The Comparison of COLE, ChainKV and BlockLSM

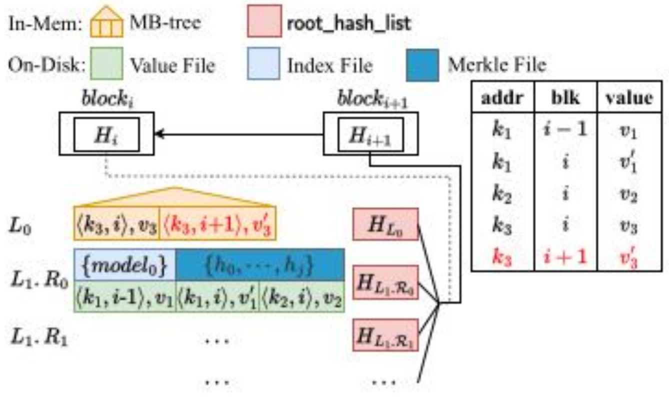
Data Storage Approach

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

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.

Components, Operation and Recovery (COLE)



addr	blk	value
k_1	$i-1$	v_1
k_1	i	v'_1
k_2	i	v_2
k_3	i	v_3
k_3	$i+1$	v'_3

Figure 3: Overview of COLE

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: Utilizes a root hash list for quick recovery, ensuring data integrity and consistency.

Components, Operations and Recovery (BlockLSM)

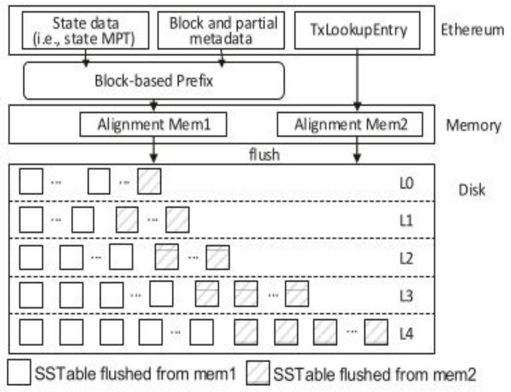

State data (i.e., state MPT)
Block and partial metadata
TxLookupEntry
Ethereum
Block-based Prefix
Alignment Mem1
Alignment Mem2
Memory
flush
Disk
L0
L1
L2
L3
L4
SSTable flushed from mem1
SSTable flushed from mem2

Fig. 3. Architecture overview of Block-LSM.

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.

Components, Operations and Recovery (ChainKV)

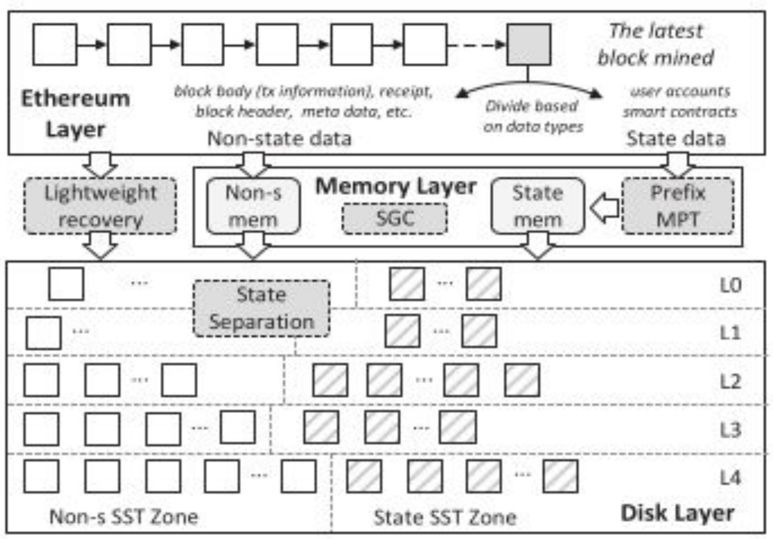


Fig. 4. The overall architecture of ChainKV.

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.

How the Zones in ChainKV are different From BlockLSM?

ChainKV: Semantic-Aware Zones

- Organizes data based on meaning and usage patterns.
- Optimizes storage and retrieval by understanding data nature.
- Groups related data semantically for efficient access.
- Maintains contextual relevance and proximity of related information.
- Improves query performance and reduces access times.

BlockLSM: Prefix-based Hashing

- Uses prefix-based hashing to organize data.
- Groups related data under common prefixes.
- Maintains data order and ensures efficient lookups.
- Reduces fragmentation and improves read performance.
- Enhances range query efficiency and overall performance by maintaining data locality.

Key Differences

- **Organizational Principle**:
  - ChainKV: Semantic meaning and usage patterns.
  - BlockLSM: Common prefixes.
- **Optimization Focus**:
  - ChainKV: Data access based on semantics.
  - BlockLSM: Data locality and order through prefix hashing.

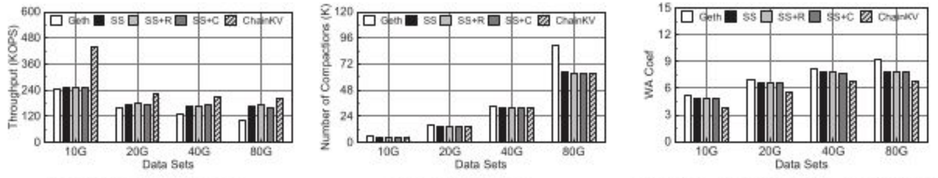
Compaction Reduction

BlockLSM	ChainKV	COLE
Yes	Yes	N/A

Explanation

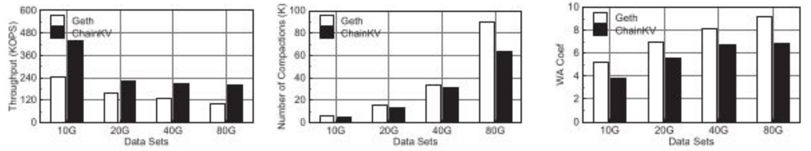
- BlockLSM includes a compaction process that reduces data redundancy and storage costs by periodically merging data blocks and eliminating obsolete data.
- ChainKV also implements compaction techniques to manage data more effectively, similar to BlockLSM, which helps in maintaining storage efficiency.
- 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.

Compaction Behavior in ChainKV



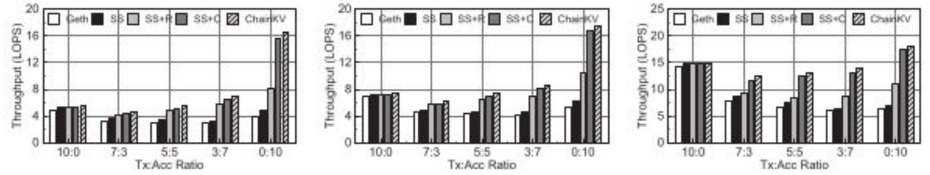
(a) Write Throughput. (b) Compaction. (c) Write Amplification (WA) Coef.

Fig. 9. Breakdown analysis of data writes.



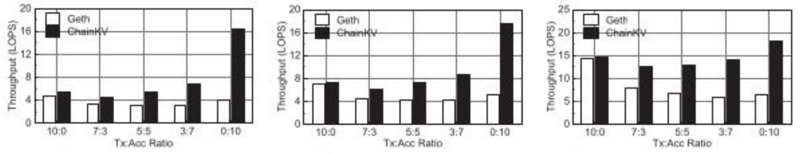
(a) Write Throughput. (b) Compaction. (c) Write Amplification (WA) Coef.

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



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

Fig. 10. Breakdown analysis of data reads.



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

Fig. 8. The overall performance comparison for data reads

Graphs Analysis

Write Throughput:

- ChainKV achieves high write throughput, especially with larger data sets, by optimizing compaction processes.
- The data shows consistent performance improvements in write throughput as data set size increases, similar to BlockLSM.

Number of Compactions:

- ChainKV reduces the number of necessary compactions through its semantic-aware zoning.
- The graph indicates that ChainKV performs fewer compactions than traditional methods, leading to reduced write amplification.

Write Amplification Coefficient (WA Coef):

- ChainKV maintains a low WA coefficient, minimizing data rewrite during compactions.
- The graph illustrates that ChainKV has a consistently lower WA coefficient, similar to BlockLSM, enhancing overall system efficiency.

Reasons

- **Semantic-Aware Zones**: By organizing data based on semantics and usage patterns, ChainKV optimizes compaction processes, reducing the need for frequent data reorganization.

- **Optimized Compaction Strategies**: ChainKV employs strategies tailored to its semantic-aware zones, ensuring compactions are performed efficiently with minimal write amplification.

Algorithm for Reduction

Semantic-Aware Zoning:

- ChainKV organizes data into semantic-aware zones based on the meaning and usage patterns of the data. This approach optimizes the storage and retrieval processes and reduces the frequency of compactions.

Level-Based Compaction:

- ChainKV uses a level-based compaction strategy similar to Log-Structured Merge (LSM) trees. Data is compacted from higher levels to lower levels, with each level having a larger capacity, thus reducing the overall number of compactions.

How it Works?

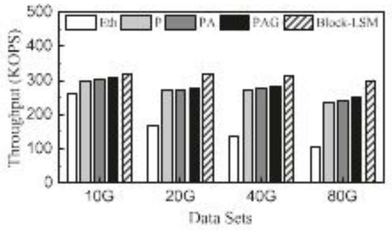
Semantic-Aware Zoning:

- 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.

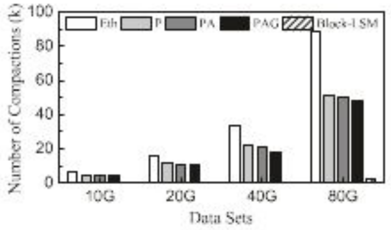
Level-Based Compaction:

- 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.

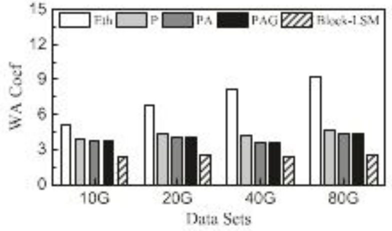
Compaction Behavior in BlockLSM



(a) Write Throughput.

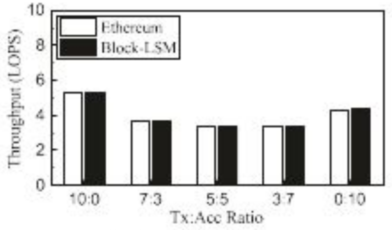


(b) Number of Compactions

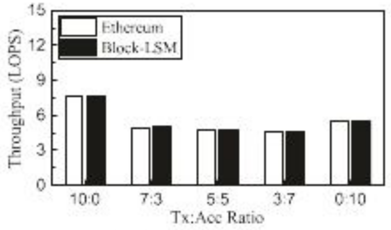


(c) WA Coef.

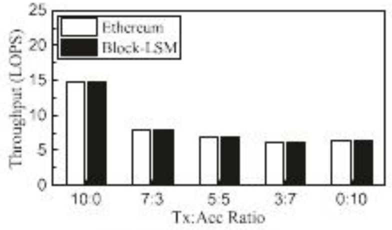
Fig. 8. Breakdown analysis of data synchronization.



(a) Normal Random.



(b) Scrambled Zipfian.



(c) Skewed Zipfian.

Fig. 9. The system read performance.

Graph Analysis

Write Throughput:

- BlockLSM shows higher write throughput compared to traditional systems due to its efficient compaction mechanism.
- The data in the graph shows that as the data set size increases from 10G to 80G, BlockLSM maintains a higher throughput, indicating effective compaction.

Number of Compactions:

- BlockLSM performs fewer compactions compared to other systems.
- The graph demonstrates that the number of compactions in BlockLSM is significantly lower across various data set sizes (10G, 20G, 40G, 80G), which reduces write amplification.

- **Write Amplification Coefficient (WA Coef)**:
  - BlockLSM exhibits a lower WA coefficient, meaning less data is rewritten during compaction.
  - The graph highlights that BlockLSM consistently maintains a lower WA coefficient, contributing to improved performance and reduced write costs.

Reasons

- **Efficient Compaction Algorithms**: BlockLSM uses advanced algorithms to minimize the frequency and cost of compactions.

- **Data Locality**: By maintaining data locality through prefix-based hashing, BlockLSM reduces the overhead associated with data rearrangement during compaction.

Algorithm for Reduction

Prefix-Based Compaction:

- BlockLSM uses a prefix-based hashing mechanism to group related data together under common prefixes. This approach helps maintain data locality, reducing the overhead of compaction processes.

Merge Sort Algorithm:

- BlockLSM employs a merge sort algorithm during compaction to merge sorted data efficiently. This reduces the frequency of compactions and minimizes the write amplification by keeping the data well-organized.

Write-Ahead Logging (WAL):

- Standard Write-Ahead Logging is used to ensure data integrity and consistency during compactions. This helps in reducing the overall write cost by logging only the changes instead of rewriting entire datasets.

How it Works?

Prefix-Based Hashing:

- 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.

Efficient Merging:

- 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.

Key Diffrences

Organizational Principle:

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

Compaction Frequency and Efficiency:

- **BlockLSM**: Fewer compactions, achieved through efficient algorithms and data locality maintenance.
- **ChainKV**: Reduced compactions via semantic-aware zoning, which minimizes unnecessary data reorganization.

Compaction Strategies:

- **BlockLSM**: Uses prefix-based hashing and merge sort algorithms to maintain data locality and efficiently merge data during compactions.
- **ChainKV**: Employs semantic-aware zoning and level-based compaction to optimize the compaction process based on the semantics and usage patterns of the data.

Impact on Performance:

- **BlockLSM**: Achieves fewer compactions and lower write amplification by maintaining data locality through prefix-based hashing and efficient merging.
- **ChainKV**: Reduces compactions and write amplification by organizing data semantically and employing level-based compaction, ensuring efficient and targeted compactions.

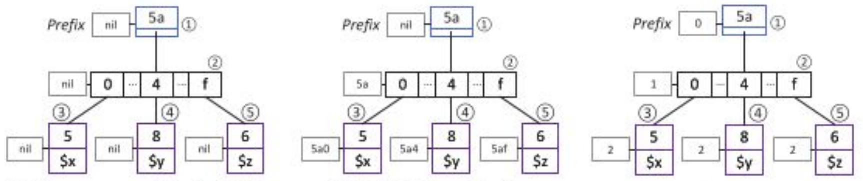
Data Locality

BlockLSM	ChainKV	COLE
Maintained via Prefixes	Preserved via Prefix MPT	Column Based

Explanation

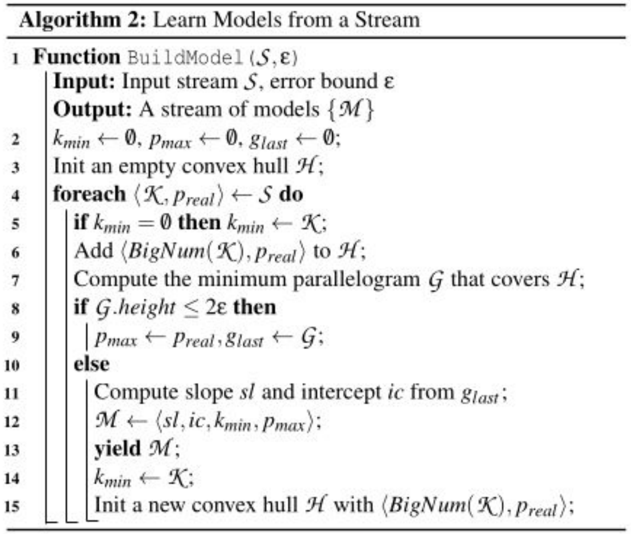
- Data locality in BlockLSM is maintained through the use of prefixes, which help group related data together, improving access speed.
- ChainKV preserves data locality by using a prefix-based Merkle Patricia Trie (MPT), which ensures that related data remains closely located.
- COLE improves data locality 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



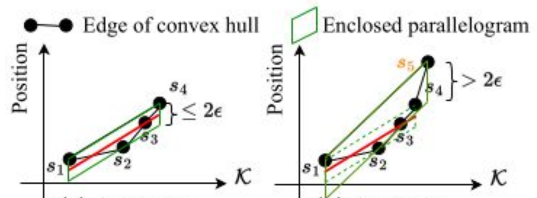
(a) The structure of Prefix MPT. (b) With Column Prefix. (c) With Row Prefix.

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

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



(a) Adding s_4 (b) Adding s_5

Figure 5: An Example of Model Learning

Cache Management

BlockLSM	ChainKV	COLE
Memory Buffers	Gaming Cache	Asynchronous Merges

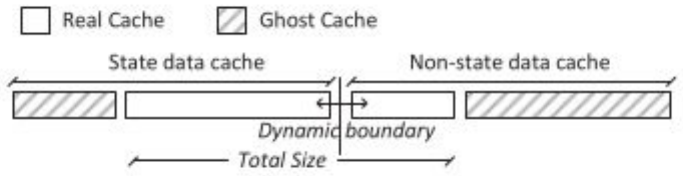


Fig. 6. The Structure of the SGC design.

VS

Algorithm 5: Write Algorithm with Asynchronous Merge

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

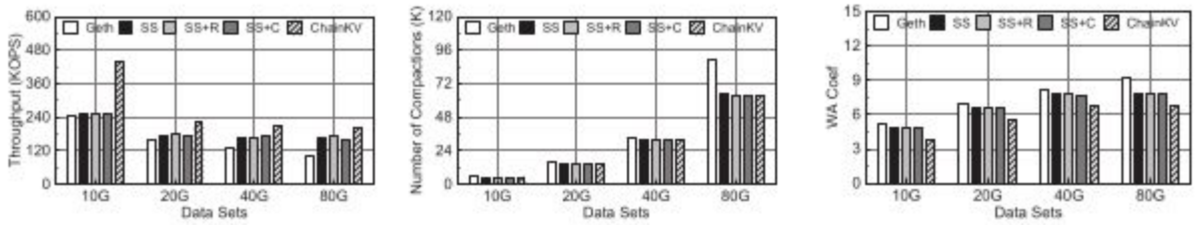
Explanation

- 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

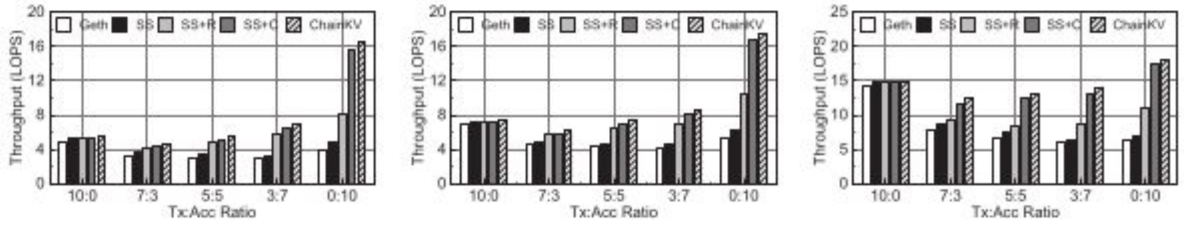
BlockLSM	ChainKV	COLE
Standard WAL	Lightweight Node Recovery	Asynchronous Merge Checkpoints

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.

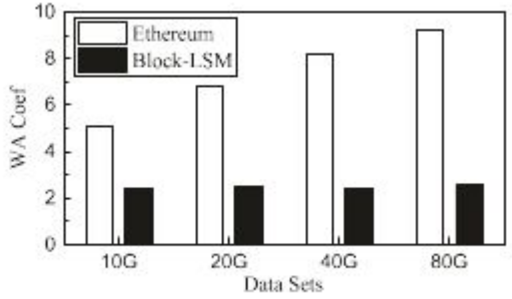
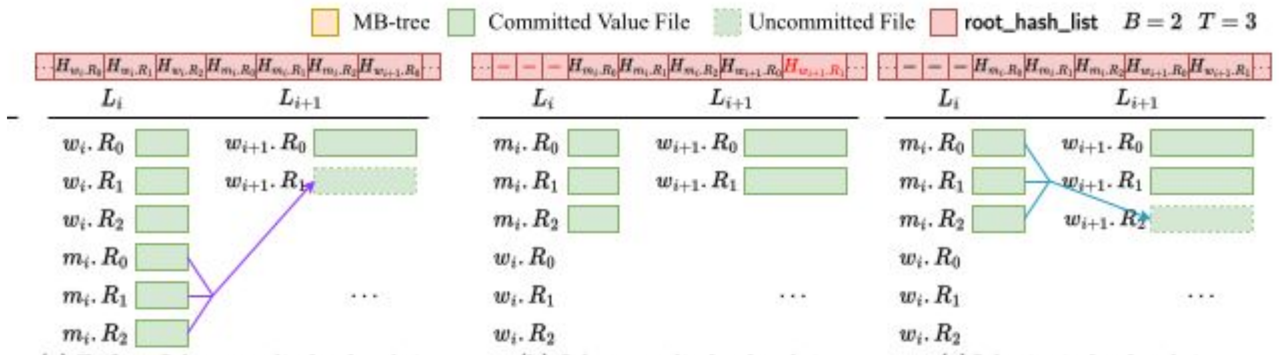


Fig. 6. The WA coef during data synchronization.

CheckPoints in COLE



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

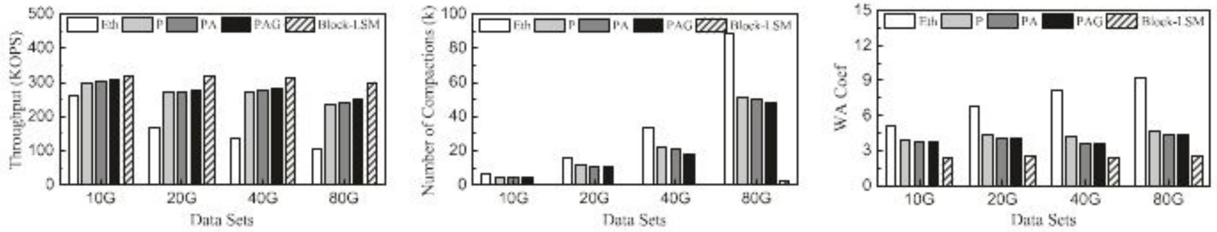
Explanation

- 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

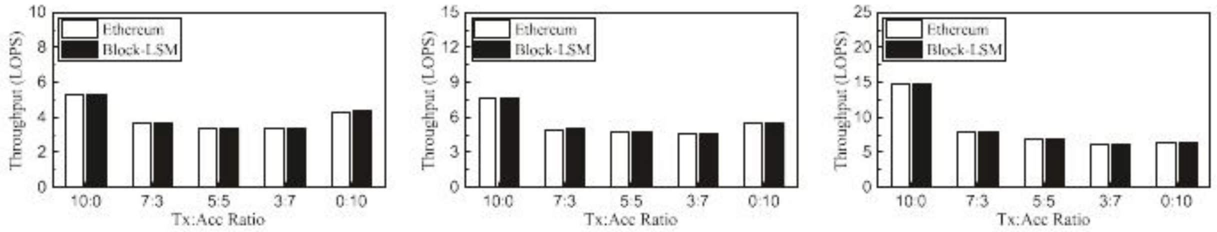
BlockLSM	ChainKV	COLE
Up to 182.75%	Up to 4.20x	Up to 5.4x
The performance improvement in BlockLSM comes from its efficient use of prefix-based hashing and compaction reduction. These mechanisms help to reduce write amplification and maintain data locality, leading to faster data retrieval and lower 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.

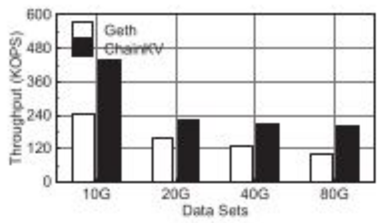
Fig. 8. Breakdown analysis of data synchronization.



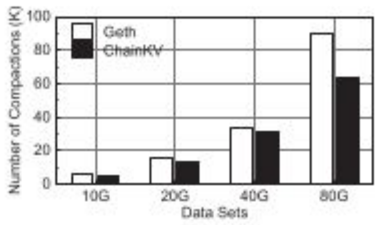
(a) Normal Random. (b) Scrambled Zipfian. (c) Skewed Zipfian.

Fig. 9. The system read performance.

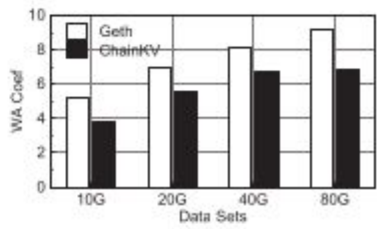
Read Performance in ChainKV



(a) Write Throughput.

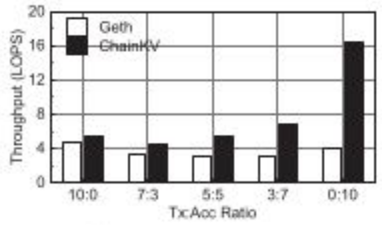


(b) Compaction.

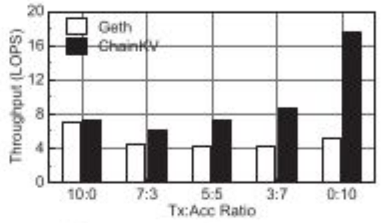


(c) Write Amplification (WA) Coef.

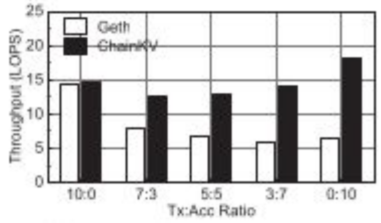
Fig. 7. The overall performance comparison for data writes.



(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

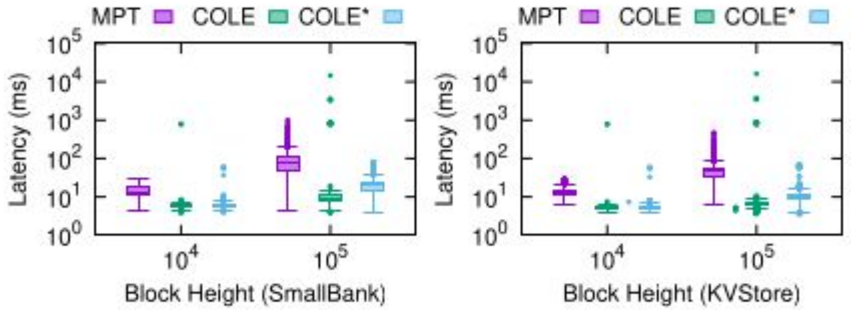


Figure 12: Latency Box Plot

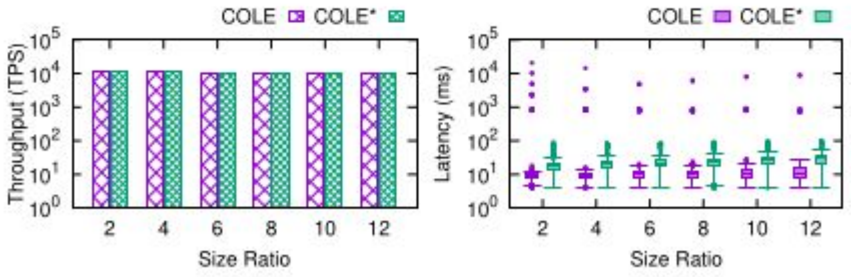


Figure 13: Impact of Size Ratio

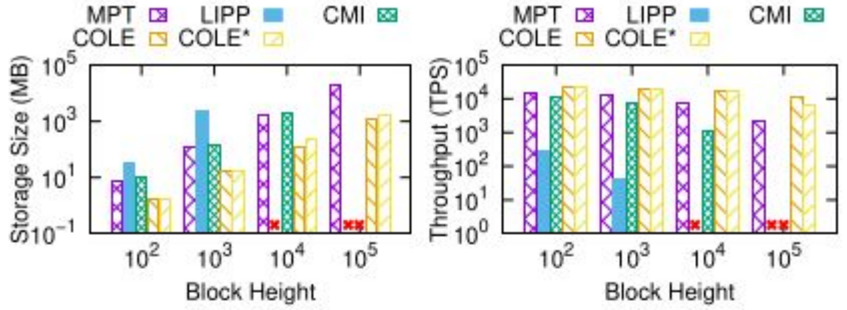


Figure 9: Performance vs. Block Height (SmallBank)

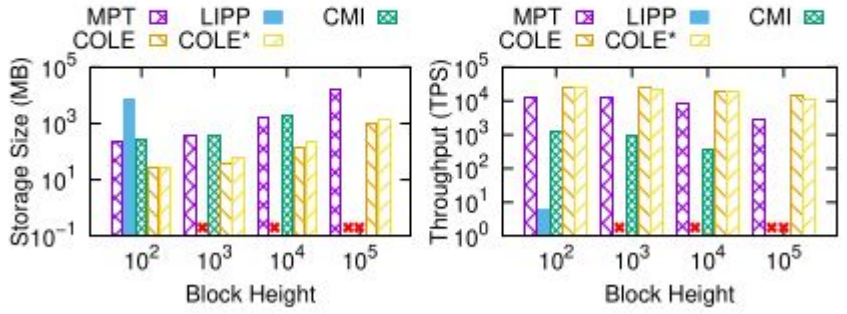


Figure 10: Performance vs. Block Height (KVStore)

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.