The Comparison of COLE, ChainKV and BlockLSM

Data Storage Approach

BlockLSM	ChainKV	COLE
Prefix-based Hashing	Semantic-Aware Zones	Column-Based Design

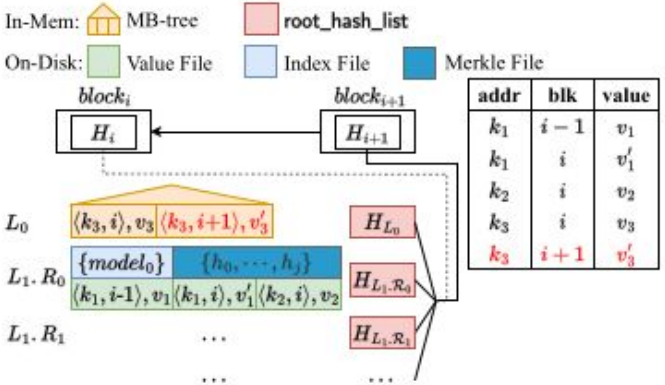


Figure 3: Overview of COLE

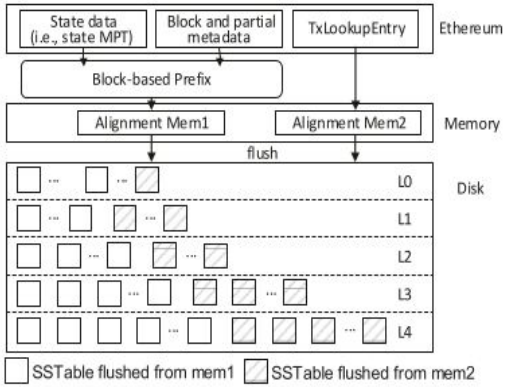


Fig. 3. Architecture overview of Block-LSM.

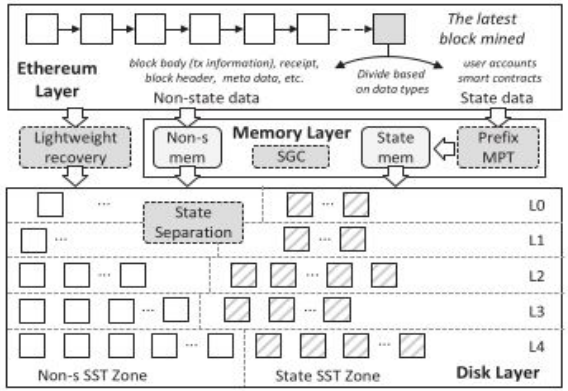


Fig. 4. The overall architecture of ChainKV.

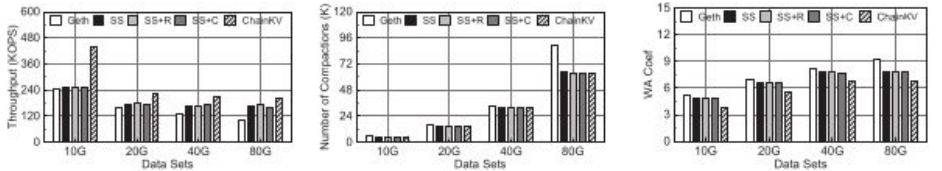
Explanation

- 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, which means that it categorizes 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.

Compaction Reduction

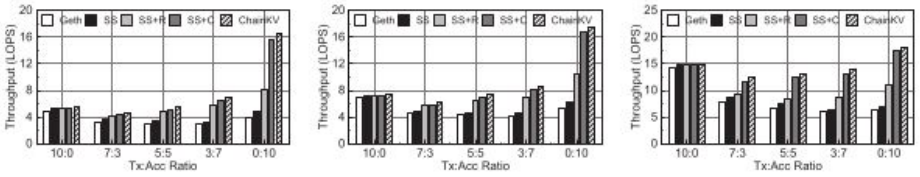
BlockLSM	ChainKV	COLE
Yes	Yes	N/A

Compaction Behavior in ChainKV



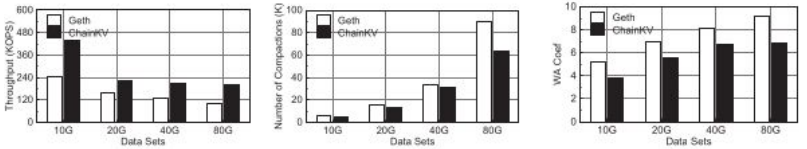
(a) Write Throughput. (b) Compaction. (c) Write Amplification (WA) Coef.

Fig. 9. Breakdown analysis of data writes.



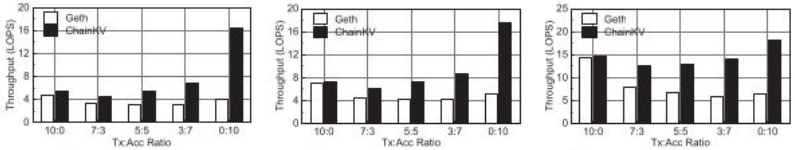
(a) Normal Random. (b) Scrambled Zipfian. (c) Skewed Latest Zipfian.

Fig. 10. Breakdown analysis of data reads.



(a) Write Throughput. (b) Compaction. (c) Write Amplification (WA) Coef.

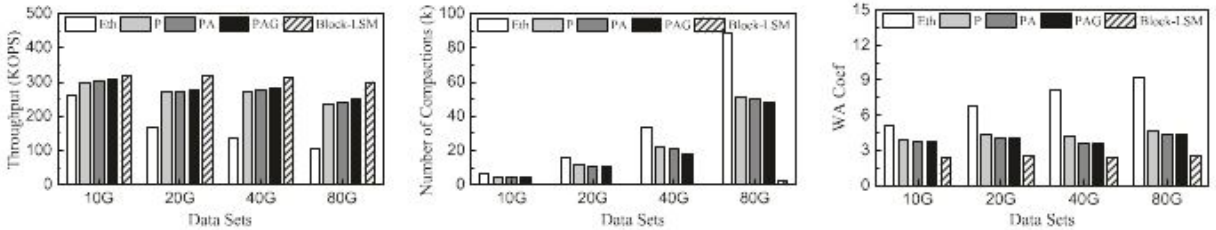
Fig. 7. The overall performance comparison for data writes.



(a) Normal Random. (b) Scrambled Zipfian. (c) Skewed Latest Zipfian.

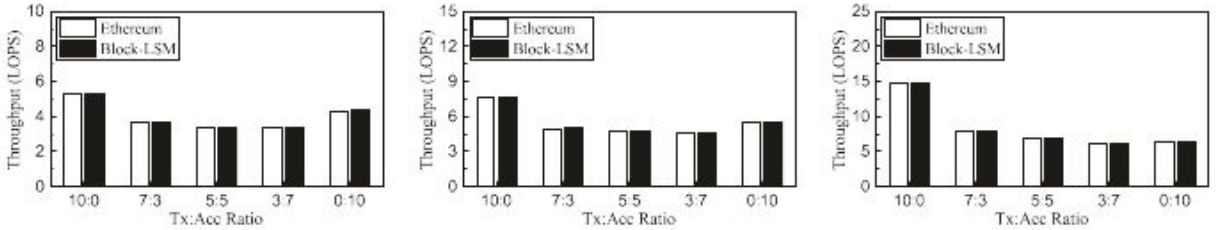
Fig. 8. The overall performance comparison for data reads

Compaction Behavior in BlockLSM



(a) Write Throughput. (b) Number of Compactions (c) WA Coef.

Fig. 8. Breakdown analysis of data synchronization.



(a) Normal Random. (b) Scrambled Zipfian. (c) Skewed Zipfian.

Fig. 9. The system read performance.

Explanation

- BlockLSM includes a compaction process that reduces data redundancy and storage costs by periodically merging data blocks and eliminating obsolete data.
- ChainKV also implements compaction techniques to manage data more effectively, similar to BlockLSM, which helps in maintaining storage efficiency.
- COLE does not apply traditional compaction techniques. Instead, it relies on its column-based design and learned indexing to manage data efficiently without the need for periodic data compaction.

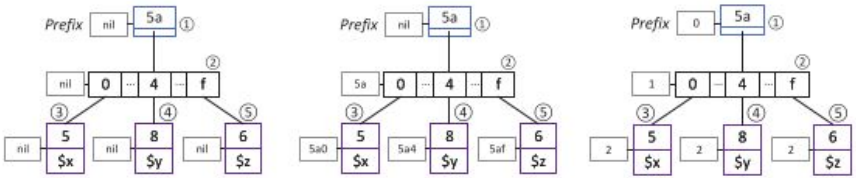
Data Locality

BlockLSM	ChainKV	COLE
Maintained via Prefixes	Preserved via Prefix MPT	Improved via Learned Index

Explanation

- 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

MPT in ChainKV & Learned-Index in COLE



(a) The structure of Prefix MPT. (b) With Column Prefix. (c) With Row Prefix.

Fig. 5. (a) demonstrate the structure of Prefix MPT, which contains 3 accounts mentioned in Figure 1. (b) and (c) illustrate the method to generate prefixes for column strategy and row strategy, respectively.

Algorithm 2: Learn Models from a Stream

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

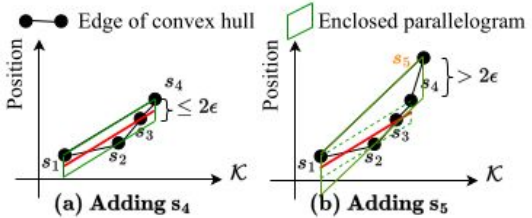


Figure 5: An Example of Model Learning

Cache Management

BlockLSM	ChainKV	COLE
Memory Buffers	Gaming Cache	Asynchronous Merges

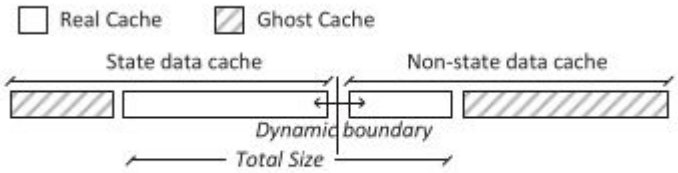


Fig. 6. The Structure of the SGC design.

VS

Algorithm 5: Write Algorithm with Asynchronous Merge

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

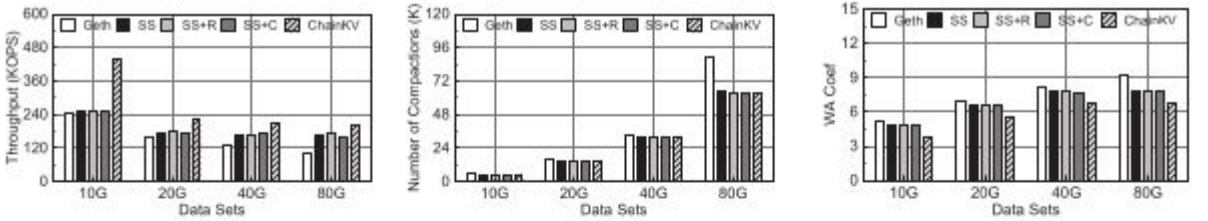
Explanation

- BlockLSM uses memory buffers to manage its cache. This approach helps in temporarily storing data in memory for quick access before it is written to disk.
- ChainKV employs a space-gaming cache strategy, which optimizes the use of available cache space to improve performance.
- COLE utilizes asynchronous merges for cache management, which allows it to perform data merging operations in the background without affecting the system's performance.

Recovery Mechanism

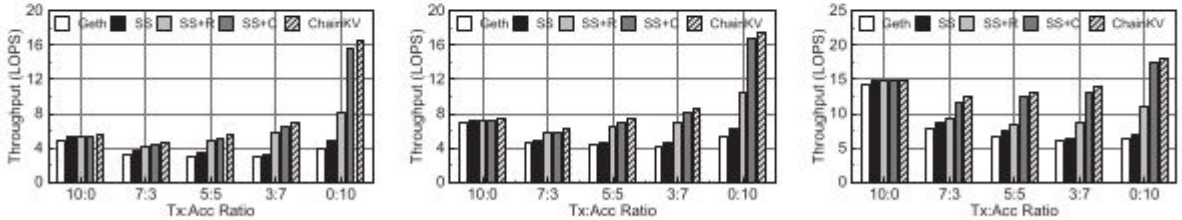
BlockLSM	ChainKV	COLE
Standard WAL	Lightweight Node Recovery	Asynchronous Merge Checkpoints

WA Performance in BlockLSM



(a) Write Throughput. (b) Compaction. (c) Write Amplification (WA) Coef.

Fig. 9. Breakdown analysis of data writes.



(a) Normal Random. (b) Scrambled Zipfian. (c) Skewed Latest Zipfian.

Fig. 10. Breakdown analysis of data reads.

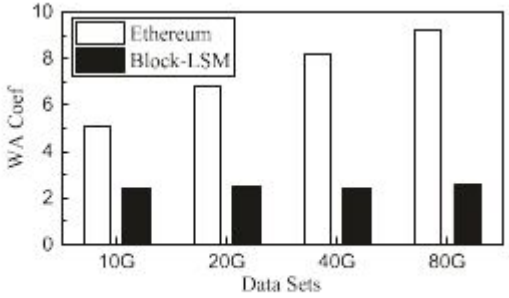
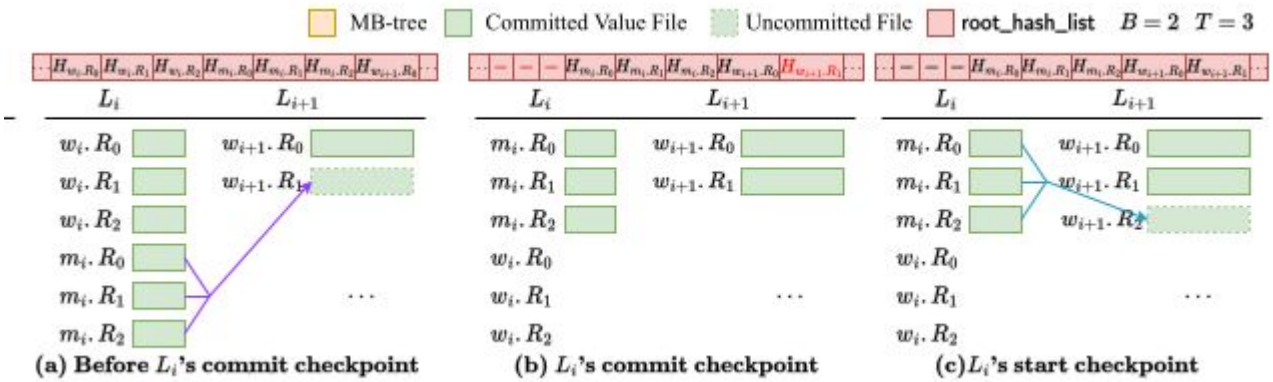


Fig. 6. The WA coef during data synchronization.

CheckPoints in COLE



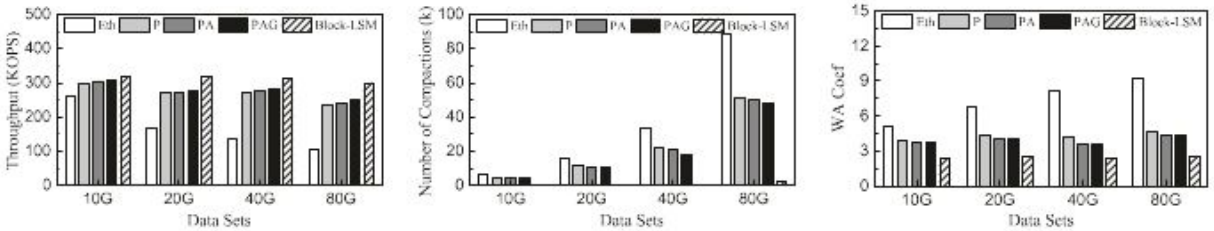
Explanation

- 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.

Performance Improvement

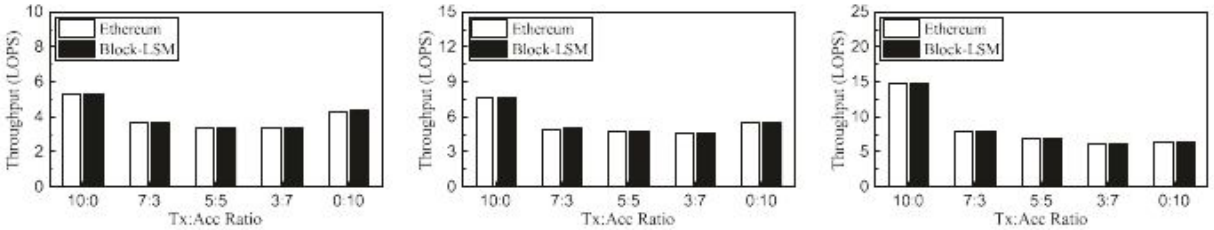
BlockLSM	ChainKV	COLE
Up to 182.75%	Up to 4.20x	Up to 5.4x
The performance improvement in BlockLSM comes from its efficient use of prefix-based hashing and compaction reduction. These mechanisms help to reduce write amplification and maintain data locality, leading to faster data retrieval and lower latency.	ChainKV achieves performance improvements through the use of semantic-aware zones and space-gaming cache management. These techniques optimize data placement and retrieval, reducing overhead and enhancing system throughput.	It is driven by its column-based design and the use of learned indexes. The column-based storage reduces redundancy and speeds up data retrieval, while learned indexes optimize data access patterns, resulting in significant performance gains.

Read Performance in BlockLSM



(a) Write Throughput. (b) Number of Compactions (c) WA Coef.

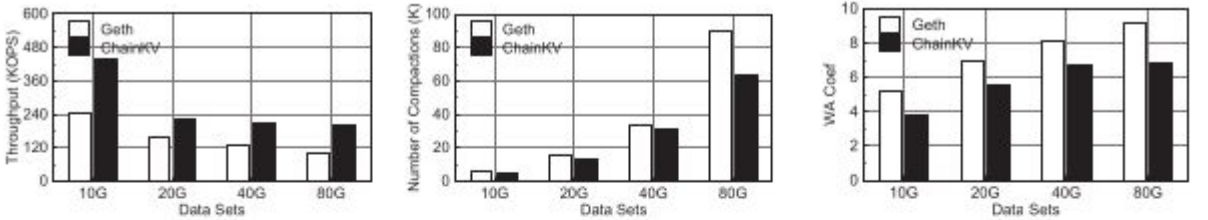
Fig. 8. Breakdown analysis of data synchronization.



(a) Normal Random. (b) Scrambled Zipfian. (c) Skewed Zipfian.

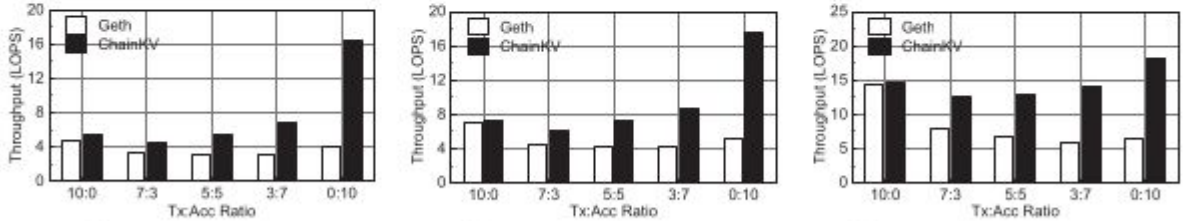
Fig. 9. The system read performance.

Read Performance in ChainKV



(a) Write Throughput. (b) Compaction. (c) Write Amplification (WA) Coef.

Fig. 7. The overall performance comparison for data writes.



(a) Normal Random. (b) Scrambled Zipfian. (c) Skewed Latest Zipfian.

Fig. 8. The overall performance comparison for data reads

Performance and Storage Reduction in COLE

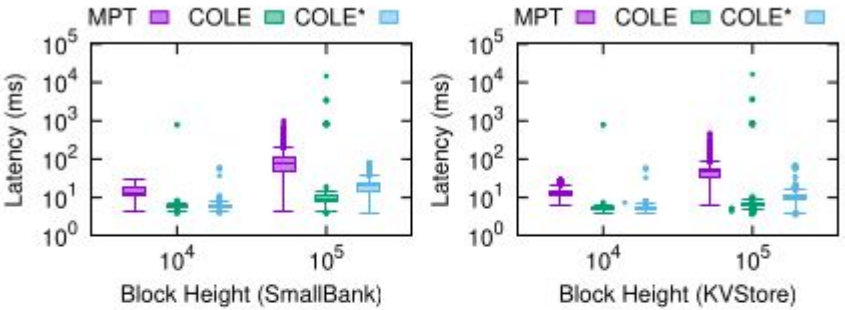


Figure 12: Latency Box Plot

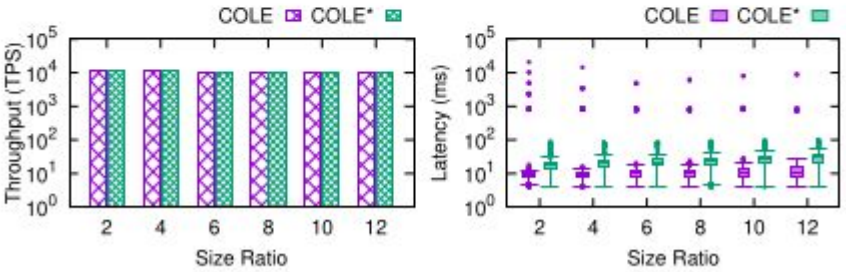


Figure 13: Impact of Size Ratio

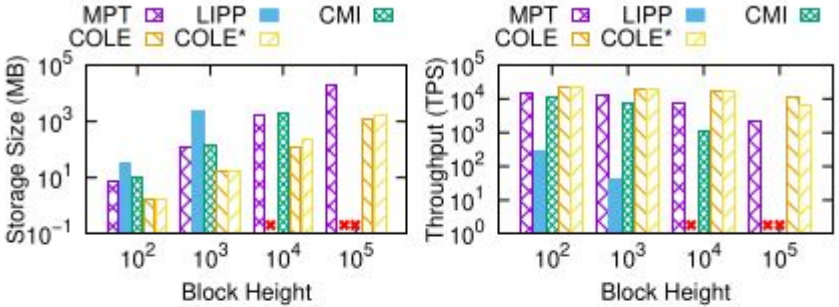


Figure 9: Performance vs. Block Height (SmallBank)

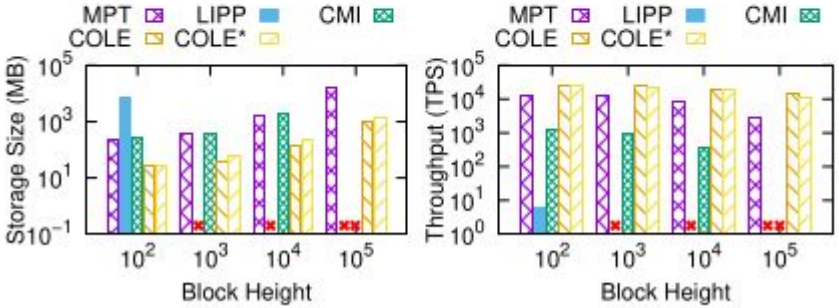


Figure 10: Performance vs. Block Height (KVStore)

Storage Reduction

BlockLSM	ChainKV	COLE
Modrate	Modrate	Up to 94%
through its prefix-based hashing and efficient compaction strategies.	by utilizing semantic-aware zones and efficient cache management.	COLE achieves significant storage reduction, up to 94%, through its novel column-based design and learned indexes.

Explanation

- 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.