **In-Mem MB-tree:** An in-memory structure that organizes the data in memory before it is flushed to disk.
- **On-Disk Value File:** Stores the actual data values on disk.
- **Index File:** Maintains an index for fast lookup of data.
- **Merkle File:** Ensures data integrity through hash chains.

Operation:

- **Blocks:** Data is divided into blocks, each containing multiple values.
- **Leveling:** Data is organized into different levels (L0, L1, etc.), with L0 being the most recent and highest level containing older data.
- **Columnar Storage:** Data is stored in columns, allowing for high compression and efficient retrieval.

Recovery: COLE utilizes a root hash list for quick recovery, ensuring data integrity and consistency.

3.2.2 BlockLSM



Components:

- **Alignment Mem Tables:** Two memory tables (Mem1 and Mem2) for organizing data before flushing to disk.
- **Block-based Prefix:** Data is prefixed and organized into blocks for efficient storage and retrieval.
- **SSTables:** Sorted String Tables store data on disk, organized by levels (L0 to L3).

Operation:

- **Flushing:** Data from memory tables is flushed to disk in the form of SSTables.
- **Compaction:** Periodic merging of SSTables to maintain efficiency and reduce redundancy.
- **Levels:** Data is managed across multiple levels, with higher levels containing older, less frequently accessed data.

3.2.3 Detailed Mechanisms

Ethereum's current LSM-tree structure is divided into two main components: a memory component and a disk component. Data first enters the memory component, where it is organized into larger, sequential batches for efficient writing. Once the memory component is full, the data is flushed to disk, where it is stored in levels. However, as the amount of synchronized data increases, the need for compaction operations also rises sharply. Compaction, which involves merging and sorting overlapping data ranges, consumes significant disk bandwidth and slows down data writing.

The problem with compaction in Ethereum's current setup arises from the loss of sequence information during data conversion into KV pairs. The hashing process used in traditional LSM-trees strips away the block order, leading to inefficient data organization and increased compaction overhead.

Block-LSM addresses these issues through the following innovations:

- **Prefix-based Hashing:** Block-LSM uses block numbers as prefixes for KV entries, ensuring that the block order is preserved during data storage. This approach reduces the need for costly compaction operations by maintaining the original sequence of Ethereum data.
- **Block Group-based Prefix:** To minimize space overhead, Block-LSM groups multiple blocks together and uses a group number as a prefix. This technique reduces the number of unique prefixes required, thereby saving space and improving efficiency.
- **Attribute-oriented Memory Buffers:** Special metadata, such as `TxLookupEntry`, is managed separately from other data to avoid key range overlaps. This separation reduces unnecessary compaction and maintains query performance.

3.2.4 Performance Evaluation

The authors conducted extensive experiments to evaluate Block-LSM's performance. They synchronized various sizes of transaction blocks and measured key metrics such as throughput, compaction operations, write amplification, and execution time. The results show that Block-LSM significantly improves data synchronization performance compared to Ethereum's original setup. Block-LSM achieves higher throughput, reduces compaction operations, and lowers the write amplification coefficient, making data synchronization faster and more efficient.

3.2.5 Architecture

1. Ethereum Data Types:

   The architecture handles three main types of Ethereum data:

   State Data (i.e., state MPT): This represents the Merkle Patricia Trie (MPT) used by Ethereum to store state data, which includes account balances, contract states, and more.

   Block and Partial Metadata: This includes the actual block data and some associated metadata, essential for maintaining the blockchain's integrity.

   TxLookupEntry: This is a specialized metadata used for transaction lookups, helping to locate specific transactions within the blockchain efficiently.

2. Block-based Prefix:

   A key innovation in Block-LSM is the use of a block-based prefix. This prefix preserves the order of blocks, ensuring that data associated with each block remains grouped together during storage. The prefix is derived from the block number, which simplifies the process of data retrieval and maintains the natural sequence of blockchain transactions.

3. Alignment Memory Buffers:

   Block-LSM uses two primary in-memory buffers, Alignment Mem1 and Alignment Mem2. These buffers temporarily store incoming data before it is written (flushed) to disk:

   Alignment Mem1: Handles most of the state data and block metadata.

   Alignment Mem2: Specifically manages the TxLookupEntry data.

   These buffers help organize the data in memory before it is committed to the disk, ensuring that it is written in an efficient manner that respects the block order.

4. Disk Storage and SSTables:

   Once the in-memory buffers (Mem1 and Mem2) are full, the data is flushed to disk and stored in Sorted String Tables (SSTables). SSTables are immutable, sorted data files that are the core of the LSM-tree structure. The SSTables are arranged in a hierarchical manner across different levels:

   L0 (Level 0): This is the initial level where the flushed data is stored. SSTables from Mem1 and Mem2 are written to L0, with Mem1's data and Mem2's data being differentiated by shading in the figure.

   L1 to L4 (Level 1 to Level 4): As the data grows and compaction operations occur, the SSTables are moved to deeper levels (L1, L2, etc.). Each level contains larger, more compacted SSTables, with fewer overlapping key ranges, which improves query performance.

3.2.6 Data Flow and Operations

Flushing and Compaction:

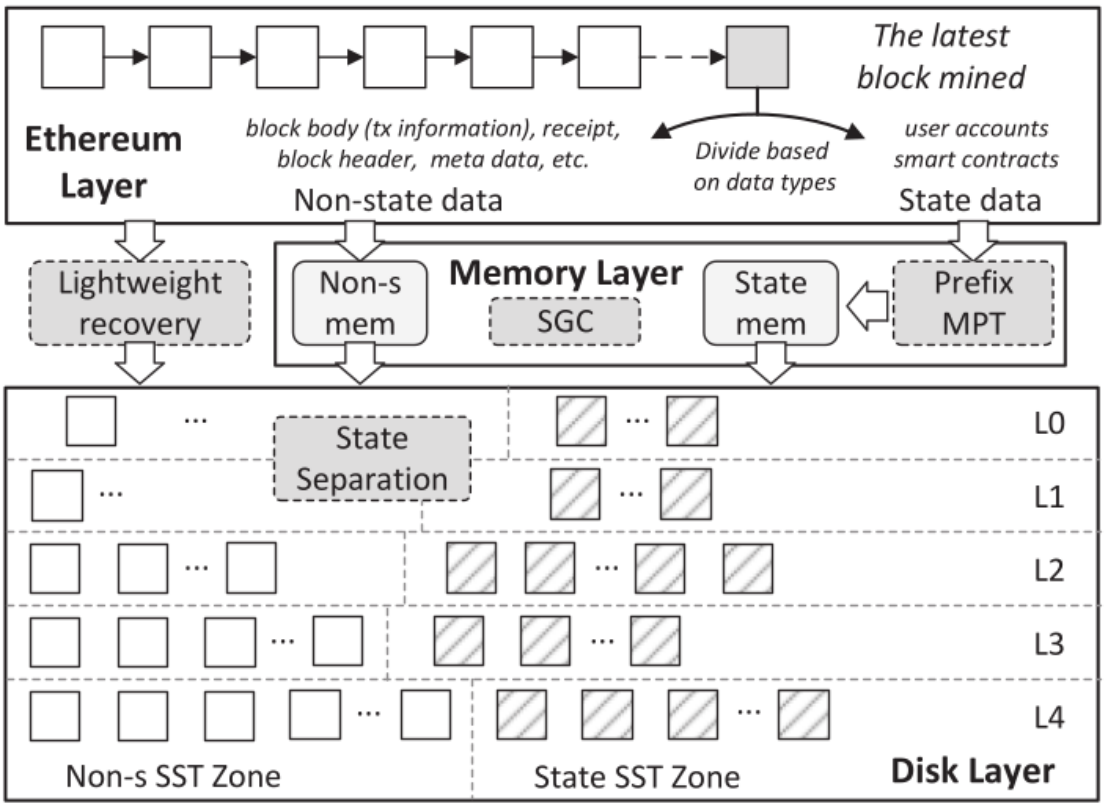
The process begins with data being written to the in-memory buffers. Once these buffers are full, the data is flushed to disk as SSTables in L0.

Over time, as more data is flushed, the SSTables in L0 are compacted and moved to lower levels (L1, L2, etc.). Compaction involves merging SSTables to reduce the number of files and eliminate key overlaps, which enhances read performance.

Separation of Metadata:

To avoid unnecessary compactions and maintain query efficiency, Block-LSM separates certain metadata, such as TxLookupEntry, from the main block and state data. This separation ensures that different types of data are compacted and accessed in ways that are most efficient for their specific use cases.

3.2.7 ChainKV



Components:

- **Ethereum Layer:** Handles block data and smart contract execution.
- **Memory Layer:**
  - **SGC (Space Gaming Cache):** Optimizes cache usage to enhance performance.
  - **Prefix MPT (Merkle Patricia Trie):** Ensures efficient storage and retrieval of state data.
- **Disk Layer:** Separates state and non-state data into different SST (Sorted String Table) zones.

Operation:

- **State Separation:** Differentiates between state and non-state data, optimizing storage.
- **Recovery:** Lightweight recovery mechanism ensures quick restoration of data.
- **Levels:** Data is organized into levels (L0 to L4), with each level managing data in SST zones.

3.3 Zone Differences between ChainKV and BlockLSM

ChainKV organizes data based on meaning and usage patterns. This optimizes storage and retrieval by understanding the nature of the data, grouping related data semantically, and improving query performance by maintaining contextual relevance and proximity of related information.

BlockLSM uses prefix-based hashing to organize data, grouping related data under common prefixes, maintaining data order, and ensuring efficient lookups. This reduces fragmentation and enhances range query efficiency.

3.4 Compaction Reduction in ChainKV

- **BlockLSM:** Includes a compaction process that reduces data redundancy and storage costs by periodically merging data blocks and eliminating obsolete data.
- **ChainKV:** Implements compaction techniques similar to BlockLSM, helping maintain storage efficiency.
- **COLE:** Does not apply traditional compaction techniques but relies on its column-based design and learned indexing to manage data efficiently without the need for periodic data compaction.

3.4.1 Compaction Behavior Analysis

ChainKV achieves high write throughput, especially with larger data sets, by optimizing compaction processes. The system reduces the number of necessary compactions through its semantic-aware zoning, maintaining a low write amplification coefficient (WA Coef), which minimizes data rewrites during compactions.

By organizing data based on semantics and usage patterns, ChainKV optimizes compaction processes, reducing the need for frequent data reorganization. it also employs strategies tailored to its semantic-aware zones, ensuring compactions are performed efficiently with minimal write amplification. **Algorithm for Reduction:**

- **Semantic-Aware Zoning:** Organizes data into semantic-aware zones based on data meaning and usage patterns, optimizing storage, retrieval processes, and reducing compaction frequency.

  By organizing data based on its semantics, ChainKV can ensure that related data is compacted together. This reduces unnecessary data rewrites and improves the efficiency of the compaction process.
- **Level-Based Compaction:** Similar to Log-Structured Merge (LSM) trees, data is compacted from higher to lower levels, reducing the overall number of compactions.

  Data is moved from higher levels (with smaller capacity and more frequent compactions) to lower levels (with larger capacity and fewer compactions). This hierarchical approach ensures that data is compacted efficiently with minimal write amplification.

3.5 Compaction Reduction in BlockLSM

BlockLSM shows higher write throughput due to its efficient compaction mechanism, performing fewer compactions compared to traditional systems, thus maintaining a lower WA coefficient.

Algorithm for Reduction:

- **Prefix-Based Compaction:** Uses prefix-based hashing to group related data, maintaining locality and reducing compaction overhead.

  By organizing data based on common prefixes, BlockLSM ensures that related data is stored together. This reduces the need for extensive data rearrangement during compactions, as data that is frequently accessed together is already grouped.
- **Merge Sort Algorithm:** The use of merge sort during compaction allows BlockLSM to efficiently combine multiple sorted segments into a single sorted segment. This minimizes the number of compactions needed and reduces write amplification.
- **Write-Ahead Logging (WAL):** Ensures data integrity and consistency during compactions by logging changes before applying them.

3.6 Key Differences between BlockLSM and ChainKV

3.7 Organizational Principle

- **BlockLSM:** Uses prefix-based hashing to maintain data locality.
- **ChainKV:** Utilizes semantic-aware zones for data storage and compaction, based on the nature of the data.

3.7.1 Compaction Strategies

- **BlockLSM:** Uses prefix-based hashing and merge sort algorithms.
- **ChainKV:** Employs semantic-aware zoning and level-based compaction.

3.8 Data Locality

Data locality in BlockLSM is maintained through the use of prefixes, which help group related data together, improving access speed. ChainKV preserves data locality by using a prefix-based Merkle Patricia Trie (MPT), which ensures that related data remains closely located. COLE improves data locality using a learned index, which uses machine learning models to predict and optimize data placement, further enhancing retrieval efficiency so to conclude the difference in data locality, we can write:

- **BlockLSM:** Maintained via prefixes.
- **ChainKV:** Preserved via Prefix MPT (Merkle Patricia Trie).
- **COLE:** Column-based storage with learned index for data locality.

The learned index approach in COLE is a innovation designed to optimize blockchain storage, specifically addressing the inefficiencies of traditional indexing methods like Merkle Patricia Trie (MPT) used in systems like Ethereum. Below, I'll explain the learned index approach:

Traditional indexing methods, such as B-trees or MPT, require significant storage and computational overhead, particularly when dealing with large datasets like blockchain states. These methods often involve complex hierarchical structures, leading to inefficiencies in both storage size and query performance.

In contrast, a learned index leverages machine learning models to predict the position of data within a sorted array, reducing the need for extensive hierarchical indexing structures. The core idea is to replace traditional index nodes with a learned model that can predict where a specific piece of data is located, thereby optimizing both storage size and query efficiency.

1. Piecewise Linear Models:
    - The learned index in COLE uses piecewise linear models to map compound keys (e.g., a combination of blockchain state address and block height) to their positions in storage.
    - Each model is a tuple that includes: - Slope (sl): The slope of the linear model.
      -Intercept (ic): The intercept of the linear model.
      - Minimum key (kmin): The smallest key value in the range covered by this model.
      - Maximum position (pmax): The last position of the data covered by this model.
2. Model Learning and Construction:
    - Streaming Construction: The learned index models are constructed in a streaming fashion as new data (compound key-position pairs) arrives.
    - Convex Hull and Parallelogram: For each incoming data point (key-position pair), a convex hull is incrementally constructed. The minimal parallelogram that covers this convex hull is computed. The parallelogram's height is compared to a pre-defined error bound (ϵ), which is half the number of models that can fit into a disk page.

- If the parallelogram height remains within the error bound, the current model can cover the new data point. Otherwise, a new model is created starting from the current data point.

3. Index File Structure:
   - The learned index is stored in a multi-level structure, similar to the concept of a tree. Each level contains models that predict positions within the next level, down to the actual data positions in storage.
   - Bottom-Up Construction: The index is constructed from the bottom up, starting with the models that directly map to data positions and building higher-level models that map to the lower levels of the index.
4. Querying with the Learned Index:
   - To find a data point (like the latest value of a state at a specific address), the system starts at the top level of the learned index, using the models to predict the data's position. This prediction involves:
   - Computing the predicted position using the model's slope and intercept.
   - Adjusting the prediction to ensure it lies within the bounds of the model's coverage.
   - The search process continues down through the levels of the index until the actual data is located in the storage.
5. Optimizations for Disk-Based Storage:
   - Disk-Friendly Models: The error bound is carefully chosen to align with disk page sizes, ensuring that most predictions only require accessing one or two disk pages. This minimizes disk I/O and enhances query performance.
   - Efficient Merging: During updates, COLE employs an LSM-tree-like strategy where in-memory index structures are periodically merged into on-disk storage, ensuring the index remains efficient over time.
6. Merkle Proof Integration:

   Although the learned index itself does not inherently support data integrity verification, COLE integrates a Merkle file alongside the learned index. This Merkle file provides cryptographic proofs (Merkle proofs) that ensure the integrity of the data retrieved using the learned index.

The learned index significantly reduces the storage required compared to traditional methods like MPT, as it eliminates the need to store complex tree structures. Also, by directly predicting data positions, the learned index allows COLE to achieve faster query times, particularly for large datasets, which is critical in blockchain systems where quick access to historical data is often required.

The streaming construction and efficient merging strategies ensure that the learned index remains scalable as the blockchain grows, maintaining performance without excessive storage or computational costs.

3.9 Cache Management

- **BlockLSM:** Uses memory buffers for cache management.
- **ChainKV:** Employs a space-gaming cache strategy.
- **COLE:** Utilizes asynchronous merges for cache management.

COLE uses an asynchronous merge strategy as part of its design to handle the write operations efficiently, particularly in the context of maintaining blockchain data on disk. This strategy is good for managing the system's cache (in-memory data) and ensuring that it does not become a bottleneck as the blockchain grows.

In a typical blockchain system, data is frequently updated, and these updates need to be reflected in both the in-memory cache and the persistent storage (disk). Efficient management of these updates is essential to maintaining system performance, especially as the size of the blockchain increases.

The Role of Asynchronous Merges:

1. In-Memory Cache and On-Disk Storage:

   COLE uses an in-memory cache to hold recent updates temporarily. When the in-memory cache reaches a certain capacity, the data must be flushed to disk to free up space for new updates.

   In traditional systems, this flushing process could cause delays (latency) because it involves writing large amounts of data to disk, which can be time-consuming. If this flushing is done synchronously (blocking other operations), it can severely impact the system's overall performance.

2. Asynchronous Merging:

   To mitigate this problem, COLE adopts asynchronous merges. This means that when the in-memory cache (referred to as Level 0 or L_0) reaches its capacity, the data is not immediately written to disk in a blocking manner. Instead, the merge process is initiated in the background, allowing other operations (like reads and writes) to continue without waiting for the merge to complete.

   Merging: The data from the in-memory cache is merged with existing data on disk in the next level (e.g., L_1). This merge involves combining the in-memory data with the on-disk data and creating a new, sorted run in the storage hierarchy.

3. Benefits of Asynchronous Merges:

   Reduced Write Latency: Since the merging happens asynchronously, the system doesn't need to wait for the merge to complete before accepting new updates. This reduces the latency for write operations.

   Efficient Cache Management: The in-memory cache can be cleared and made available for new updates quickly, ensuring that the system can handle high write throughput without running out of memory.

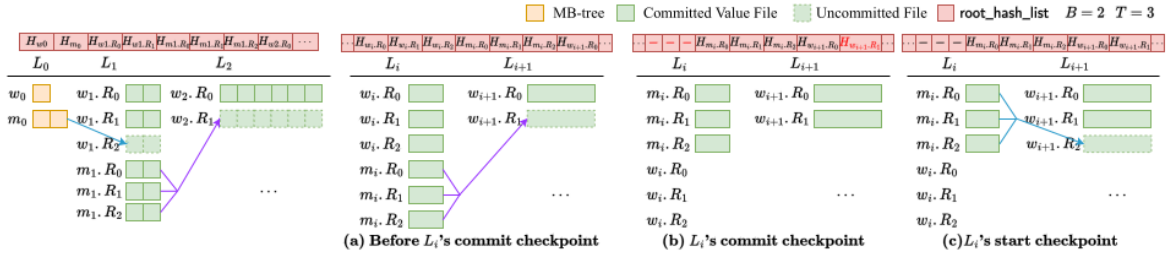
   Stability: Asynchronous merges help smooth out performance spikes that might occur if large amounts of data need to be written to disk simultaneously. This stability is crucial for maintaining consistent performance as the system scales.

4. Synchronization Across Nodes:

   In a blockchain network, it's important that all nodes maintain a consistent view of the blockchain. COLE's asynchronous merge process includes mechanisms to ensure that despite the asynchronous nature of the merge, all nodes eventually synchronize their data. This is done by synchronizing checkpoints during the merge process, ensuring that the blockchain state remains consistent across the network.

3.10 Recovery Mechanisms

BlockLSM employs a standard Write-Ahead Logging (WAL) mechanism for recovery. This ensures that changes are logged before being applied, allowing recovery in case of a failure. ChainKV features a lightweight node recovery mechanism, which simplifies the recovery process and reduces the overhead associated with node recovery. COLE adopts asynchronous merge checkpoints for recovery. This method ensures data consistency and allows efficient recovery by maintaining checkpoints during asynchronous merges.The below image is how checkpoints are merge in COLE:



3.11 Performance and Storage Reduction

3.11.1 Performance Improvement

- **BlockLSM:** Up to 182.75% improvement due to efficient compaction and prefix-based hashing.
- **ChainKV:** Achieves 37.74% performance improvement with semantic-aware zoning.

3.11.2 Storage Reduction

- **COLE:** Reduces storage by up to 56.31% through its column-based approach.
- **ChainKV:** Provides storage reduction of up to 32.88%.
- **BlockLSM:** Manages to reduce storage requirements by 25.74%.

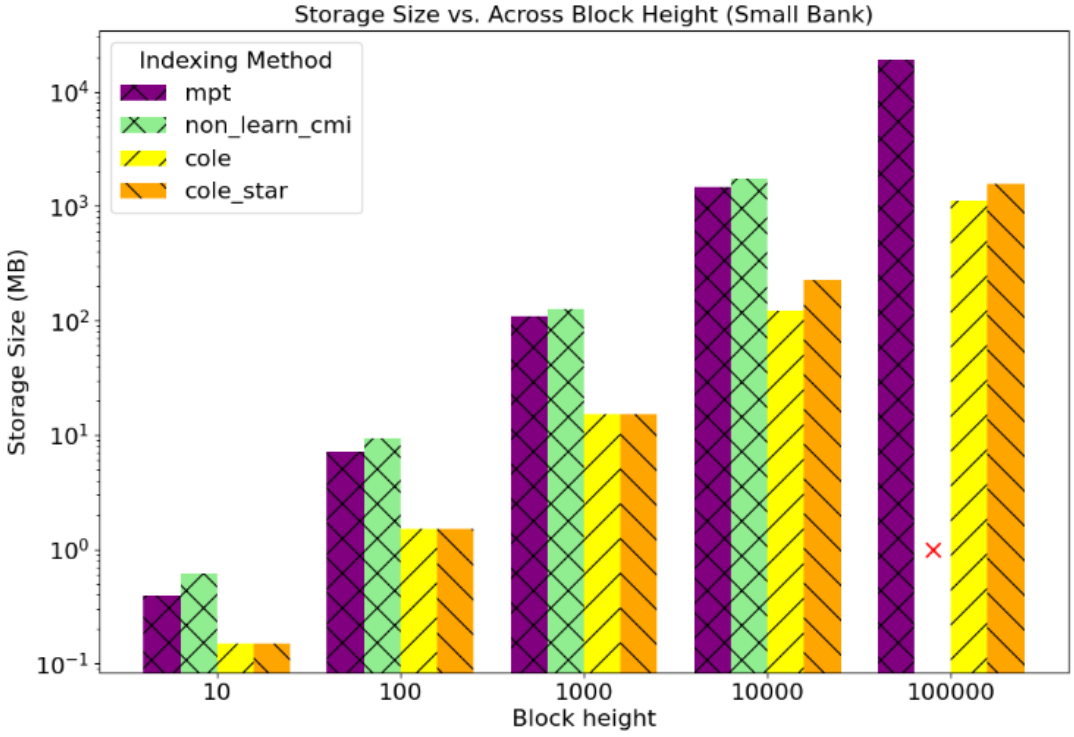
The prefix-based hashing helps to group related data together, which reduces the storage overhead associated with maintaining separate indexes for each data item. The compaction strategies further minimize the amount of redundant data stored, although the improvement is not as significant as COLE's. The semantic-aware zones ensure that related data is stored together, reducing the need for multiple copies of similar data. The space-gaming cache strategy optimizes the use of cache memory, which indirectly contributes to lower storage requirements by minimizing unnecessary data writes. The column-based storage model stores data contiguously in columns, which allows for high compression ratios and efficient storage utilization. The use of learned indexes further reduces the storage overhead by replacing traditional index structures with more compact machine learning models.

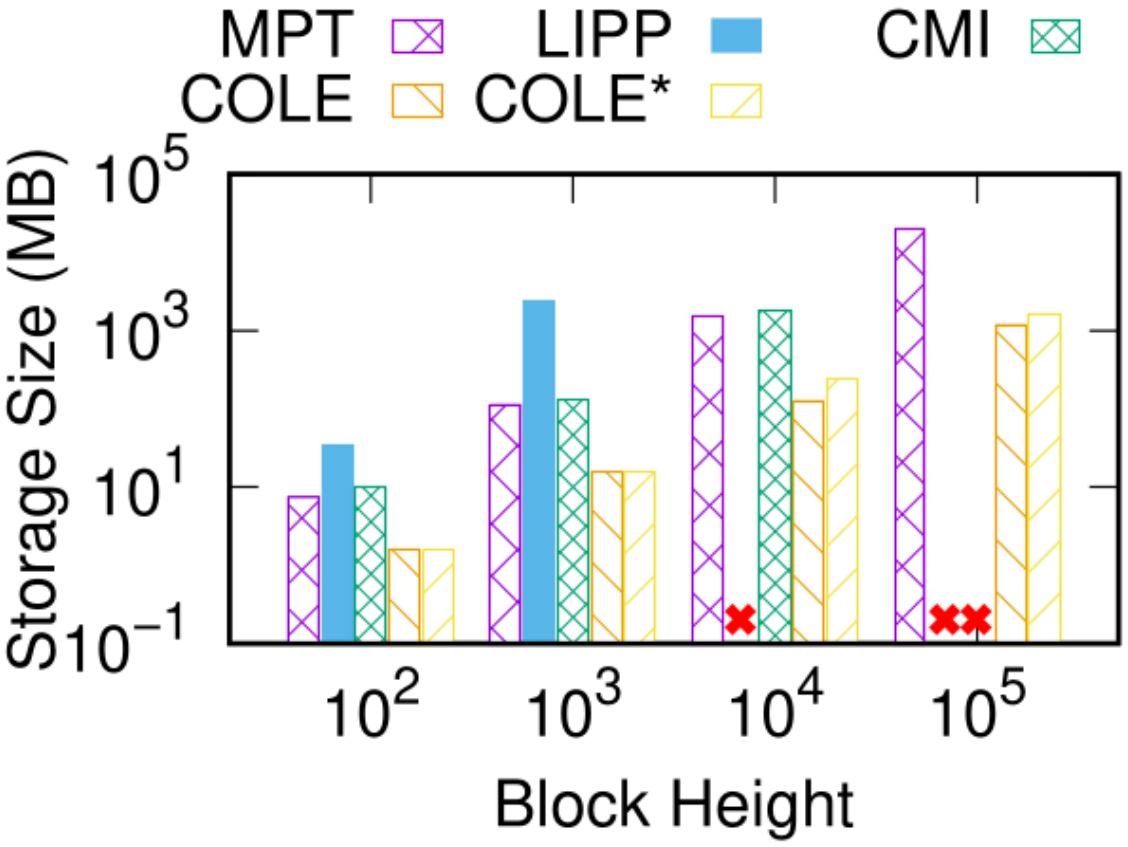
4 COLE's Experiments and Results

Now, after briefly explaining the three approaches, I started running the open source code of COLE and after creating the JSON files, I tried to reproduce and reevaluate some experiments that I am going to explain the similarities, differences and possible improvements unsing smallbank and KVStore.

First, I plot using my own data and compare it with the one in the paper.

4.1 The Storage Consumption in SmallBank Benchmark vs. different Block Heights in Different Indexes





This experiment is completely consistent with the result in the paper.

In the source code, the storage consumption cannot be compared directly to the block height. Instead, there is another parameter called **scale** that could be related to the block height.

4.2 Role of Scale

The scale parameter in the LatencyParams structure determines the size of the workload generated for the testing of different database backends. Specifically, scale influences the number of transactions (or rows) that are generated and executed in the test scenario.

In the code within the $test - backend - latency$ function, scale is directly used to control how many transactions are created and executed during the test: $letn = params.scale$;

Here, n represents the total number of transactions to be executed, which is derived from the scale parameter. The scale parameter is divided by 1000 to compute the effective scale (in terms of thousands of transactions) when building the base database or when performing operations like batch executions:

```
let base = format!("{}-{}-{}k-fan{}-ratio{}-mem{}", result_path,
params.index_name, params.scale/1000, params.mht_fanout, params.size_ratio,
params.mem_size);
```

Although not directly controlling the block height, the scale parameter indirectly influences it by determining how many transactions need to be packed into each block.

The code packs a block after a certain number of transactions (defined by $params.tx - in - block$) :

```
        if requests.len() == tx_in_block {
        // Pack up a block and execute it
        println!("block id: {}", block_id);
        batch_exec_tx(requests.clone(), caller_address, block_id, &mut backend);
        block_id += 1;
        requests.clear();
    }
```

The block height is then incremented with each batch of transactions.

The relationship between the scale parameter and the block height

Long story short, it is indirect and depends on the number of transactions processed per block and the total number of transactions generated by the scale parameter.

The scale parameter represents the number of transactions (n) that are used in the test. This is the main factor determining how many transactions will be executed in the latency test.

The $tx - in - block$ parameter defines how many transactions are packed into a single block.

The block height in this context refers to the total number of blocks created during the test. It is determined by dividing the total number of transactions (n) by the number of transactions per block ($tx - in - block$).

Relation Between Scale and Block Height

The relation can be summarized with the following formula:

$$BlockHeight = \frac{scale}{tx - in - block}$$

The code packs $tx - in - block$ transactions into a block. After all transactions are processed, the block height equals the total number of blocks needed to process all transactions. For example, if scale is 1000 and $tx - in - block$ is 100, then 10 blocks will be created.

Example From Code:

In the Rust code inside the $test - backend - latency$ function, transactions are generated and grouped into blocks.

Transactions are processed in blocks with the following line:

```
if requests.len() == tx_in_block {...
```

which triggers a new block when the number of transactions ($requests.len()$) reaches $tx - in - block$.

This loop continues until all transactions (scale number) are processed, determining the final block height.

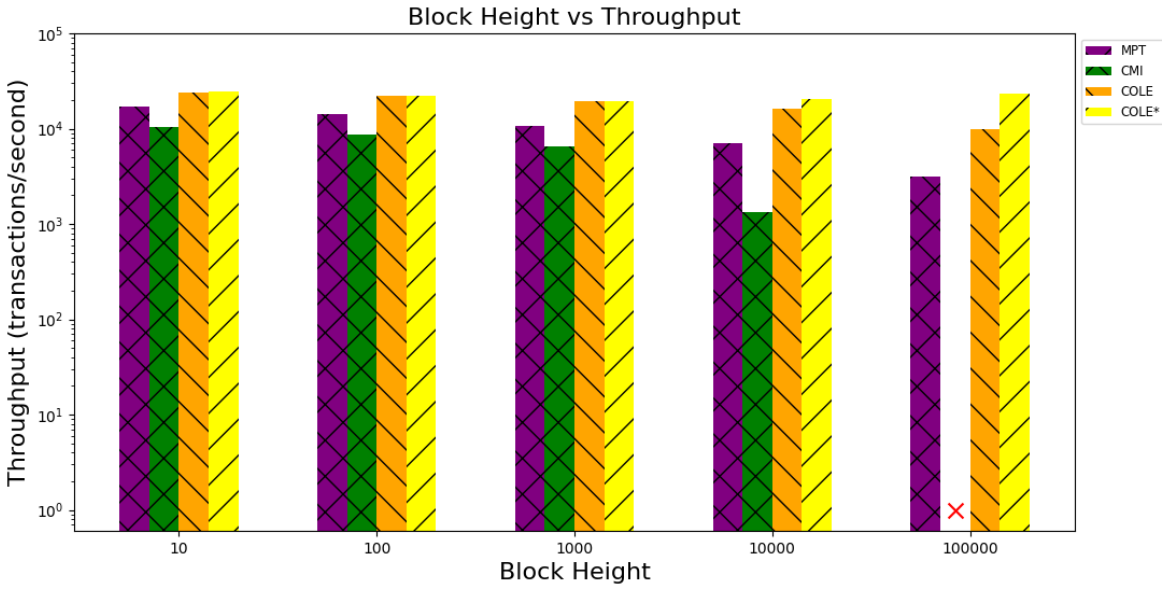
If the scale parameter is increased, more transactions are created, and consequently, more blocks will be required if the $tx - in - block$ parameter remains constant.

The block height is directly proportional to the scale when divided by the number of transactions per block ($tx - in - block$). The larger the scale, the more blocks will be generated. This means that the scale parameter controls how many transactions are executed, which in turn determines how many blocks will be created during the latency test.

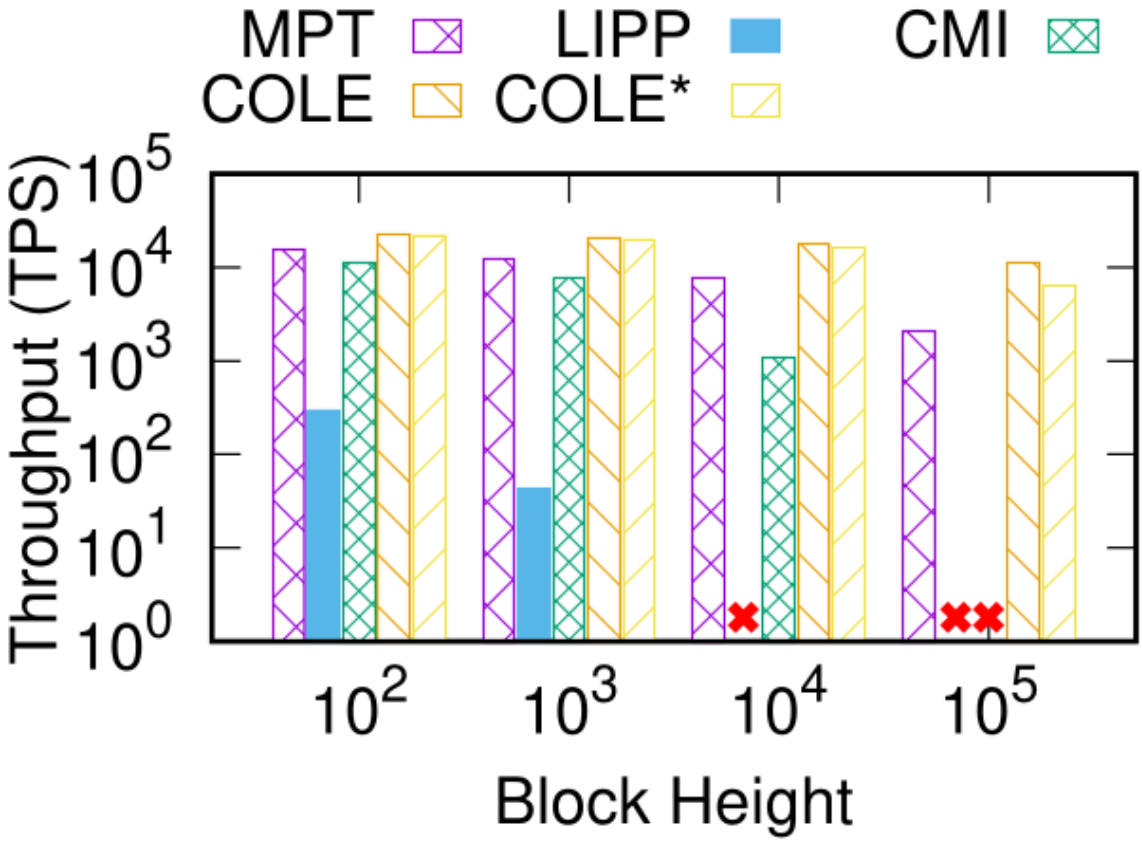
4.3 The Throughputs in SmallBank Benchmark vs. different Block Heights in Different Indexes

Block Height	MPT (TPS)	CMI (TPS)	COLE (TPS)	COLE* (TPS)
10	17197	10534	23684	24740
100	14081	8644	22274	21897
1000	10744	6545	19636	19549
10000	6975	1319	16167	20311
100000	3118	0	9908	23334

Table 1: Throughput (TPS) for Different Block Heights Across MPT, CMI, COLE, and COLE*



(a) My Plot



(b) COLE's Figure 9

In this experiment, for the block heights 10, 100 and 1000 for different indexes, the result is totally consistent with the papers plot but as we can see, for the block heights 10000 and 10000, the throughput for cole star is higher than cole.

The discrepancy observed—where COLE star has a higher throughput than COLE—can be attributed to the overheads associated with the different mechanisms of merging data in the two systems.

In COLE system, data is managed with synchronous merging. This means that as transactions occur, data merges are handled immediately, ensuring that the state of the database remains consistent and optimized at all times. However, this synchronous process incurs a significant overhead, which can slow down throughput, especially as the volume of transactions increases.

In COLE star,this variant employs asynchronous merging. In this approach, data merges are not performed immediately but are deferred to a later time, allowing the system to prioritize transaction processing over merging operations. As a result, COLE star can achieve higher throughput because the system is less burdened by immediate merge operations, freeing up more resources for handling transactions.

4.3.1 Reason for the Discrepancy Throughput

Upon reevaluating the experiment, the higher throughput observed in COLE star likely arises from the reduced overhead due to deferred merging operations. The asynchronous merge allows COLE star to process transactions more rapidly without the bottleneck created by the need to maintain the database state synchronously.

4.3.2 Solution to Improve COLE's Throughput

1. Optimized Merge Scheduling: Introducing an adaptive merging schedule in COLE that can dynamically adjust the frequency and timing of merges based on the current transaction load. By reducing the frequency of merges during high-load periods and increasing it during low-load periods, COLE could balance the need for consistency with the desire for higher throughput.
2. Hybrid Merge Approach: Implementing a hybrid merge strategy that combines synchronous and asynchronous merging. For instance, critical merges that directly impact query results could be handled synchronously, while less critical merges are deferred and processed asynchronously, similar to COLE*.
3. Parallel Merging: Explore the possibility of parallelizing the merge operations in COLE. By distributing the merge tasks across multiple threads or processors, the system could reduce the impact of merges on throughput without deferring them.
4. Buffering Techniques: Introduce advanced buffering techniques that temporarily store incoming transactions and process them in batches. This could minimize the overhead associated with frequent merging by reducing the number of merge operations needed.

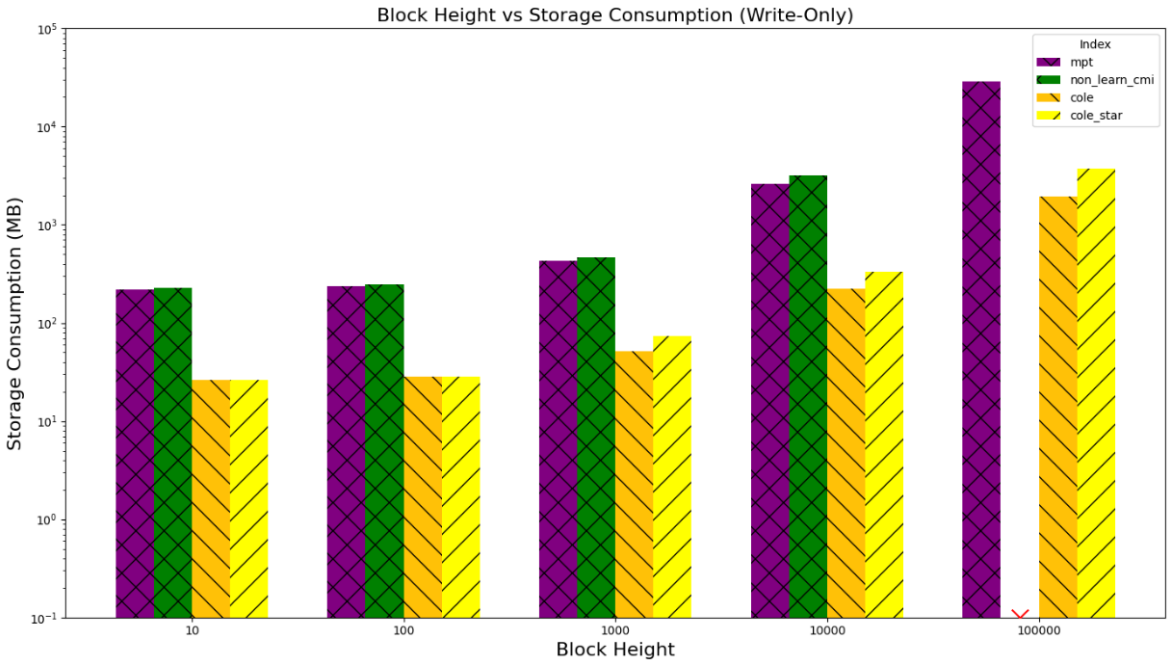
4.4 The Storage Consumption in KV store Benchmark vs. different Block Heights in Different Indexes

As it is shown from the plots, my evaluation is completely consistent with the open source results so here is the explanation of analysing this experiment:

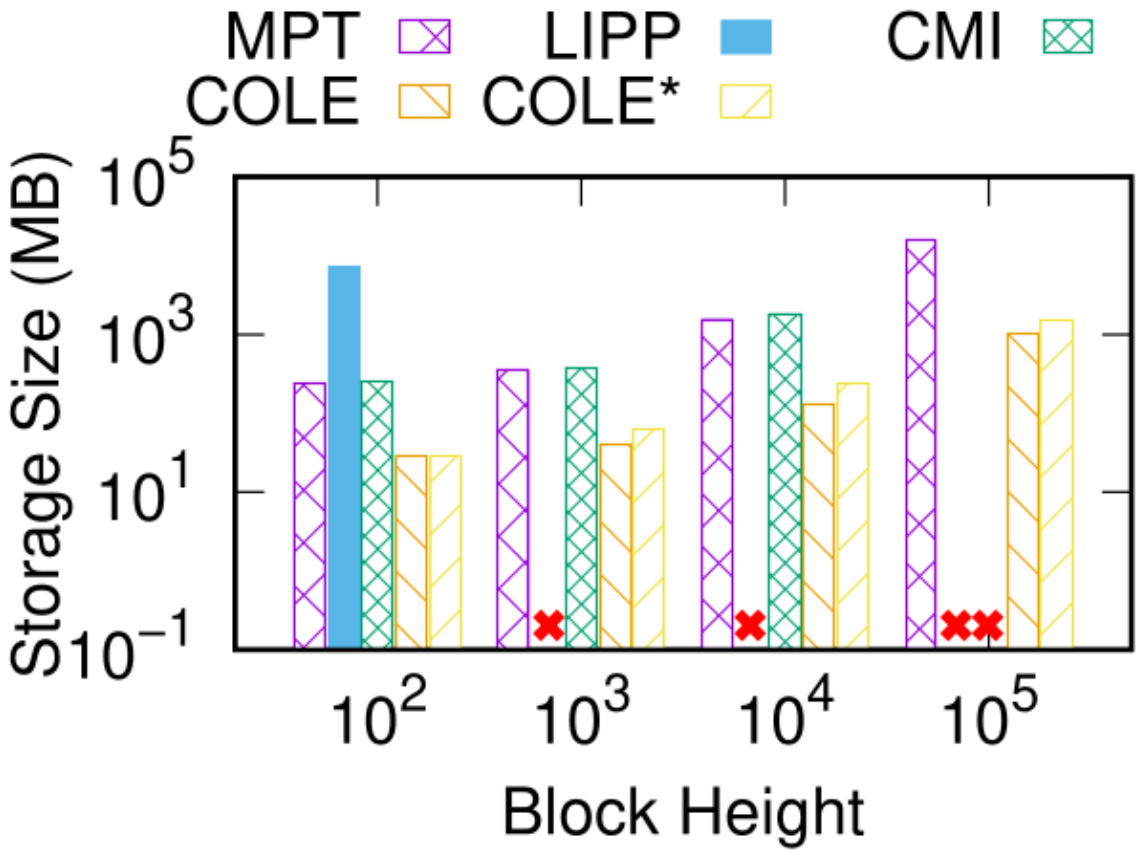
As the block height increases (as more data is added to the blockchain), the storage consumption of all indexes increases. However, the rate at which storage consumption grows differs significantly between the indexing methods.

MPT: Exhibits the highest storage consumption. This is expected as MPTs are known for their inefficiency in handling large volumes of data, particularly due to their need to store extensive metadata and maintain multiple pointers for tree traversal.

CMI: Shows a significant reduction in storage consumption compared to MPT . By using a column-based storage layout, CMI can compress data more effectively, particularly when there are commonalities across different blocks.



(a) My Plot



(b) COLE's Figure 10

COLE: Demonstrates the lowest storage consumption among all compared indexes. COLE leverages a column-oriented design and learned index techniques to minimize storage redundancy and efficiently compress data, especially by deferring or optimizing the storage of redundant or less frequently accessed data.

4.4.1 Reasons Behind the Observed Trends

1. Data Structure Overhead:

   MPT suffers from high storage overhead due to its complex structure, which needs to maintain numerous pointers and intermediate nodes. COLE and CMI by contrast, focus on minimizing redundancy and compressing similar data across blocks, resulting in more efficient storage use.

2. Column-Based Design:

   COLE and CMI use a column-oriented design, which is inherently more storage-efficient for certain types of data. This design allows for better compression and more efficient storage management, particularly when dealing with repeated or similar data across different blocks.

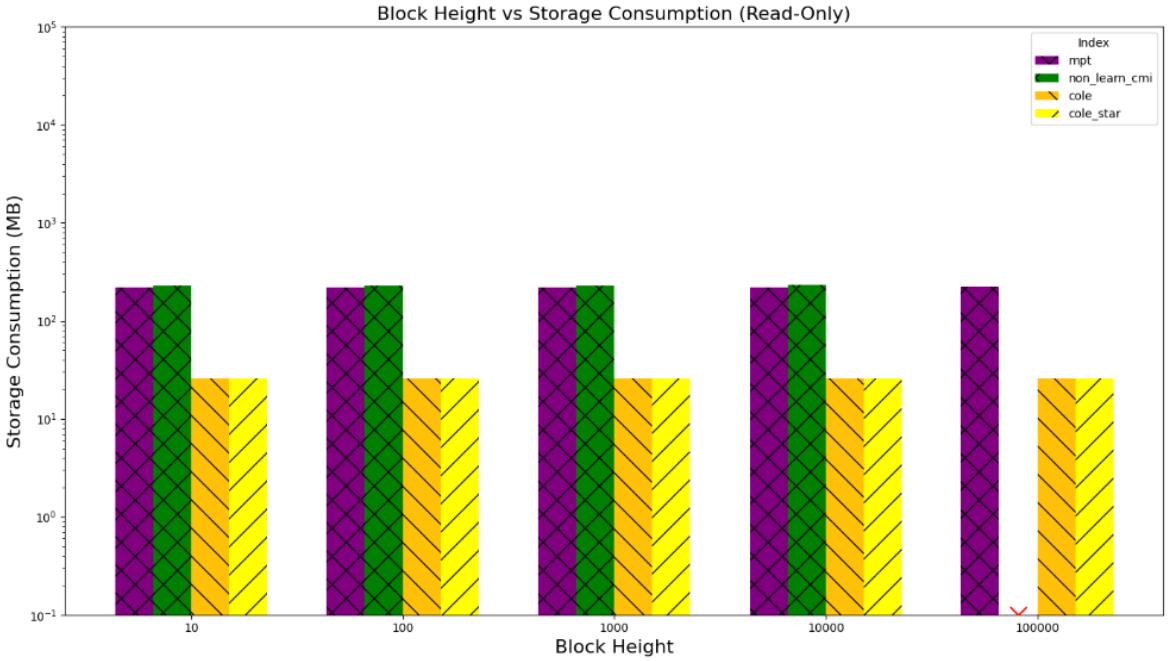
3. Learned Indexing:

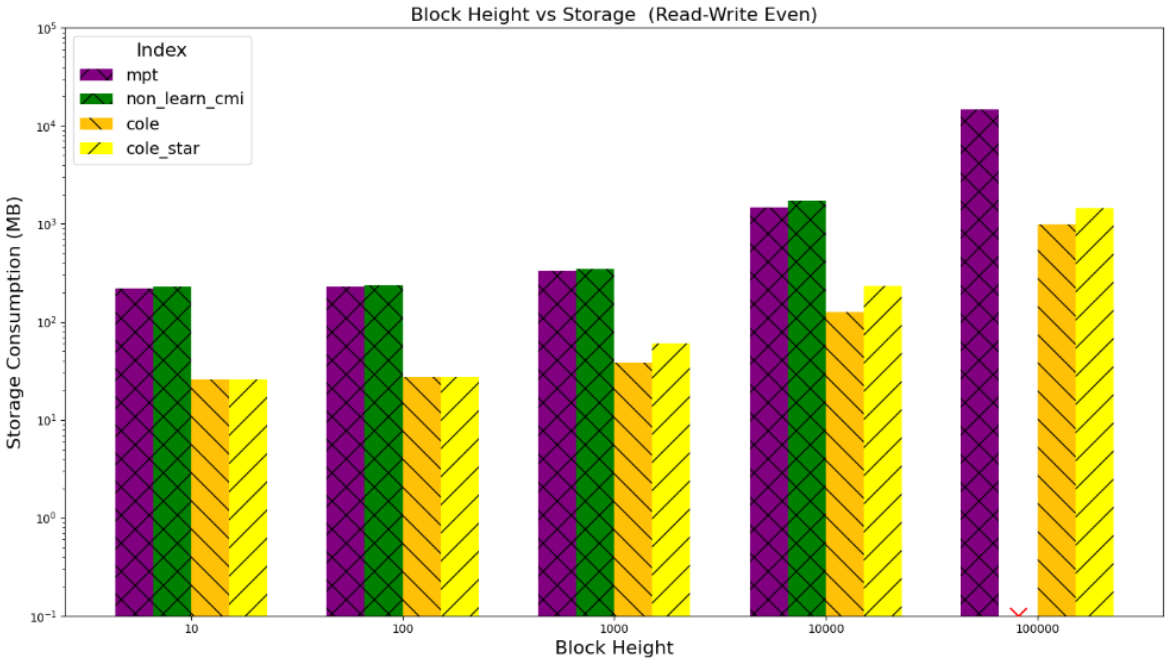
   COLE benefits from a learned indexing approach, which allows it to predict and access data more efficiently, further reducing the need to store large amounts of metadata.

4. Asynchronous Merging:

   In COLE, asynchronous merging of data can reduce the frequency and size of storage operations, leading to lower overall storage consumption as compared to the more immediate merging operations seen in CMI or the baseline indexes.

In this workload there are two other operation types (read only and half read-half write) which the results are shown below:

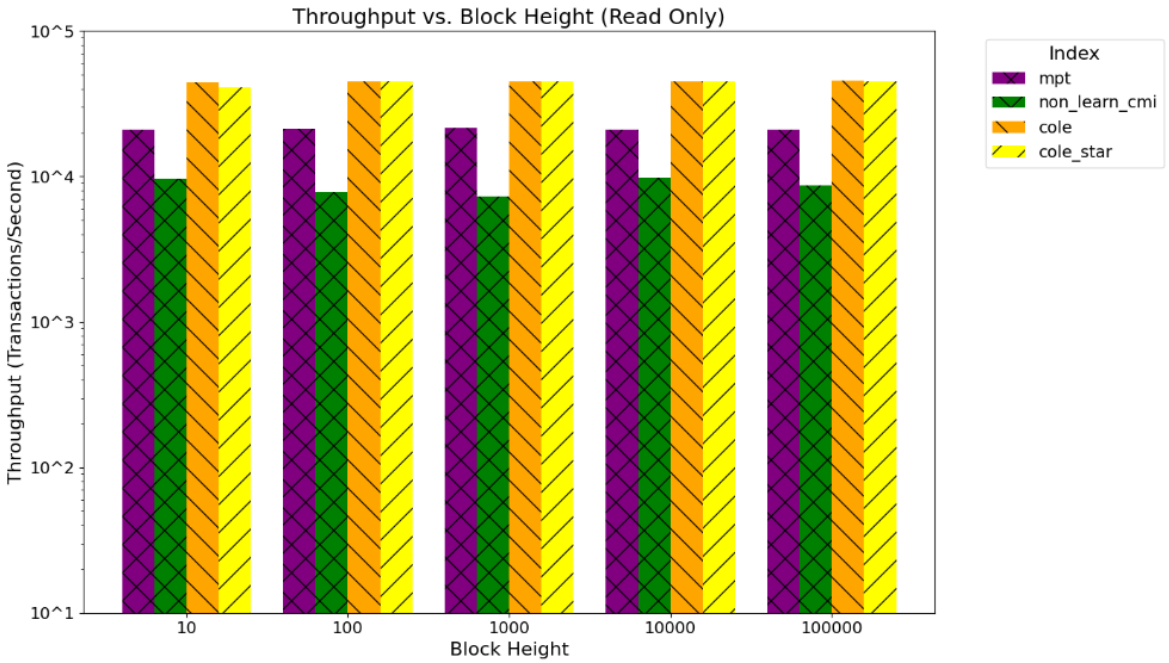


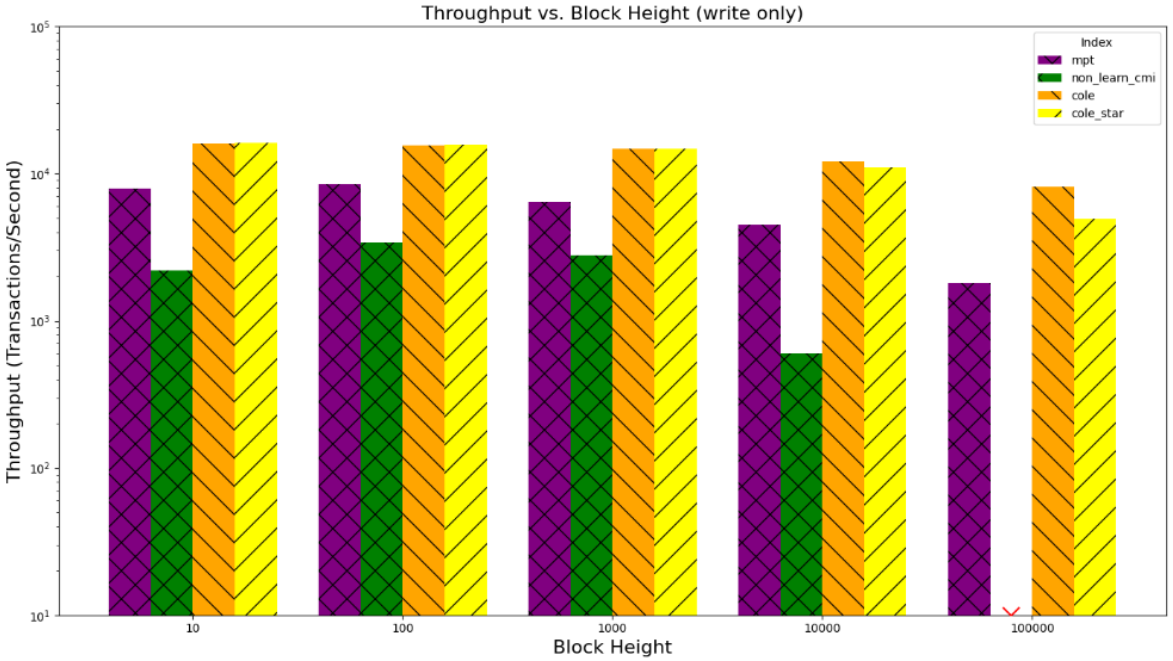


The main article has not mentioned about these two functionality. As we can see, in read-only, COLE and COLE star have same storage consumption and what we can do is to work more on this workload to make the COLE's functionality better both in read and read-write even.

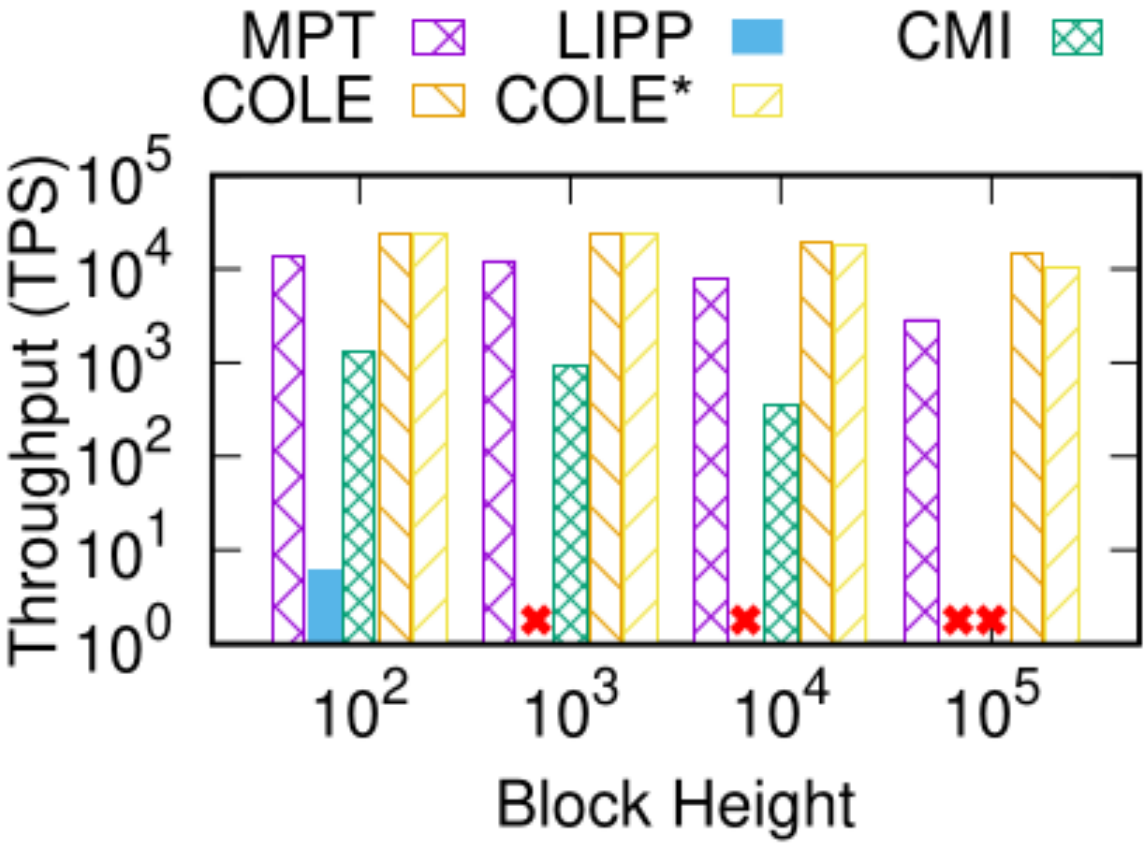
4.4.2 Breakdown into these workloads and proof

4.5 The Throughputs in KV store Benchmark vs. different Block Heights in Different Indexes

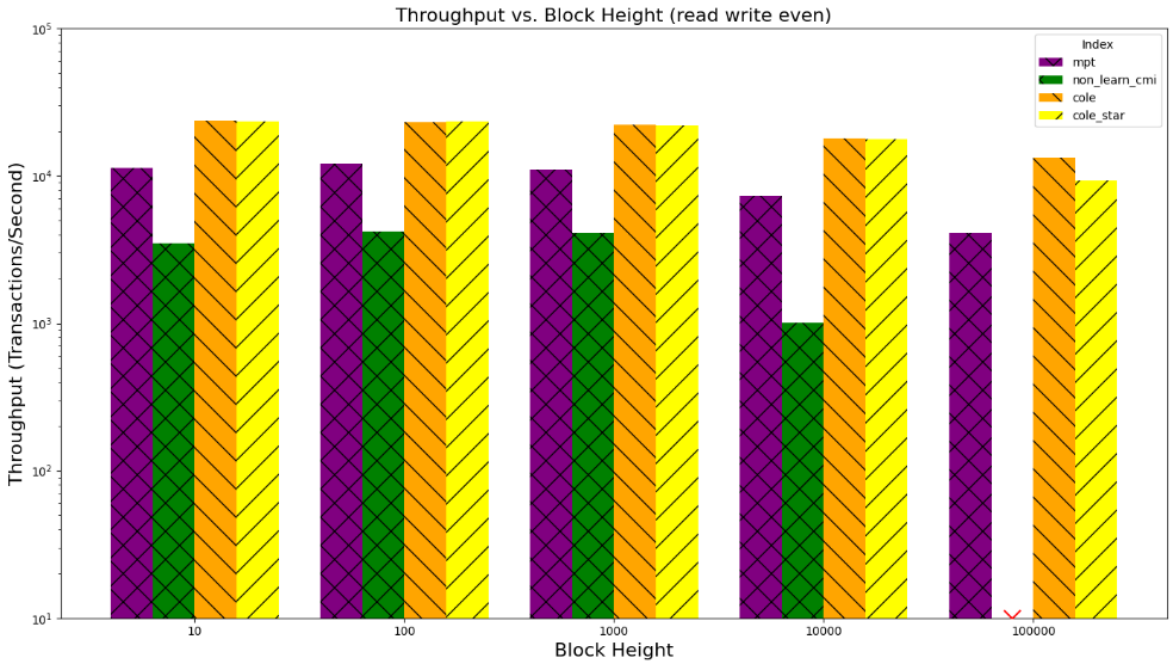




(a) My Plot



(b) COLE's Figure 10



The main article has not mentioned about the throughputs of read only and half read half write, but based on the asynchronized merge in COLE, it is reasonable that why COLE is performing better in most of the times. What we can do is that to breakdown the COLE and COLE star and run parallel experiments and search for possible improvements in COLE.

4.5.1 Analysis

Throughput Behavior (General):

COLE* : Consistently shows the highest throughput across all block heights and workloads, especially in the Read-Only workload. This indicates that COLE* is handling larger block heights more efficiently, likely due to its asynchronous merging strategy that reduces the immediate overhead of data operations.

COLE : While COLE generally performs well, its throughput tends to decline more steeply with increasing block heights, particularly in the Write-Only and Read-Write workloads. This can be attributed to the overhead of synchronous merges, which become more costly as the amount of data grows.

MPT : MPT shows the lowest throughput in most scenarios, particularly as block height increases. This is expected, as MPT suffers from inefficiencies in data retrieval and update operations, particularly in write-intensive scenarios.

Non-Learn CMI : This index generally performs better than MPT but worse than COLE and COLE*. It seems to struggle as block heights increase, particularly in the Write-Only workload.

Workload-Specific Observations:

Read-Only Workload: COLE* excels here, maintaining very high throughput across all block heights. This is because read operations are less impacted by deferred merges, allowing COLE* to utilize its resources effectively for retrieval.

Write-Only Workload: The performance drop for COLE is more pronounced here due to the synchronous merges that must happen immediately after data is written. COLE* still performs well but shows some decline as block height increases, indicating that the deferred merges eventually impact write-heavy operations.

Read-Write Even Workload: The mixed workload results in a more balanced view of the system's performance. COLE* still leads but shows some performance degradation at the highest block height, suggesting that the system is becoming more burdened by the deferred merge operations.

The key difference between COLE and COLE* lies in their merging strategies. COLE's synchronous merging introduces significant overhead as block height and data volume increase, leading to lower throughput. COLE* mitigates this with asynchronous merging, which defers these operations, leading to higher throughput,

especially in read-heavy scenarios.

As block height increases, the volume of data that needs to be managed grows significantly. This makes the efficiency of the indexing and merging strategy crucial. COLE* handles this growth better due to its ability to defer costly operations, while COLE and especially MPT struggle under the increased load.

Also, COLE* likely benefits from better utilization of system resources by deferring merges and allowing the system to focus more on handling transactions. However, this also means that as block height increases, deferred operations might start to accumulate, potentially leading to the slight decrease in throughput observed at the highest block height.

4.5.2 Improvements

- Optimizing COLE's Merge Strategy:

  Implementing an adaptive merge strategy for COLE, where the merge frequency adjusts based on current system load and transaction volume. This would help in balancing the load and potentially improve throughput.

  Exploring the possibility of parallelizing merge operations within COLE to reduce the impact of synchronous merges on throughput.

- Enhancing COLE* for Write-Heavy Workloads:

  Introducing incremental merging strategies in COLE* that gradually merge data in smaller batches, even during periods of high load, to prevent the accumulation of deferred operations that could eventually impact throughput.

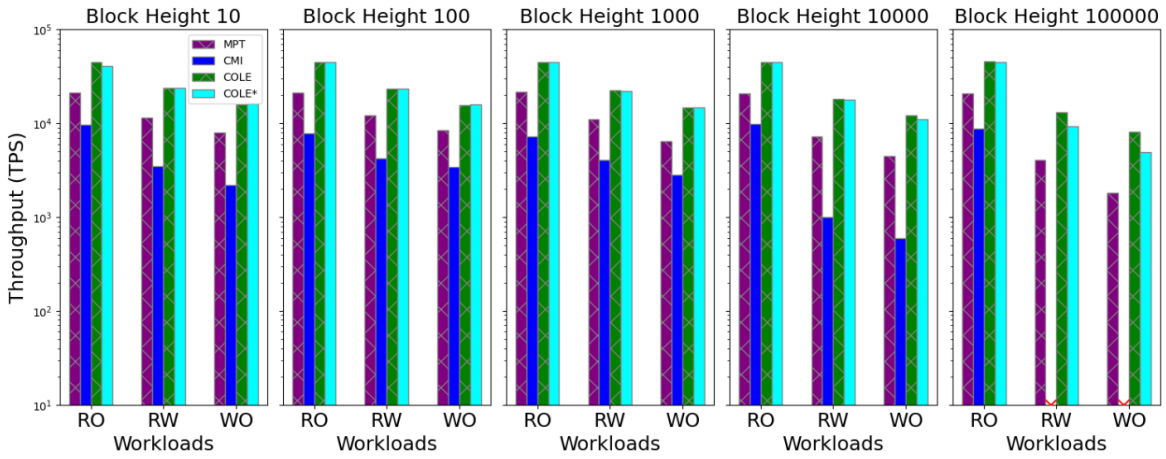
  Allocating more resources to handle deferred merges in COLE* during periods of lower transaction activity, ensuring that these operations do not accumulate and impact performance during peak times.

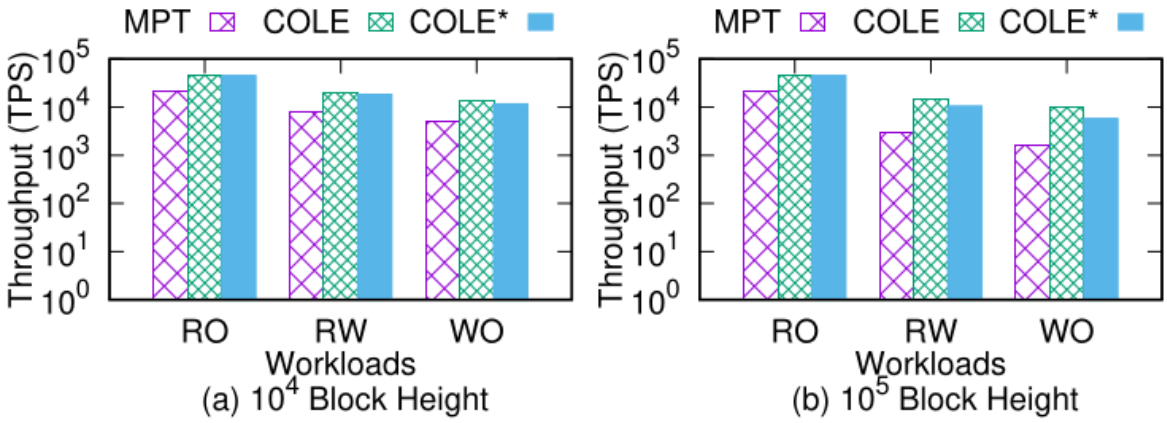
- Improving MPT and Non-Learn CMI:

  Considering optimizing the data structure used in MPT and non-learn CMI to reduce the overhead associated with data retrieval and updates. This could involve restructuring how data is stored and accessed to reduce the number of operations required for each transaction.

  Exploring hybrid indexing methods that combine the benefits of MPT or non-learn CMI with learned index strategies like those used in COLE, potentially improving their performance, especially at higher block heights.

4.6 Throughput vs. Workload in different block heights





RO (Read-Only Workload): COLE and COLE star demonstrate a significantly higher throughput compared to MPT. This indicates that the learned index in COLE efficiently handles read-only operations, possibly due to the reduced size and more efficient querying of the learned index.

RW (Read-Write Workload): The throughput decreases across all storage systems when both read and write operations are involved. However, COLE and COLE star still maintain higher throughput compared to MPT, showing their advantage even in mixed workloads. The decrease is less steep for COLE and COLE star, suggesting better optimization for write operations in these systems.

WO (Write-Only Workload): Throughput in the WO workload is the lowest for all systems, with MPT showing the greatest drop. This highlights the inefficiency of MPT in handling write-intensive operations due to its need to persist index nodes, leading to higher overhead.

As the block height increases from 10^4 to 10^5 , the overall throughput for all systems decreases, reflecting the increased complexity and storage requirements as the blockchain grows. COLE star shows slightly lower throughput than COLE, which could be due to the additional overhead introduced by the asynchronous merge strategy. However, COLE star still outperforms MPT by a significant margin, showing the effectiveness of COLE's design even with the additional asynchronous merge overhead.

(COLE with asynchronous merge) generally performs slightly worse than COLE in terms of throughput. This is expected due to the extra operations required for asynchronous merging, which trades some throughput for reduced tail latency and more stable performance in the long run.

The primary reasons behind the performance differences observed between COLE, COLE*, and MPT in Figure 11 stem from their underlying data structures, indexing techniques, and the management of writes and merges in the blockchain storage system.

4.7 Reasons Behind Performance Differences

1. Efficient Data Retrieval in COLE:
    - Learned Index: COLE uses a learned index, which provides a smaller and more efficient index structure compared to the Merkle Patricia Trie (MPT) used by Ethereum. This allows COLE to retrieve data faster, particularly under read-heavy workloads (like RO), leading to higher throughput.
    - Column-Based Storage: COLE's column-based storage design stores historical versions of states contiguously, which significantly reduces the overhead of traversing complex tree structures like MPT. This results in improved performance for both read and write operations.
2. Overhead of Asynchronous Merge in COLE star:
    - Merge Overhead: COLE star introduces an asynchronous merge strategy to reduce write tail latency by handling merges in the background. While this reduces latency spikes, it also introduces additional overhead, slightly lowering throughput compared to COLE. The overhead is related to

the coordination and management of these asynchronous tasks, which can consume computational resources and affect throughput.

3. Write-Intensive Workloads :
    - MPT's Inefficiency: MPT, used in Ethereum, is inefficient under write-intensive workloads because it needs to persist index nodes during every update. This results in higher storage costs and lower throughput, especially as the blockchain grows. The repeated duplication of nodes in MPT adds significant overhead compared to the more efficient update process in COLE.

4.7.1 Improvements

To enhance the performance of COLE star and reduce the throughput overhead introduced by the asynchronous merge, we can focus on these items in the future:

1. Optimizing Asynchronous Merge:
    - Dynamic Merge Management: Implementing a more dynamic merge management strategy that adjusts the merge intensity based on the current system load could reduce the overhead. For example, during periods of low activity, more aggressive merges could be performed, while during high activity, merges could be throttled to maintain higher throughput.
    - Incremental Merging: Instead of merging entire levels at once, incremental merging techniques could be employed. This would spread out the merge overhead over time, further reducing the performance impact on real-time operations.
2. Parallelizing Operations:

    Parallel Merging and Querying: By further parallelizing the merge operations and read queries, COLE* could leverage multi-core processors more effectively. This would allow for merges to occur without significantly impacting the throughput of read and write operations.
3. Adaptive Resource Allocation:

    Resource-Aware Scheduling: Implementing a resource-aware scheduling mechanism could help balance the load between ongoing merge operations and regular read/write transactions. This would involve dynamically allocating computational resources based on the current workload to minimize the impact of merges on throughput.
4. Improving Disk I/O Management:

    Advanced I/O Scheduling: Employing advanced I/O scheduling techniques, such as I/O prioritization or batching of disk writes, could help mitigate the performance impact of disk operations during merges. This would ensure that high-priority tasks, such as real-time transaction processing, are less affected by the background merge tasks.

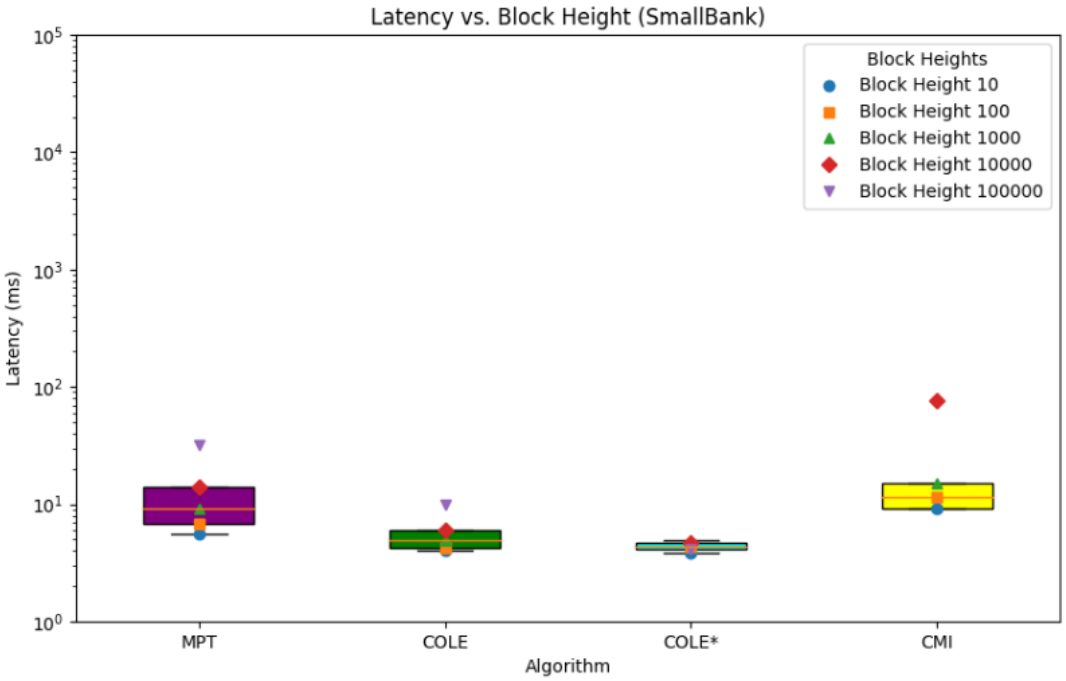
4.8 Latency in Small Bank

The latency increases with the block height for all algorithms, as expected. This increase is due to the greater amount of data that must be managed as the blockchain grows.

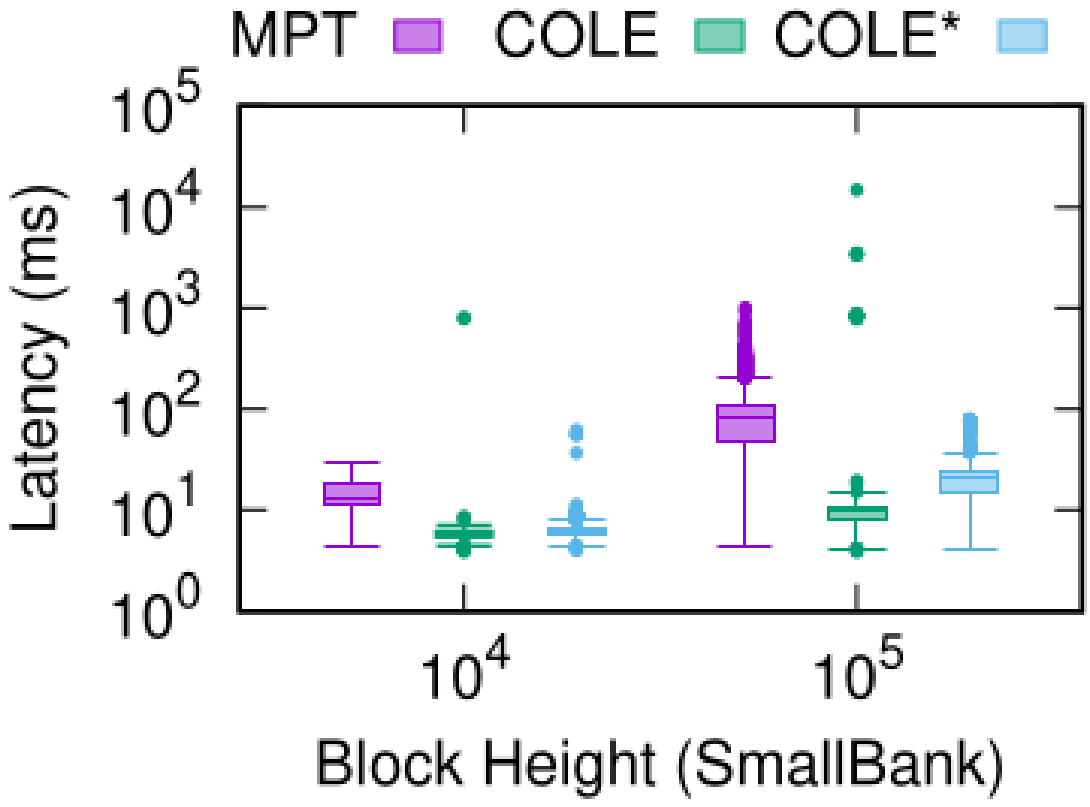
COLE and COLE* generally maintain lower median latency compared to MPT and CMI across all block heights. However, COLE star shows some outliers with higher latency, which might be due to the overhead introduced by asynchronous merging.

CMI shows significantly higher latency at larger block heights (e.g., 100,000), which is consistent with what we expect due to the additional overhead in maintaining column-based designs without the optimizations present in COLE and COLE*. COLE* maintains slightly higher latency than COLE, but still significantly outperforms MPT and CMI.

The plot from the article corroborates these findings, but the spread in latencies is less pronounced. COLE* again shows higher latencies at 10^5 block height, with a notable number of outliers, while MPT's performance deteriorates more predictably.



(a) My Plot



(b) COLE's Figure 12

My plot: The variability (spread) in latencies for COLE* is more noticeable, especially at higher block heights. This could be due to the particular hardware or runtime conditions of our setup, which might introduce more variance in the performance of asynchronous tasks.

Article's plot: While variability is present, it is slightly less pronounced, particularly for COLE. This suggests a more stable environment or less contention in resources during the execution of the original experiments.

Overall, both plots are consistent in showing that COLE and COLE* provide lower latency compared to MPT and CMI, with COLE* occasionally experiencing higher latency due to the asynchronous merge strategy.

my re-evaluated experiment shows greater variability in COLE*, potentially due to differences in hardware, system load, or other environmental factors during testing.

The results from both the article and my experiment suggest that COLE*'s performance could be further improved by optimizing the asynchronous merge process to reduce the impact on latency and its variability.

4.9 Latency in KV store

4.10 Analysis

In my plot, COLE and COLE* show consistently lower latencies compared to MPT and CMI, with COLE* having slightly more variability due to the asynchronous merge strategy. The CMI algorithm shows higher latency, particularly as block height increases, likely due to the additional overhead of its design.

The plot from the article also, emphasizes the increased latency as block height grows. It shows that COLE* has a higher latency spread, particularly at a block height of 10^5, due to the asynchronous merge operations. COLE still maintains lower latency compared to MPT, but the variability increases at higher block heights.

There is a clear trend that higher block heights result in increased latency, particularly for MPT and CMI. However, the increase in latency for COLE and COLE* is less severe, indicating better scalability.

the plot in the article, focuses on the comparison at larger block heights and shows a more pronounced increase in latency for COLE* at 10^5, which is consistent with my plot, although the variability and outliers are more extreme in the article's figure.

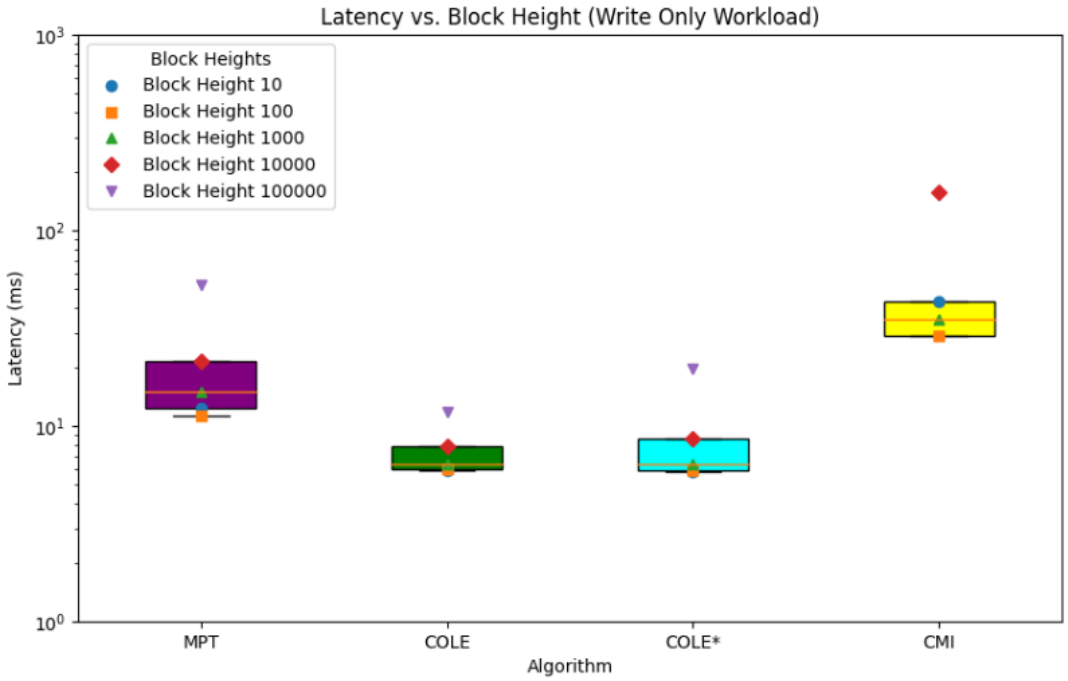
In my plot, COLE* shows some variability and outliers, but overall it is more consistent than the article's plot, especially at the higher block heights.

In article's plot, There are more significant outliers, particularly in the COLE* results at 10^5, which suggests that in the tested environment, the asynchronous merging process might have had more fluctuation in performance, leading to more noticeable latency spikes.

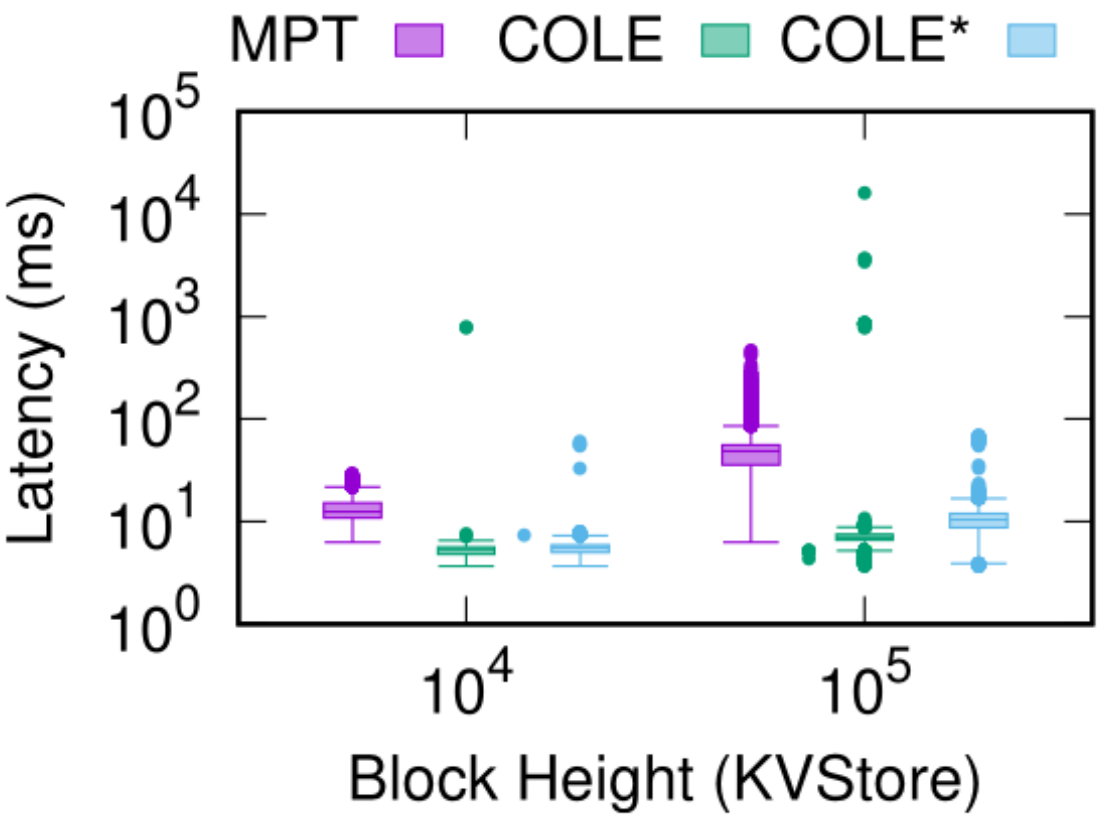
In my plot, CMI shows much higher latency compared to the other algorithms, with a wide spread, particularly at the higher block heights. This indicates that CMI struggles significantly with the write-only workload as block height increases.

The article does not include CMI in this particular plot, focusing on the comparison between MPT, COLE, and COLE*.

Overall, my results are in alignment with the findings from the article, though there are some differences in the degree of variability, which could be attributed to differences in experimental conditions.

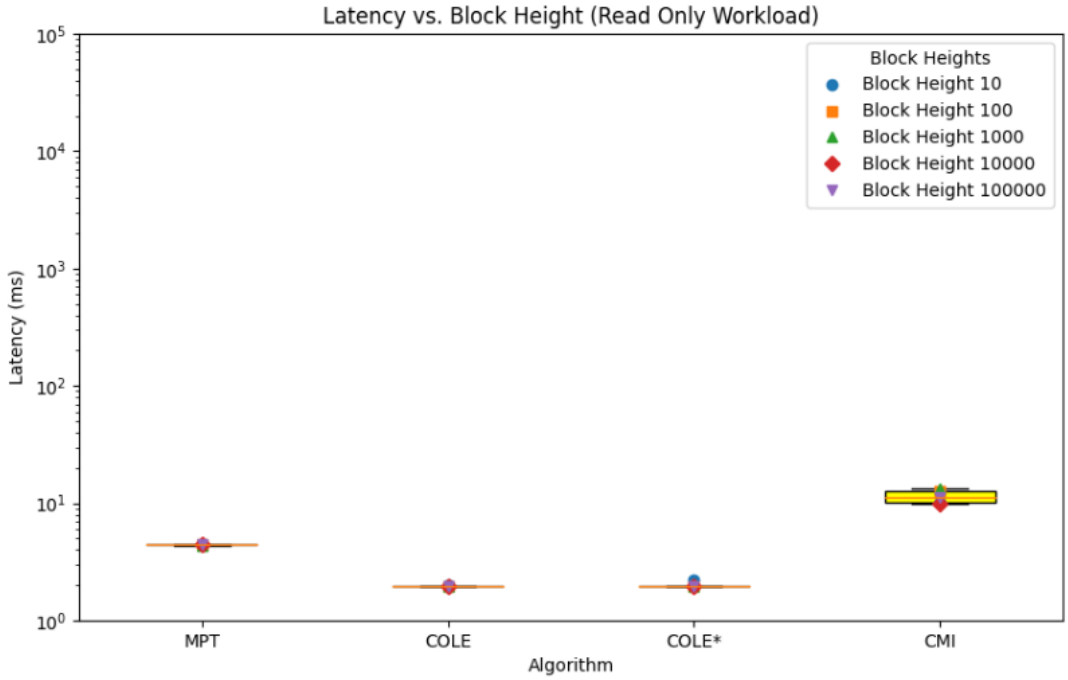


(a) My Plot



(b) COLE's Figure 12

4.11 Latency in KV store in Read only and Half Read-Half Write

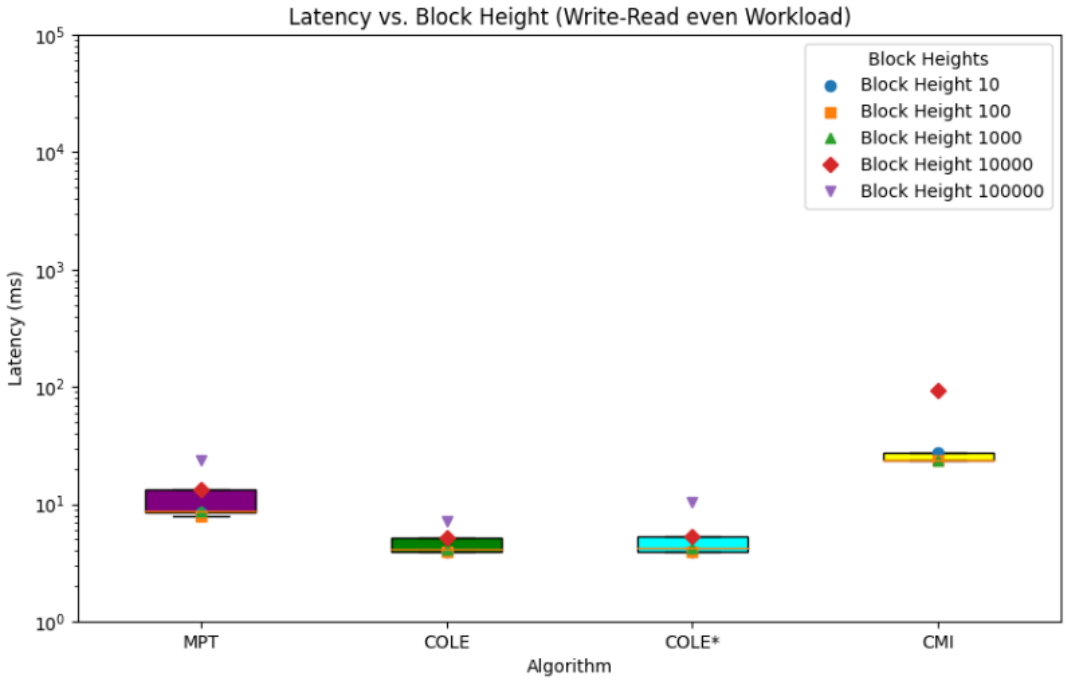


MPT, COLE, COLE*: All three algorithms show very low and nearly consistent latencies across all block heights. The latency values are tightly clustered, indicating that the read-only workload is handled efficiently by all these algorithms.

CMI: Although CMI also maintains a low latency, it shows slightly more variability compared to the other algorithms, with some outliers at higher block heights. However, the latency is still relatively low overall.

MPT, COLE, COLE*: In a read-only scenario, the algorithms do not need to handle the complexities of write operations, such as merging or updating indices. Therefore, the latency remains consistently low across all block heights. COLE and COLE* perform particularly well because their learned indices and column-based design allow for efficient data retrieval without the need for complex updates.

CMI: While CMI performs reasonably well in a read-only scenario, it shows slightly more variability due to its column-based design, which, while efficient, might introduce some overhead when accessing data in a purely read scenario.



MPT: Shows increased latency compared to the read-only workload, with a wider distribution and some noticeable outliers as the block height increases. This indicates that the mixed write-read workload introduces additional overhead that affects MPT more significantly.

COLE, COLE*: Both algorithms still maintain relatively low latencies, but COLE* shows slightly higher variability and outliers, particularly at higher block heights. This is expected due to the asynchronous merge process, which introduces some overhead during writes.

CMI: Shows the highest latency and the widest distribution, especially at higher block heights. The mixed workload, which includes write operations, exposes CMI's inefficiencies, particularly as the block height increases.

MPT: The inclusion of write operations in the workload leads to increased latency for MPT. This is due to the overhead associated with persisting index nodes during updates, which adds significant delay, especially as the blockchain grows.

COLE, COLE*: COLE continues to perform well, although COLE* shows some increased variability. The asynchronous merge process in COLE* can introduce delays during heavy write operations, leading to the observed outliers. However, both algorithms still outperform MPT due to their efficient indexing and data management.

CMI: The mixed workload exacerbates CMI's inefficiencies. The need to update and manage the column-based design during write operations, especially at higher block heights, leads to significantly higher latency and variability.

4.11.1 Possible Future Work

Further optimizations in COLE*'s asynchronous merge strategy could reduce the observed latency spikes. Techniques such as dynamic merge scheduling, incremental merging, or prioritizing critical operations during heavy write periods could be explored to smooth out performance.

The CMI algorithm could benefit from optimizations focused on reducing the overhead of write operations. For instance, integrating a more efficient index update strategy or reducing the complexity of managing columnar data could help bring down latency during mixed workloads.

Also, implementing an adaptive resource management system that dynamically allocates computational and I/O resources based on the current workload could help maintain low latency across varying scenarios.

This system could prioritize read operations during read-heavy periods and allocate more resources to write operations during write-heavy periods.

A hybrid approach that combines the strengths of learned indices with traditional indexing methods could be explored. This might involve using a learned index for the most frequently accessed data and a traditional index for less frequently accessed data, balancing performance across different workloads.

Introducing real-time monitoring tools that detect performance bottlenecks and automatically adjust system parameters could help maintain optimal performance. For example, the system could automatically adjust the aggressiveness of the asynchronous merge process based on current system load.

5 Problems and Possible Future Work

5.1 Synchronous Merging in COLE

5.1.1 Problem

Immediate Overhead: COLE uses synchronous merging, which requires the system to merge data immediately as transactions are processed. This ensures that the state of the database remains consistent and optimized at all times. However, this process introduces a significant overhead because it requires the system to pause transaction processing to handle merges, especially when dealing with large volumes of data.

Impact at Scale: As block height increases, the amount of data that needs to be managed grows, leading to larger and more frequent merge operations. The synchronous nature of these merges means that COLE's throughput decreases significantly under high transaction volumes, particularly in write-heavy workloads. The immediate merge requirement drains computational resources, leading to bottlenecks that slow down transaction processing.

5.1.2 Specific Issues Observed

Throughput Decline: In the plots, COLE shows a noticeable decline in throughput as block height increases, particularly for write-only and read-write workloads. This decline is more pronounced compared to COLE*.

Write-Heavy Workload Impact: The overhead from synchronous merging is especially problematic in write-heavy scenarios, where the system must frequently update the state and immediately merge changes. This leads to a rapid decrease in throughput as the block height increases.

5.1.3 Possible Improvements

Optimized Merge Scheduling: Implementing an adaptive merge schedule where the frequency and timing of merges adjust dynamically based on the current transaction load. This would allow the system to perform fewer merges during high transaction periods and more merges during low activity times, balancing the need for consistency with throughput performance.

Parallel Merging: Introducing parallel processing for merge operations, allowing multiple merges to occur simultaneously across different threads or processors. This could reduce the time needed for each merge operation, decreasing the impact on throughput.

Incremental Merging: Instead of performing full merges immediately, COLE could benefit from incremental merging, where data is merged in smaller, more manageable chunks. This would distribute the merging workload more evenly over time, preventing large, resource-intensive operations from slowing down the system.

5.2 Asynchronous Merging in COLE*

5.2.1 Problem

Deferred Merge Accumulation: COLE* employs asynchronous merging, which defers merge operations to be handled later, allowing the system to prioritize transaction processing. While this improves throughput by

avoiding immediate overhead, it can lead to the accumulation of deferred merges. If these deferred operations are not managed properly, they can eventually overwhelm the system, causing spikes in latency and potential drops in throughput when these operations are finally executed.

Resource Utilization Variability: The asynchronous nature of merging can lead to uneven resource usage. For example, during periods of high transaction processing, deferred merges might pile up, leading to a sudden need for intensive resource allocation when these merges are processed. This can cause variability in performance, with periods of high throughput followed by potential slowdowns.

5.2.2 Specific Issues Observed

Higher Throughput with Potential Latency Spikes: In the plots, COLE* once shows higher throughput compared to COLE, particularly in read-heavy workloads. However, there is a concern that the deferred merges could introduce latency spikes, especially as the block height increases. This could manifest as sudden drops in performance if the system needs to process a large number of deferred merges at once.

Scalability Concerns: As the block height grows, the amount of deferred work could increase, leading to scalability issues where the system struggles to maintain consistent performance across large datasets.

5.2.3 Possible Improvements

Incremental and Smarter Deferred Merging: Introducing incremental deferred merging strategies where deferred operations are processed in smaller, more frequent batches rather than accumulating into large, potentially disruptive tasks. This would help maintain a more consistent performance profile.

Dynamic Resource Allocation: Implementing a dynamic resource management system that adjusts CPU, memory, and I/O resource allocation based on the current backlog of deferred merges. This would ensure that deferred operations do not accumulate to a point where they severely impact performance when finally processed.

Hybrid Merge Approach: Consider adopting a hybrid approach that combines asynchronous and synchronous merging. For example, critical state changes that directly impact user queries could be merged synchronously, while less critical changes are handled asynchronously. This could reduce the load on the system while still maintaining high throughput.

5.3 Scalability Issues with Increasing Block Height

5.3.1 Problem

Growing Data Complexity: As block height increases, the volume of data the system needs to manage grows exponentially. This increases the complexity of both the data structures and the merging process. Larger datasets mean more data must be merged, stored, and retrieved, which places additional stress on the system's resources.

Impact on Both COLE and COLE*: Both COLE and COLE* experience performance degradation as block height increases. COLE is impacted more by the immediate overhead of synchronous merges, while COLE* faces challenges with managing deferred operations and maintaining consistent performance.

5.3.2 Specific Issues Observed

Throughput Decline Across All Systems: The plots show a general decline in throughput for both COLE and COLE* as block height increases, with COLE* maintaining higher throughput but potentially facing more variability.

Latent Performance Issues: The growing complexity could lead to latent issues where the system's performance gradually deteriorates over time as block height and data volume increase.

5.3.3 Possible Improvements

Scalability Enhancements: Implementing data structure optimizations that reduce the complexity of managing large block heights. For instance, leveraging more efficient data partitioning or sharding techniques could help distribute the workload more evenly across the system.

Better Indexing Techniques: Exploring enhancements to the learned index approach in COLE and COLE* that can scale more effectively with increasing data sizes. This might include more sophisticated models that can handle larger datasets without a proportional increase in computational overhead.

Resource Augmentation: For systems expected to operate at very high block heights, consider augmenting system resources, such as adding more memory or faster storage solutions, to handle the increased data volume more effectively.

5.4 Latency Variability, Especially in COLE*

5.4.1 Problem

Inconsistent Latency: The asynchronous merging strategy in COLE* introduces variability in latency, particularly in scenarios where deferred merges accumulate. This can lead to unpredictable performance, which is problematic for applications that require consistent and low-latency operations.

Impact on User Experience: For blockchain applications where timing and the order of transactions are critical, variability in latency can lead to inconsistencies and affect the overall reliability of the system.

5.4.2 Specific Issues Observed

Higher Latency Outliers in Plots: The plots indicate that COLE* occasionally exhibits higher latency outliers compared to COLE. These outliers are likely due to the asynchronous merging process catching up on deferred work, causing temporary spikes in latency.

Potential for Performance Dips: If the system encounters a period where it must process a large number of deferred merges, there could be a sudden dip in performance, leading to delays in transaction processing.

5.4.3 Possible Improvements

Latency Smoothing Techniques: Implementing techniques to smooth out latency, such as throttling the rate of deferred merge processing during periods of high load. This could help prevent sudden spikes in latency.

Real-Time Monitoring and Adjustment: Deploying real-time monitoring tools that detect when latency is beginning to increase and automatically adjust system parameters (e.g., slowing down the rate of new transactions or temporarily increasing merge processing) to maintain consistent performance.

Advanced Buffering: Introducing more sophisticated buffering strategies that can absorb sudden surges in demand without requiring immediate resource-intensive operations. This could involve pre-emptively merging certain data in anticipation of high-load periods.

6 Conclusion

In this report, I conducted a comprehensive comparison of Chain KV and Block LSM with COLE and COLE*, followed by an in-depth re-evaluation of the experiments presented in the COLE article. By comparing the re-evaluated results with those in the original article, some insights were gained into the strengths and weaknesses of these indexing methods.

COLE and COLE* have demonstrated considerable potential in efficiently managing blockchain data, thanks to their innovative use of learned indexes and advanced merging strategies. However, as the system scales and workloads intensify, both systems encounter specific challenges that could hinder their performance. To address these challenges and further enhance their capabilities, future work could involve experimenting

with new configurations, dissecting the performance of different indexes, and implementing proposed improvements such as adaptive merge scheduling, hybrid merging approaches, and optimized resource management.

These enhancements are important to making COLE and COLE* more resilient and scalable, ensuring that they continue to perform effectively in large-scale blockchain environments. By refining these systems to meet the demands of growing data volumes and increasingly complex workloads, COLE and COLE* can solidify their roles as powerful and adaptable solutions in the evolving landscape of blockchain technology

References

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

[2] Block LSM: An Ether-aware BLock-ordered LSM-tree based Key-Value Storage Engine

[3] ChainKV: A Semantics-Aware Key-Value Store for Ethereum System