COLE Demo Report

Yasmine Vaziri

August 19, 2024

Contents

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.

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 BlockLSM

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.

4 Data Storage Approach

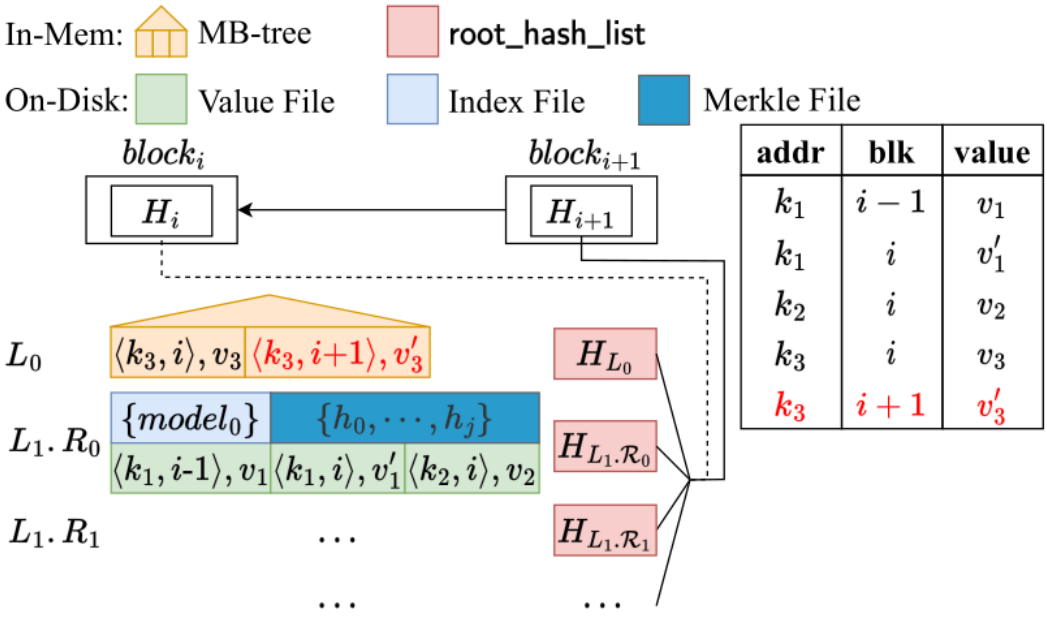
BlockLSM utilizes a **prefix-based hashing** approach to manage the storage of blockchain data efficiently. This method allows the system to maintain a structured order of data blocks, enabling quicker access and retrieval of specific data points.

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.

4.1 subComponents, Operation, and Recovery

4.1.1 COLE



Components:

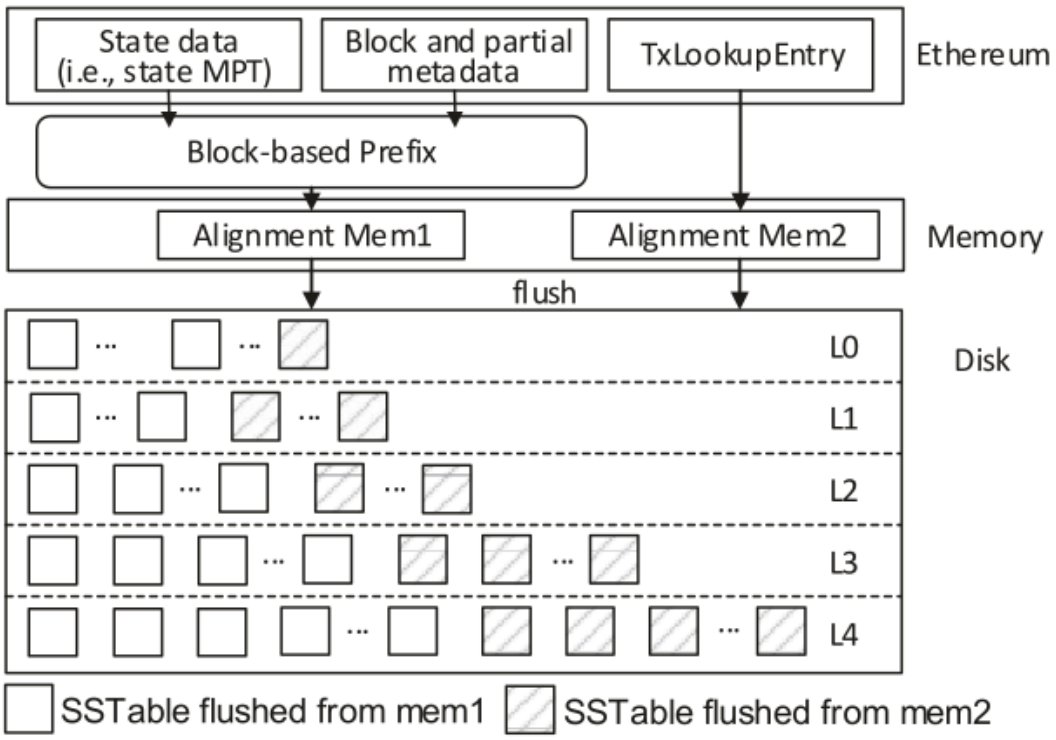
- **In-Mem MB-tree:** An in-memory structure that organizes the data in memory before it is flushed to disk.
- **On-Disk Value File:** Stores the actual data values on disk.
- **Index File:** Maintains an index for fast lookup of data.
- **Merkle File:** Ensures data integrity through hash chains.

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.

4.1.2 BlockLSM



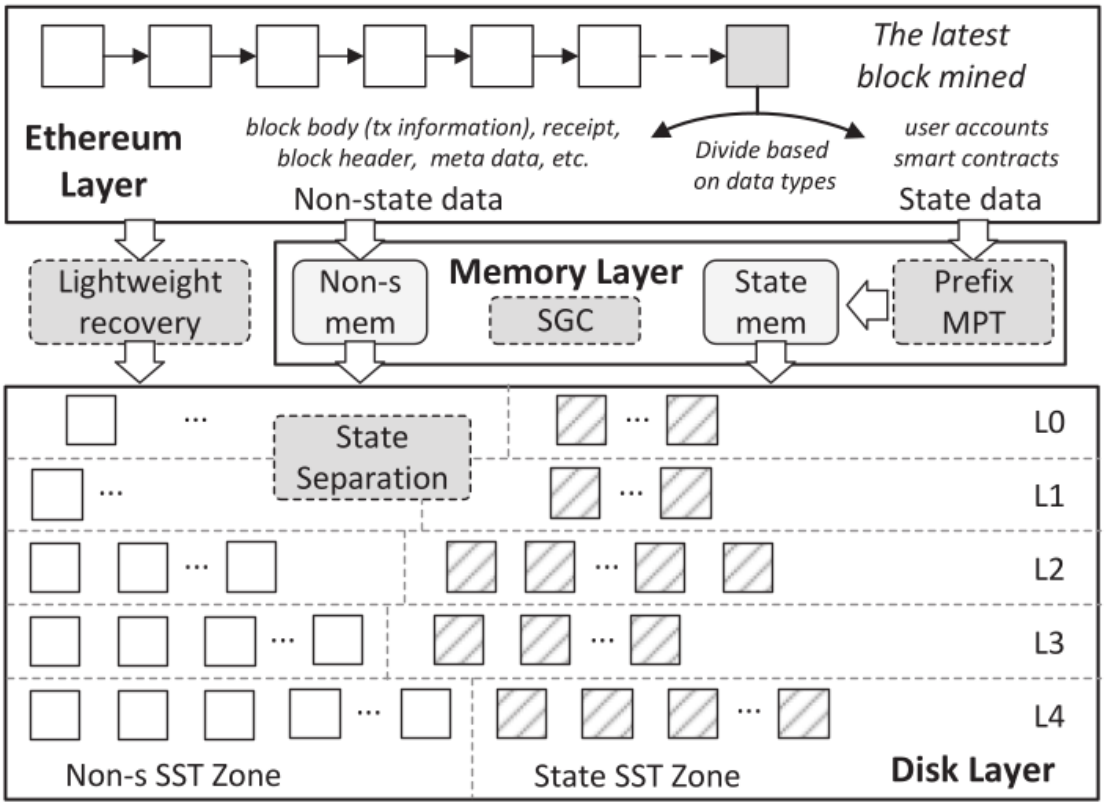
Components:

- **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.

4.1.3 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.

4.2 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.

4.3 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.

4.3.1 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.

4.4 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.

4.5 Key Differences between BlockLSM and ChainKV

4.6 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.

4.6.1 Compaction Strategies

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

4.7 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.

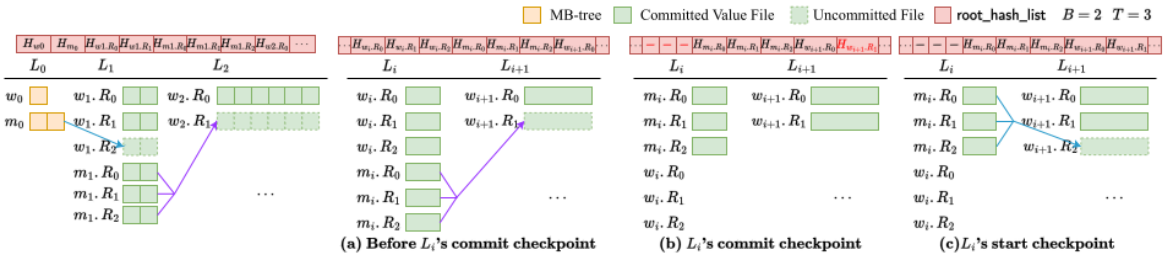
**********better to explain the learned index**********

4.8 Cache Management

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

4.9 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:



4.10 Performance and Storage Reduction

4.10.1 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.

4.10.2 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.

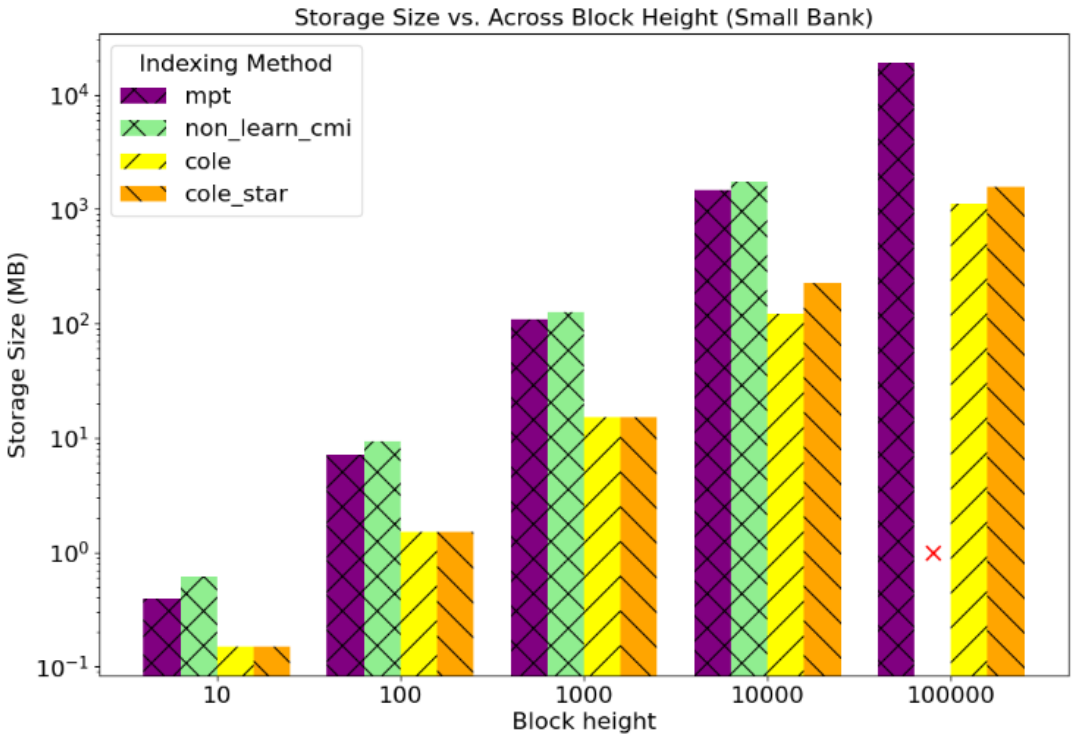
5 COLE's Experiments and Results

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.

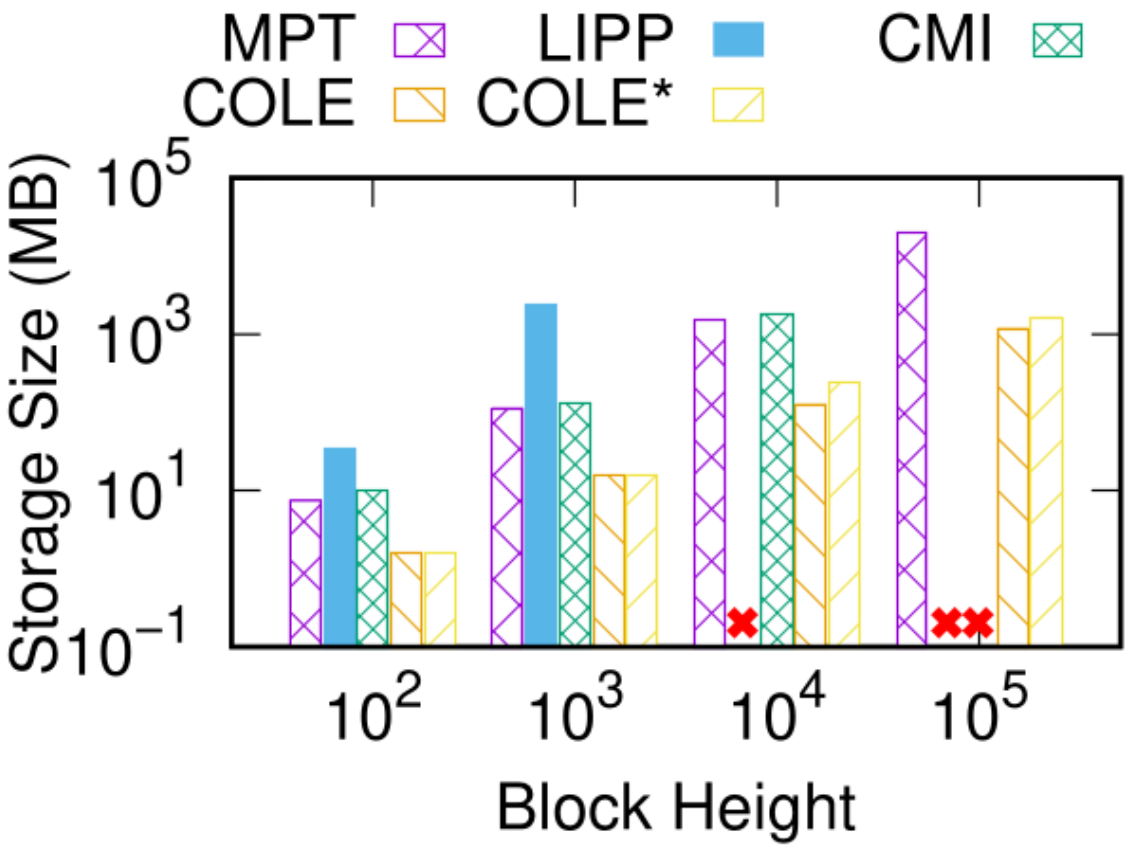
First, I plot using my own data and compare it with the one in the paper.

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

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.



(a) My Plot



(b) COLE's Figure 9

5.2 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.

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

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.

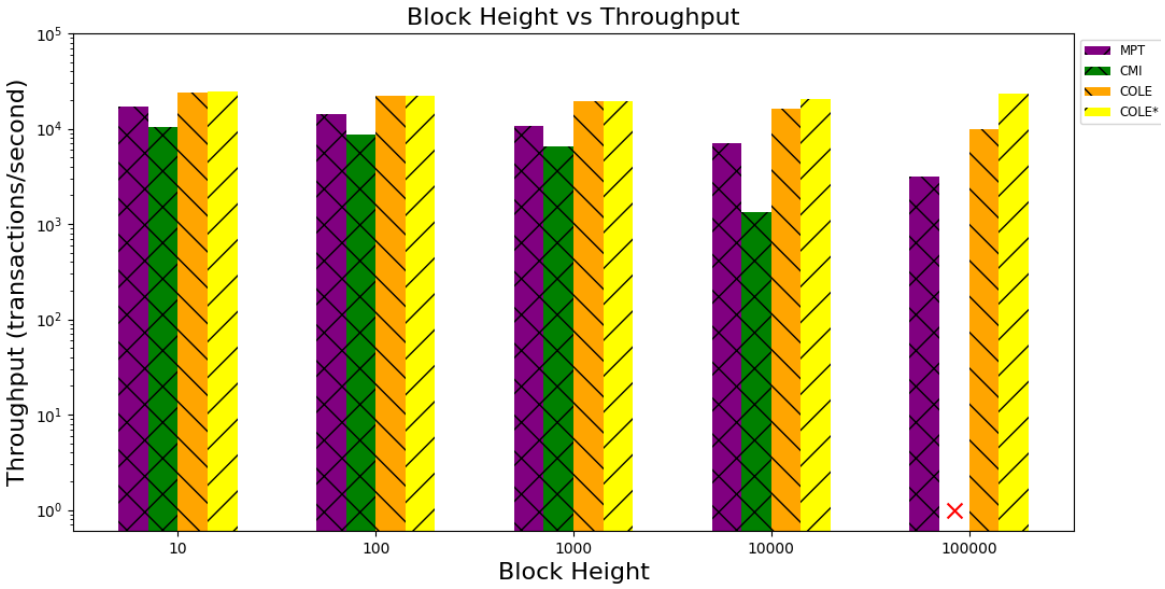
In COLE star,this variant employs asynchronous merging. In this approach, data merges are not performed immediately but are deferred to a later time, allowing the system to prioritize transaction processing over merging operations. As a result, COLE star can achieve higher throughput because the system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

5.3.1 Reason for the Discrepancy Throughput

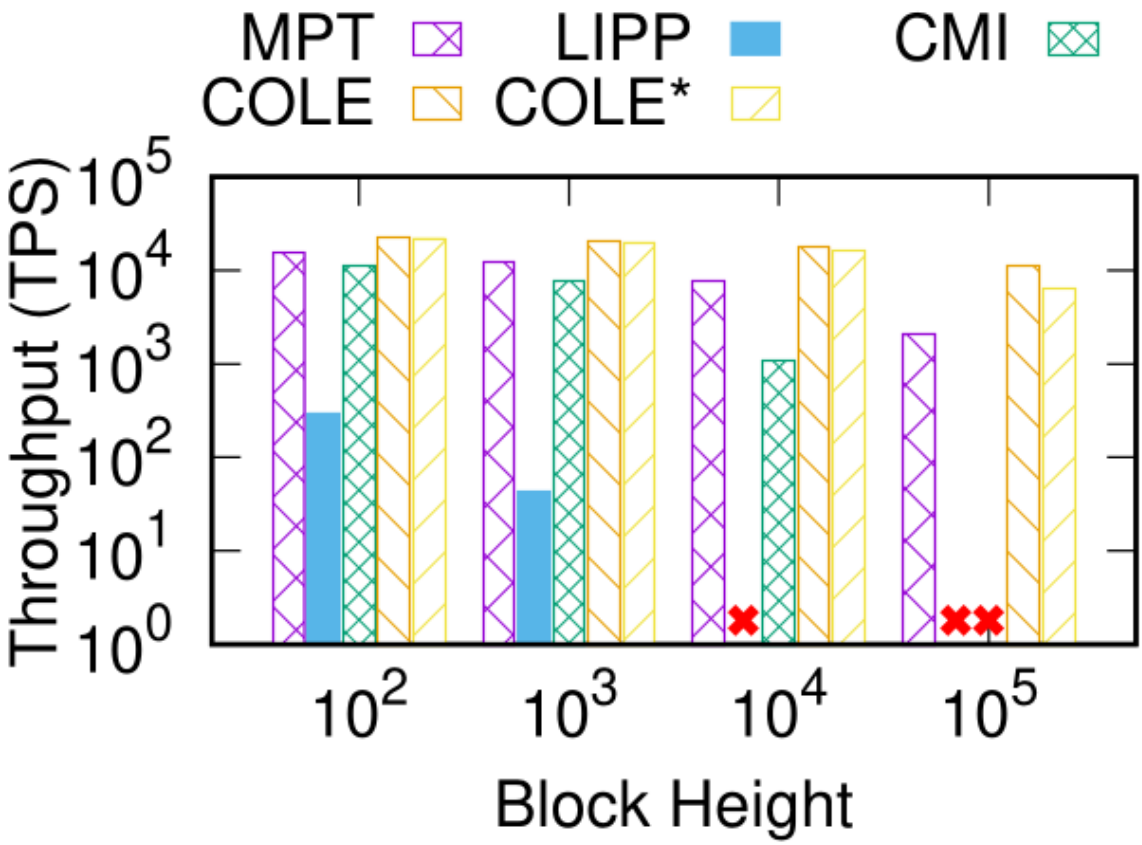
Upon reevaluating the experiment, the higher throughput observed in COLE star likely arises from the reduced overhead due to deferred merging operations. The asynchronous merge allows COLE star to process transactions more rapidly without the bottleneck created by the need to maintain the database state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

5.3.2 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.



(a) My Plot



(b) COLE's Figure 9

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.

5.4 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.

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.

5.4.1 Reasons Behind the Observed Trends

1. Data Structure Overhead:

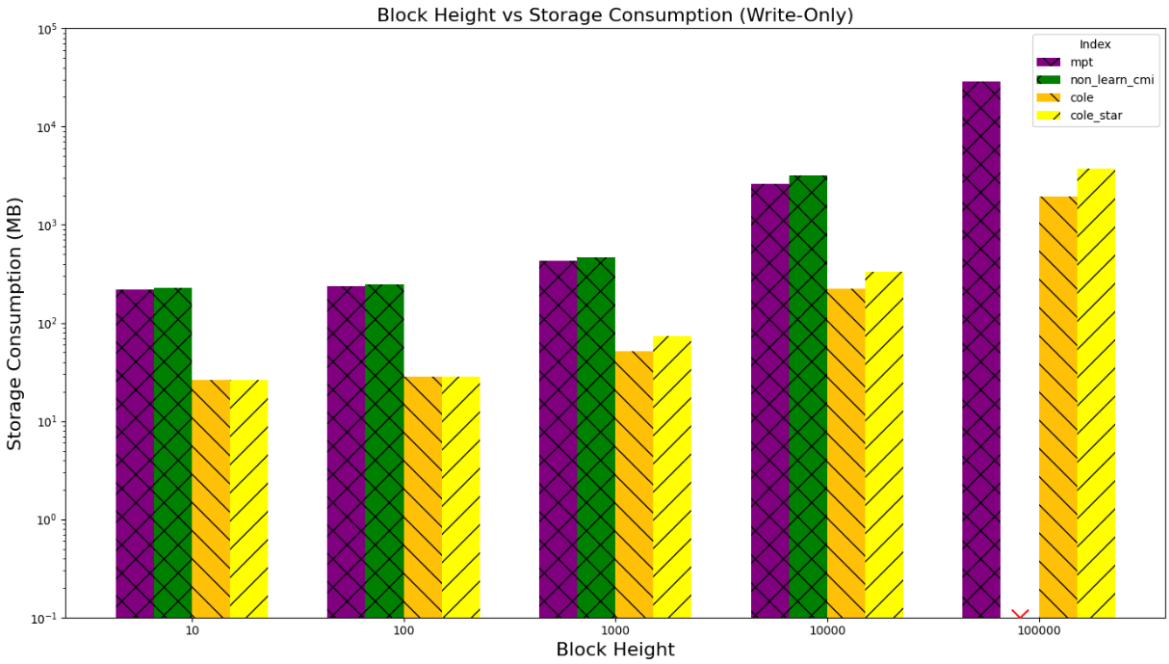
   MPT suffers from high storage overhead due to its complex structure, which needs to maintain numerous pointers and intermediate nodes. COLE and CMI by contrast, focus on minimizing redundancy and compressing similar data across blocks, resulting in more efficient storage use.

2. Column-Based Design:

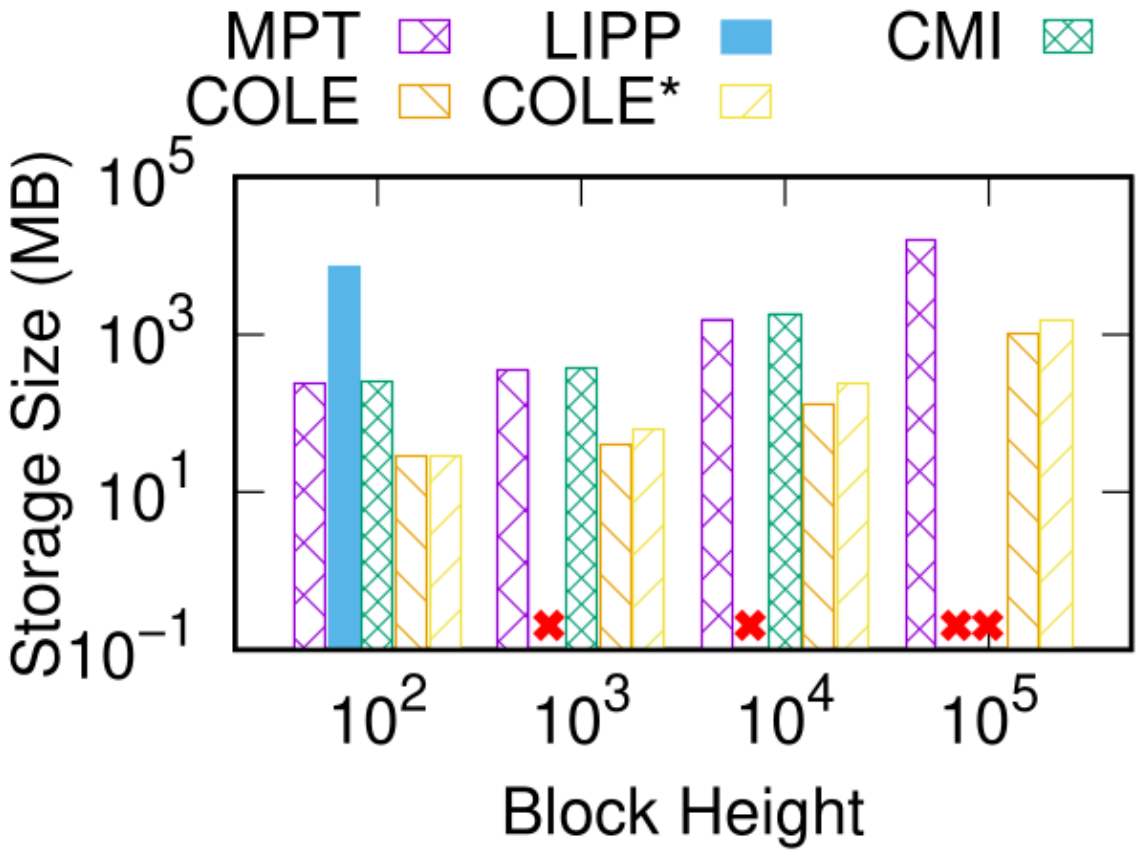
   COLE and CMI use a column-oriented design, which is inherently more storage-efficient for certain types of data. This design allows for better compression and more efficient storage management, particularly when dealing with repeated or similar data across different blocks.

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.



(a) My Plot

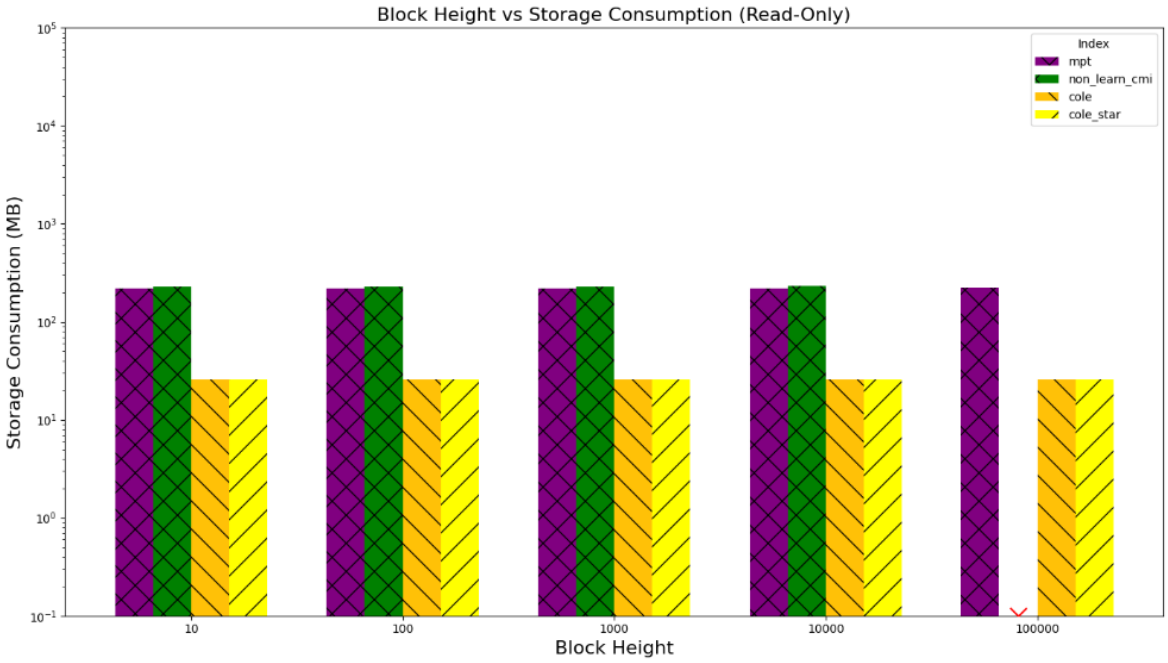


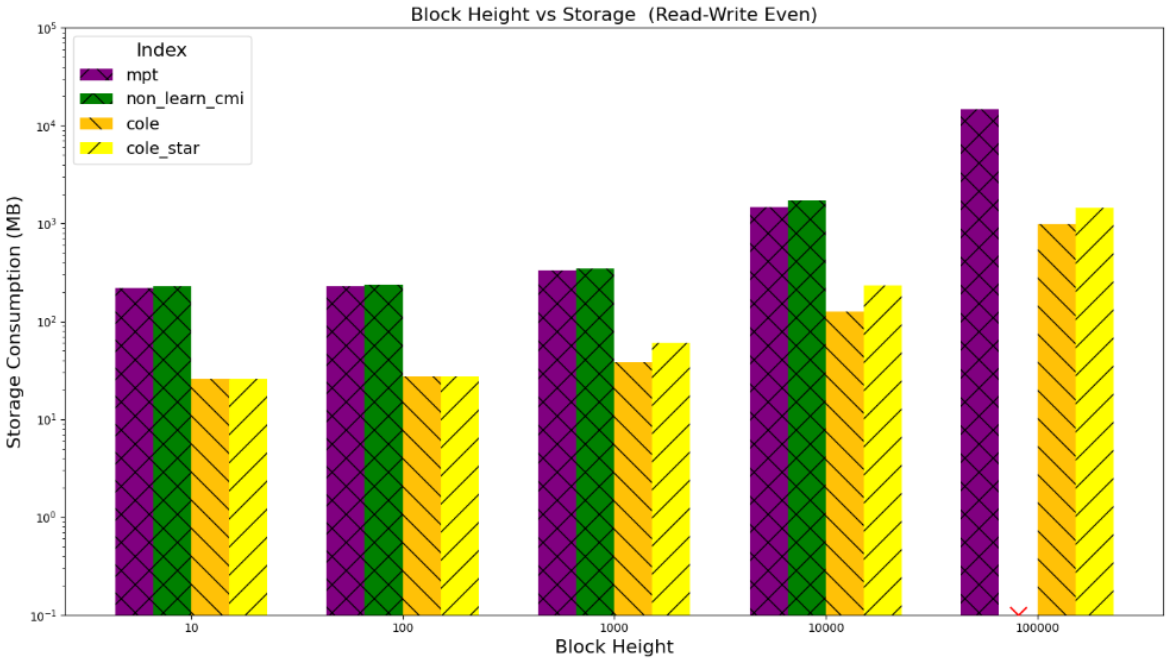
(b) COLE's Figure 9

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:

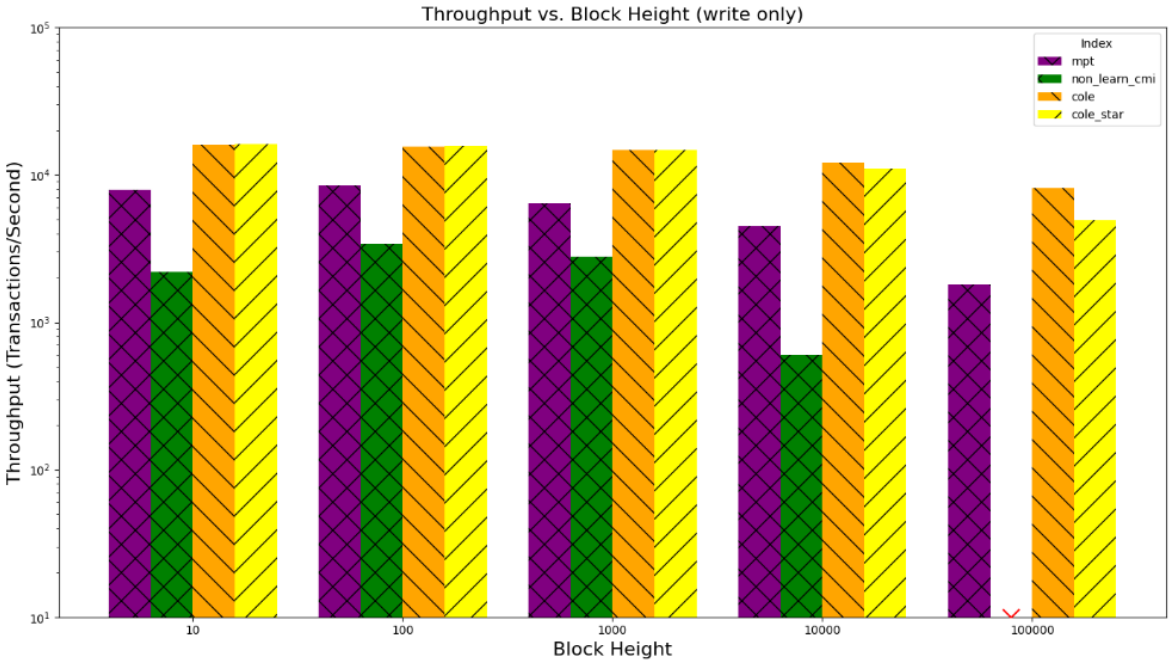


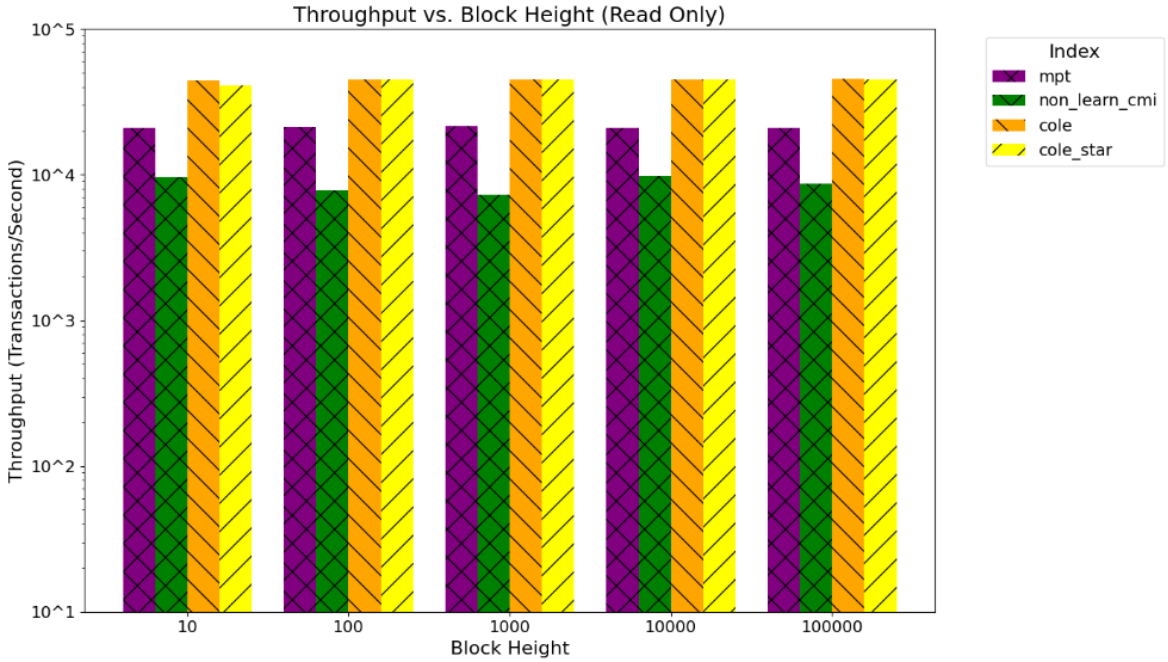


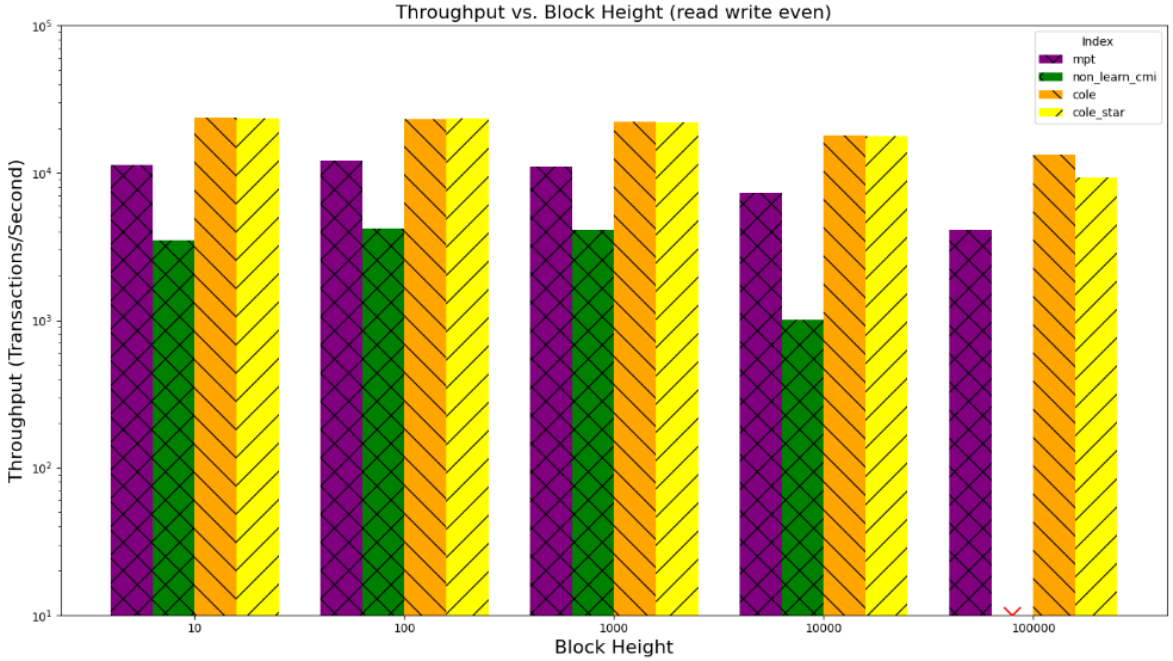
The main article has not mentioned about these two functionality. As we can see, in read-only, COLE and COLE star have same storage consumption and what we can do is to work more on this workload to make the COLE's functionality better both in read and read-write even.

******explain the reason********

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

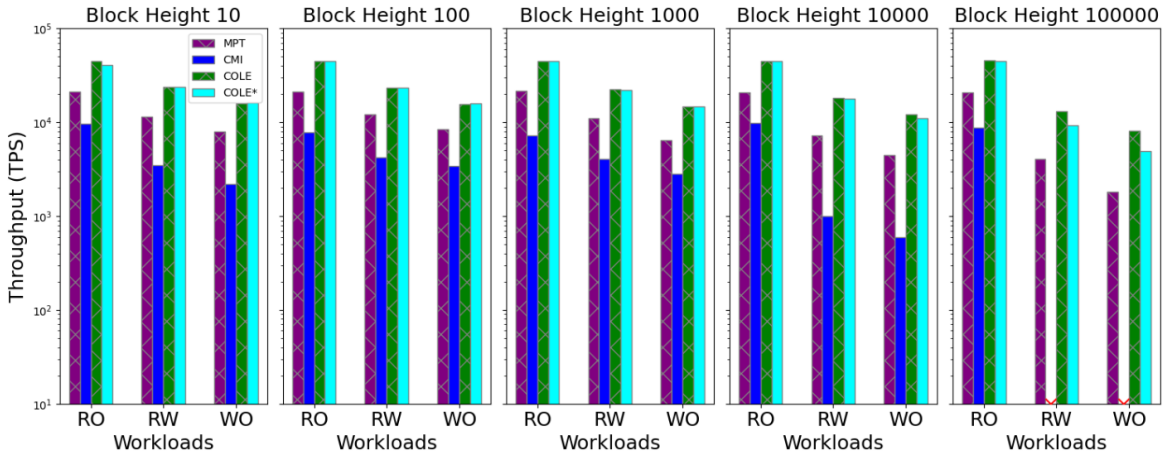






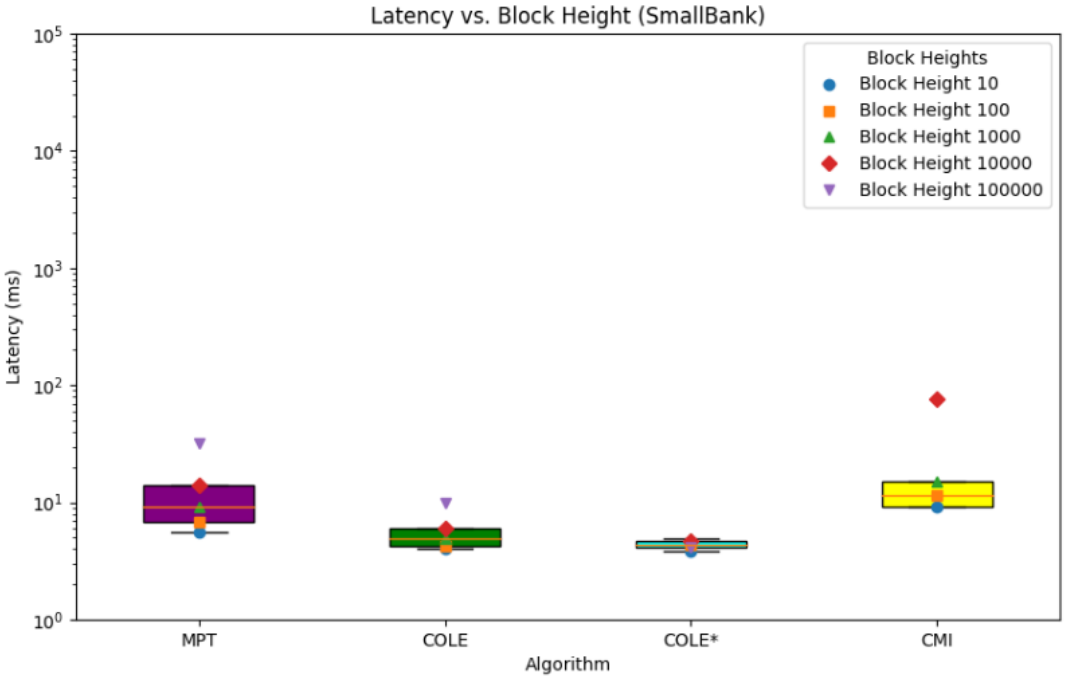
*****reasoning*****

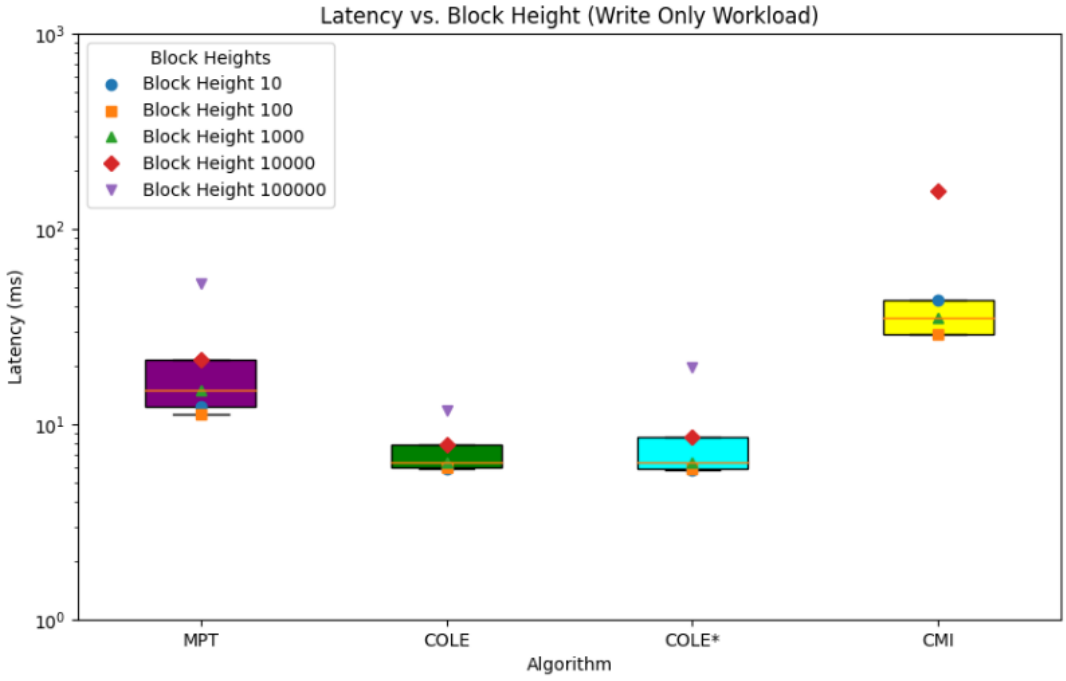
5.6 Throughput vs. Workload in different block heights

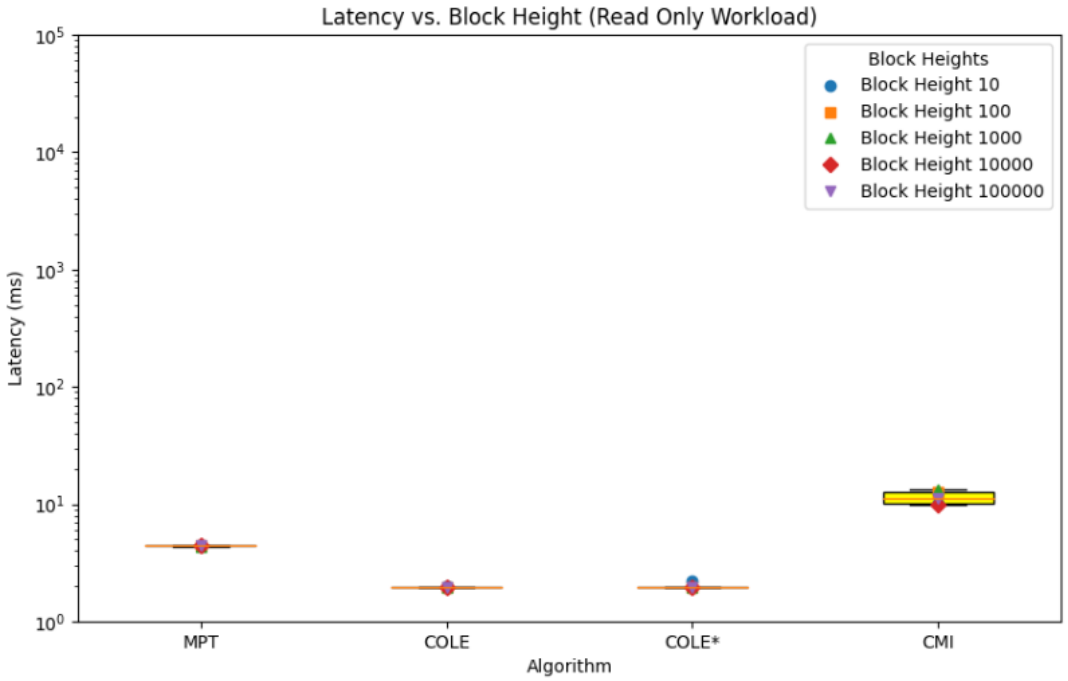


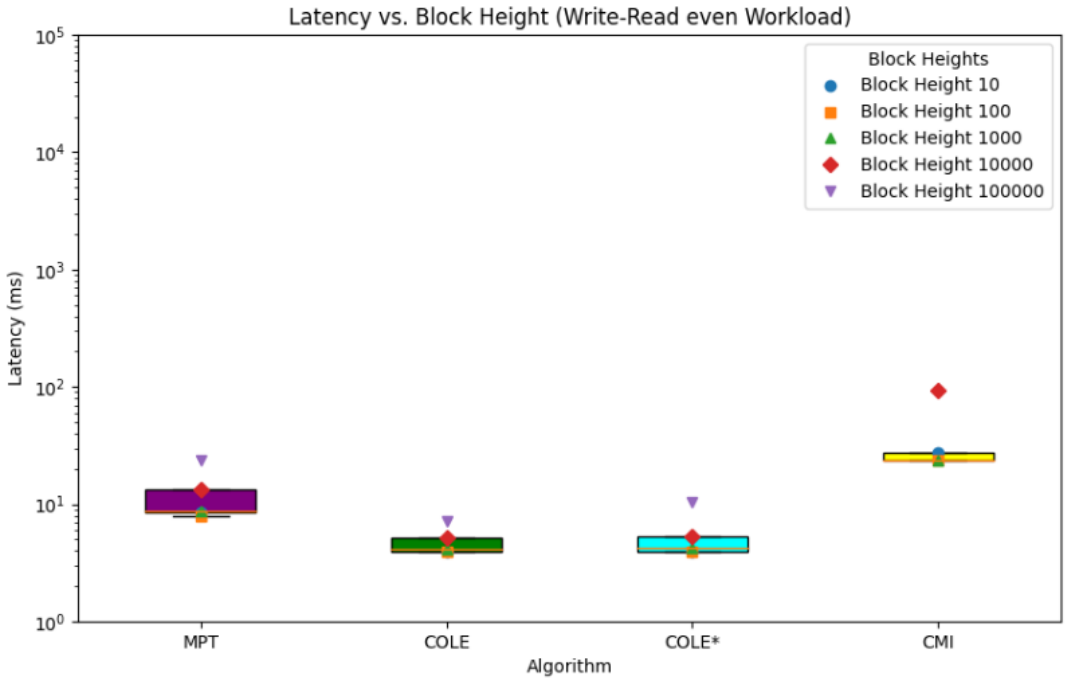
**********reason**********

5.7 Latency in Small Bank and KV Store

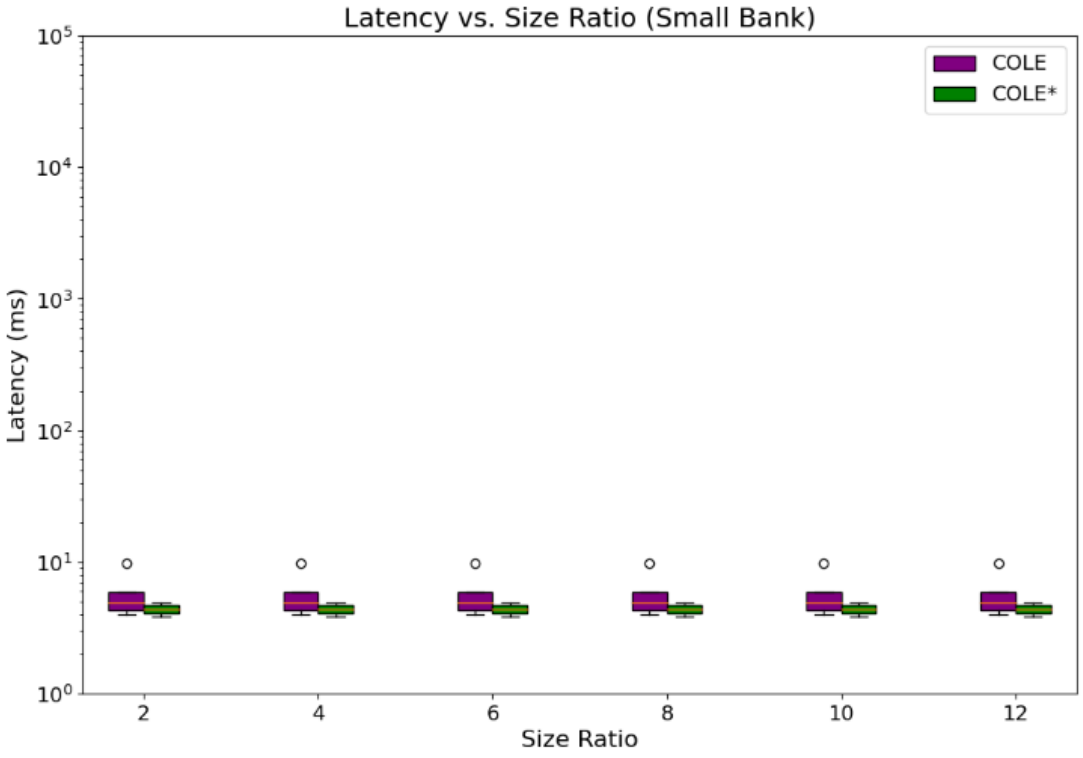


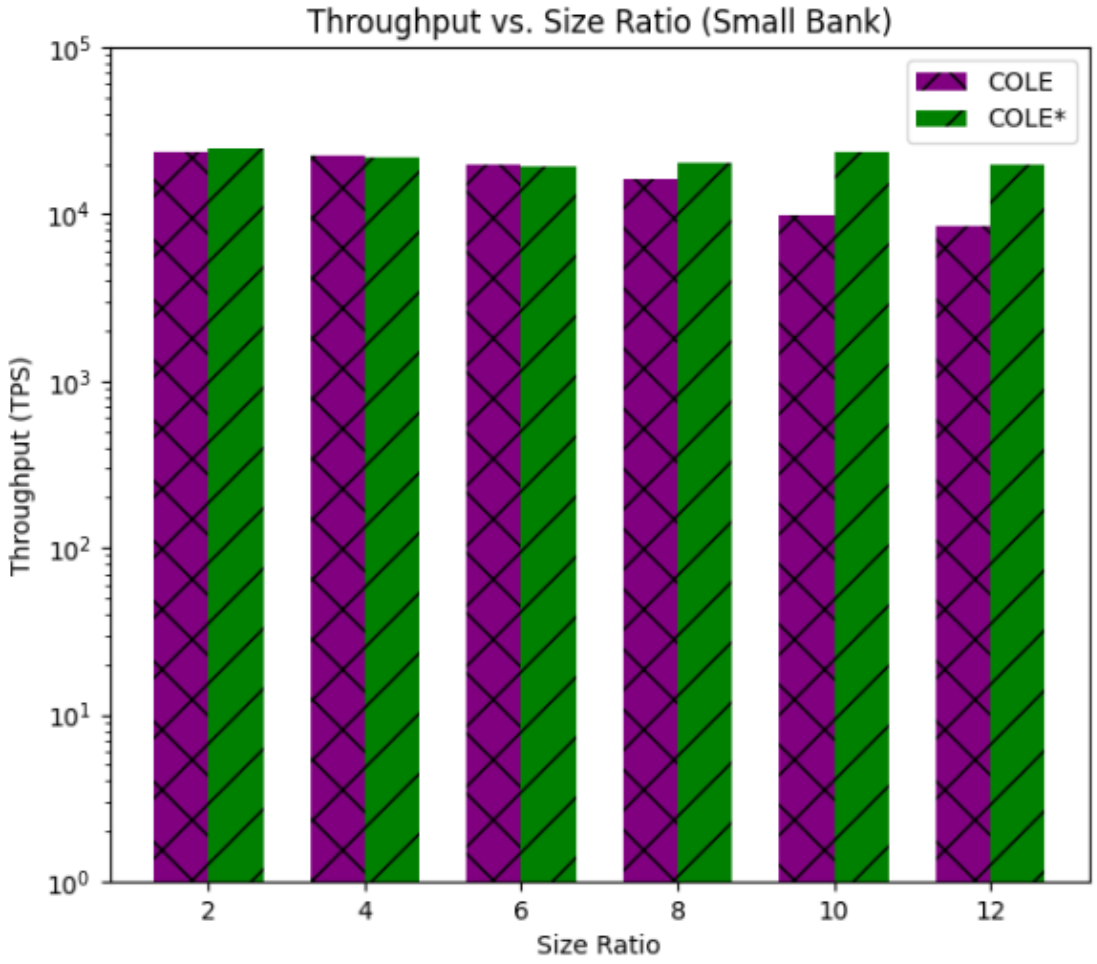



Latency vs. Block Height (Read Only Workload)
Block Heights
Block Height 10
Block Height 100
Block Height 1000
Block Height 10000
Block Height 100000
Latency (ms)
10^5
10^4
10^3
10^2
10^1
10^0
MPT
COLE
COLE*
CMI
Algorithm


Latency vs. Block Height (Write-Read even Workload)
Block Heights
Block Height 10
Block Height 100
Block Height 1000
Block Height 10000
Block Height 100000
Latency (ms)
10^5
10^4
10^3
10^2
10^1
10^0
MPT
COLE
COLE*
CMI
Algorithm

5.8 Impact of Size Ratio





**********explain and reasons**********

6 Problems

7 Future Work

8 conclution