

MTree: A Tiering-based Key-Value Store Powered by High-performance Hierarchical Data Management

HUI SUN, computer science and technology, Anhui University, Hefei, China

YINHUI CHEN, computer science and technology, Anhui University, Hefei, China

YONWEI YU, Anhui University, Hefei, China

YAJIE DENG, Anhui University, Hefei, China

YINLIANG YUE, Zhongguancun Laboratory, Beijing, China

SONG JIANG, Electrical and Computer Engineering Department, The University of Texas at Arlington, Arlington, United States

XIAO QIN, Computer Science and Software Engineering, Auburn University, Auburn, United States

Key-value stores (KV stores) anchored on log-structured merge trees (LSM-tree) provide much improved performance under write-intensive workloads but exhibit significant write amplification (WA). The tiering compaction strategy is widely adopted in KV stores to reduce the WA. However, there are three imminent issues in tiering-based KV stores. First, excessive levels in the LSM-tree structure result in high read and write amplification. Second, without key partitioning support, these KV stores increase write tail latency. Relying solely on a hash-based key partitioning scheme is insufficient, as it does not support range queries. Furthermore, a KV store with only a key range-based partitioning scheme overlooks the distribution of keys within the key space. Third, existing KV stores with level optimization can incur high costs. A common approach to reducing the number of LSM-tree levels is to increase the MemTable size. While this reduces the number of levels, it requires more memory, leading to higher costs. Therefore, it is crucial to balance the tradeoff between level optimization and monetary expenses. To address these issues, we propose a tiering-based key-value store, leveraging high-performance hierarchical data management. The key space of our proposed KV store is partitioned into m multiple partitions based on key ranges, with an LSM-*tree* in each partition. We refer to this KV store as *MTree* in this article. *MTree* reduces the number of levels in the LSM-Tree structure close to one without increasing the monetary cost. In *MTree*, a hierarchical structure that combines the log and LSM-tree curtails the number of levels in the LSM-tree by accumulating the data in the log without increasing the cost. With the hierarchical structure in place, we devise a long- and short-term key density distribution-aware partitioning scheme for the key range. This scheme dynamically adjusts the size of

This work is supported in part by National Natural Science Foundation of China under Grants 62472002 and 62072001. Xiao Qin's work is supported by the U.S. National Science Foundation (Grants IIS-1618669 and OAC-1642133), the National Aeronautics and Space Administration (Grant 80NSSC20M0044), the National Highway Traffic Safety Administration (Grant 451861-19158), and Wright Media, LLC (Grants 240250 and 240311).

Authors' Contact Information: Hui Sun, computer science and technology, Anhui University, Hefei, Anhui, China; e-mail: sunhuiworking@gmail.com; Yinhui Chen, computer science and technology, Anhui University, Hefei, Anhui, China; e-mail: e21301222@stu.ahu.edu.cn; Yonwei Yu, Anhui University, Hefei, Anhui, China; e-mail: ywyu@stu.ahu.edu.cn; Yajie Deng, Anhui University, Hefei, Anhui, China; e-mail: yjdeng@stu.ahu.edu.cn; Yinliang Yue, Zhongguancun Laboratory, Beijing, Beijing, China; e-mail: yylhust@qq.com; Song Jiang, Electrical and Computer Engineering Department, The University of Texas at Arlington, Arlington, Texas, United States; e-mail: song.jiang@uta.edu; Xiao Qin, Computer Science and Software Engineering, Auburn University, Auburn, Alabama, United States; e-mail: xqin@auburn.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](https://permissions.acm.org).

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM 1553-3077/2025/11-ART40

<https://doi.org/10.1145/3736587>

partitioning, balancing data in the partition and storing most data in large-sized logs. Then, MTree reduces the number of levels in the LSM-tree, thereby optimizing the read/write amplification. The experimental results show that MTree limits the WA to as low as two. MTree enhances the random write throughput of the state-of-the-art KV stores by up to 6.9 $\times$.

CCS Concepts: • **Information systems** → **Storage management; Information storage technologies; Hierarchical storage management;**

Additional Key Words and Phrases: Key-value store, tiering, key density, write amplification, hierarchical data management

ACM Reference Format:

Hui Sun, Yinhui Chen, Yonwei Yu, Yajie Deng, Yinliang Yue, Song Jiang, and Xiao Qin. 2025. MTree: A Tiering-based Key-Value Store Powered by High-performance Hierarchical Data Management. *ACM Trans. Storage* 21, 4, Article 40 (November 2025), 40 pages. <https://doi.org/10.1145/3736587>

1 Introduction

Key-value stores have been widely used in storage engines of modern computers and distributed systems, such as caching systems [15, 18, 22], file systems [25, 45], task queues [2], and **online transaction processing (OLTP)** [32]. A **log-structured merge tree (LSM-tree)** adopts the log structure and compaction mechanism to achieve high performance under write-intensive workloads [41]. Thus, LSM-tree, being one of the predominant structures in KV stores, provides a solution to process large-scale data like BigTable [11] and RocksDB [20]. LSM-tree buffers write KV pairs in MemTable in memory—and periodically flushes immutable tables into disks. The **sorted string tables (SSTable)** in a disk are hierarchically organized, and SSTables at each level i ($i > 0$) are sorted. The data volume increases proportionally from a lower level to a higher one. Meanwhile, data moves to a higher level from a lower level through the compaction procedure until the data reaches the highest level and stabilizes. The LSM-tree introduces advantages to KV stores regarding high write performance, high space utilization, and efficient concurrency control.

However, the LSM-tree brings a high write-amplification issue. KV stores perform compaction operations to organize data and enhance query performance. Furthermore, a large number of compaction procedures cause data to be frequently rewritten in disks: the procedure consumes CPU cycles and I/O bandwidth, degrading the system's write performance. Especially for SSDs, frequent writes can shorten SSD lifetime. Therefore, optimizing the write amplification issue of the LSM-tree is of paramount importance. Two factors influence the amount of data involved in compaction. The first factor is the number of the LSM-tree levels (**H**) because compaction occurs at each level, resulting in data rewrites. The second one is the number of rewrites in a level (denoted as **R**), which is determined by the compaction strategy. Thus, (**H** $\times$ **R**) can be used to approximate the write amplification caused by the compaction process. Current state-of-the-art methods to reduce write amplification include tiering compaction strategy [16], key-value separation [10, 35], and fine-grained compaction [46]. In this study, we focus on tiering, which allows overlapping key ranges to exist at the same level in an LSM-tree. For example, PebblesDB [44] uses well-sorted guards to implement tiering, allowing overlapping key ranges within the SSTables of a guard. Similarly, LWC-store [52] employs well-sorted DTables for tiering, where the key ranges of data segments within DTables overlap. RocksDB [20] achieves tiering through sorted runs in its tiering compaction strategy. Although SSTables within a sorted run are fully ordered, key-range overlaps are permitted between sorted runs. In ALDC [9], lower-level SSTables accumulate partial fragments from the upper level that overlap with their key ranges before compaction. This approach reduces the amount of data rewritten during compaction. Though various tiering schemes have

been proposed, three interconnected problems remain to be addressed, which are elaborated as follows.

PROBLEM 1. *In a tiering-based KV store, read requests require a linear search through all key ranges at a given level due to overlapping key ranges. Bloom filters [5] and Cuckoo filters [21] are widely utilized in search operations to reduce lookup overhead. Furthermore, the maplets mechanism introduced in Mapped SplinterDB [13] can optimize this process. If the data is not found at this level, the search continues through other levels until the key is located or the largest level is reached. This process results in high read amplification, especially when there are numerous levels in an LSM-tree, thereby increasing the cost of Get/Scan operations [9, 44]. For instance, PebblesDB [44] implements tiering using well-sorted guards, but the key ranges of the SSTables within a guard can overlap. As a result, read requests must perform a linear search through each SSTable in the target guard at every level. With a 100 GB workload, PebblesDB loads data across five levels, and the presence of multiple levels contributes to higher read latency. SplinterDB [14] implements a tiering strategy using a well-sorted trunk node of B^e -tree, and key ranges of the branch within a trunk node can overlap. The Mapped SplinterDB [13] employs a quotient maplet to efficiently identify the subset of branches within a trunk node that may contain the target key. This approach eliminates the linear scans across branches with overlapping key ranges. But SplinterDB has many levels in B^e -tree. In ALDC [9], the lower-level trigger compaction strategy delays compaction, causing the upper level to accumulate multiple SSTables with overlapping key ranges. This also results in higher read latency when searching for the target key. Additionally, having many levels in KV stores can lead to high write amplification. Zhao et al. [57] try to deal with this issue by enlarging the size of in-memory components, which reduces the number of levels but increases monetary costs [36]. Considering the two factors (the number of the LSM-tree levels and the number of rewrites in a level), it is crucial to optimize the number of the LSM-tree levels for tiering-based KV stores, aiming to reduce read and write amplification.*

In addition, in LSM-trees, efficient data querying requires frequent compactions to reorganize SSTables and reduce overlapping key ranges. In contrast, key-value stores that separate keys from values adjust the indexing granularity to key-value pairs. This approach allows for direct data lookup in memory using the key, thereby eliminating the need for compaction to reorder data in disk. Consequently, write amplification is significantly reduced. The key can be stored in memory or in disk. For instance, Wiskey opts to store keys in disk, which introduces additional disk search overhead during read operations, thereby increasing read costs. Storing all keys in memory eliminates high read costs but causes large memory overhead.

Tiering-based KV stores exhibit high read and write amplification due to the increased number of levels, but they incur lower memory overhead. In contrast, key-value separation-based KV stores that use in-memory indexing can quickly locate data via keys in memory, resulting in lower read and write amplification but higher memory overhead. In this article, MTree combines the strengths of both approaches: the tiering-based KV store requires less memory, while the in-memory key-value separation index reduces the number of levels in the tiering-based KV store. We introduce BLog, a data structure composed of an internal memory component and an external memory component, designed to reduce the number of the LSM-tree levels without requiring additional memory, thereby avoiding high read latency.

PROBLEM 2. *A handful of tiering-based KV stores [16, 20, 38, 42] rarely provide a key partitioning scheme, thereby leading to a large data volume involved in a compaction procedure coupled with a high write tail latency. For example, Dostoevsky [16] and LSM-Bush [38] use sorted runs to achieve tiering. While the SSTables in sorted runs are ordered, the overlapping key ranges between sorted runs are permitted. Each tiering compaction operation involves the entire level, resulting in significant write tail latency. dCompaction [42] employs a delayed compaction strategy to achieve tiering. At each level,*

SSTables with overlapping key ranges undergo virtual compaction, while real compaction involves a large number of SSTables, resulting in high write tail latency. The hash-based key partitioning scheme [49], cutting back data volume in a compaction procedure, is independent of the key distribution in the key space and does not favor scan operations. The key partitioning scheme controls the data volume involved in both a single flush and a single compaction by distributing keys across different partitions. For example, LSM-trie [49] employs a hash-based key partitioning strategy, where the HTable in the key partitioning allows overlapping key ranges. The hash-based key partitioning strategy causes adjacent keys to be hashed into different partitions, which does not support scans. In contrast, our proposed MTree offers a key range partitioning scheme for tiering, which is conducive to range queries and helps avoid long-tail delays.

PROBLEM 3. *KV stores powered by tiering using key range partitioning schemes rarely consider short-term key-density volatility [52, 57] in different partitions of the key space—that enlarges the number of levels in the LSM-tree. For instance, LWC-store [52] employs a well-sorted DTable for tiering, where key ranges do not overlap. However, DTable cannot adapt key ranges to short-term changes in key density, which may result in an increased number of the LSM-tree levels. Similarly, WipDB [57] employs a well-sorted DTable to achieve tiering, with overlapping key ranges in SSTables within a bucket. Since the memory buffer size for each bucket is identical, the amount of written data remains nearly constant. Consequently, the memory cache space in any bucket cannot adapt to short-term changes in key density, thereby leading to an increased number of the LSM-tree levels. Key-density volatility entails that the key density of a partition changes over time. In addition, some of these KV stores ignore the long-term key-density stability [23] across partitions within the key space, resulting in imbalanced partitioned data and a performance cliff. For example, PebblesDB’s guard selection strategy is associated with the Murmur hash algorithm. Thus, the data distribution in each guard can be imbalanced over time, which can negatively impact the stability of the system’s performance. LightKV [23] applied partitions to achieve tiering, and the SSTables in a partition have overlapping key ranges. The key ranges of these partitions are rooted in the access pattern from the previous stage of the workload and remain unchanged, causing data volume imbalances across partitions. Key density for a partition is defined as the ratio of the number of write keys in that partition to the total number of write keys in the key space over a period. In MTree, we consider the key distribution within the key space and leverage the long-term stability of key density to balance the data volume in each bucket. Once the data volume in a bucket reaches a preset threshold, a bucket splitting operation is triggered to dynamically adjust the key ranges of each bucket to align with the key distribution in the key space. Therefore, MTree avoids the system’s performance degradation. We also dynamically adjust the BLog size of each partition according to the short-term key density fluctuations, thus reducing the number of levels in the LSM-tree and lowering read/write amplification.*

We reckon that a tiering-based KV store should satisfy the following conditions: (1) reduced levels in the LSM-tree, (2) a cost-effective key partitioning scheme supporting scan operations, and (3) dynamic partition-size adjustment according to the key distribution in key space.

In this article, we propose a new KV store named *MTree* to address the challenges in tiering-based KV stores. *MTree* trims the number of levels in an LSM-tree to around one without incurring monetary cost, lowering read/write amplification. In addition, *MTree* provides a suitable key partitioning scheme that matches the key distribution in a key space, thereby achieving high write performance. In *MTree*, we propose an internal and external memory data organization—BLog, in which the index information of each KV pair is registered in memory using a B^+ -tree structure. The footprint of the total B^+ -tree has the same size as that of a MemTable. Written data is buffered in the external memory using a log to accumulate data recorded to the LSM-tree maximally. Then, we employ the key-range partitioning scheme, in which each partition is referred to as a bucket

that contains a small BLog and LSM-tree to support range query and slash data volume involved in a compaction procedure. Data volume in each bucket can be balanced according to the long-term key-density stability of workloads. BLog and LSM-tree comprise a hierarchical partitioning structure, in which we focus on two aspects within a bucket: (1) the size ratio of the capacity of neighboring levels in the LSM-tree and (2) the size ratio between the BLog and LSM-tree in a bucket. In terms of the size ratio of the capacity of neighboring levels in the LSM-tree, tiering prevents data from being rewritten at the same level. In relation to the size ratio of the capacity between BLog and LSM-tree, we apply the short-term key-density volatility to dynamically adjust the size of BLog in each bucket in real time. Different-sized BLogs can store data at various levels in the LSM-tree, thereby reducing the number of levels and optimizing read/write amplification. The contributions of our work are tabulated below.

- We propose a new KV-store based on a hierarchical partitioning of the LSM-tree and BLog. This approach reduces the number of levels in the LSM-tree and provides a key partitioning scheme for tiering that favors range queries.
- We not only consider the key distribution in the key space and long-term key-density stability, but also utilize bucket splitting to balance the data volume in each bucket, aiming to avert a performance cliff.
- We thoroughly leverage the short-term key-density volatility to adjust each partitioned BLog size in a real-time manner, thereby curtailing the number of levels in the LSM-tree and achieving low read/write amplification.
- We implemented MTree based on HyperLevelDB in the experiment. The evaluation results show that MTree outperforms the existing tiering-enabled KV stores (such as RocksDB-tiering and PebblesDB) by $3.0\times$ and $6.9\times$ in terms of throughput under random-write workloads. MTree presents the lowest 99th percentile latency compared with alternative KV stores under write-intensive workloads.

The rest of the article is organized as follows. Section 2 presents background. We illustrate motivation in Section 3. In Section 4, we elaborate on the design and implementation of MTree. We present our experimental setup in Section 5. In Section 6, we evaluate MTree's system performance. Section 8 discusses the related work. Finally, we conclude the article in Section 9.

2 Background

We introduce the Log-Structured Merge tree and index structures in this section.

2.1 Log-Structured Merge Tree

Log-structured merge tree or LSM-tree [41], a persistent data structure, converts random-write operations into sequential ones to leverage the high-sequential write performance of disks. LSM-tree-based key-value stores or KV stores have become popular in cloud computing. The performance and reliability of KV stores have been enhanced by optimizing data structures, algorithms, and architectures. LSM-tree consists of a memory component (C_0) and multiple disk-level components ($C_1, C_2, \dots$, and C_k), as demonstrated in Figure 1(a). The data is first written to a sorted data structure (C_0) in memory. When the growing C_0 tree first reaches its threshold size, C_0 is merged with C_1 on a disk—and the result is written into C_1 . As data volume increases, the data in C_1 is further merged and moved to higher-level disk components ($C_2, \dots, C_k$). With this level-by-level merging, the LSM-tree optimizes write performance and disk I/O efficiency, while improving query performance through periodic-merge operations.

KV stores based on LSM-tree, such as LevelDB [33] and RocksDB [20], have been widely used in academia and industry. In RocksDB, that is depicted in Figure 1(b), MemTable is adopted to manage

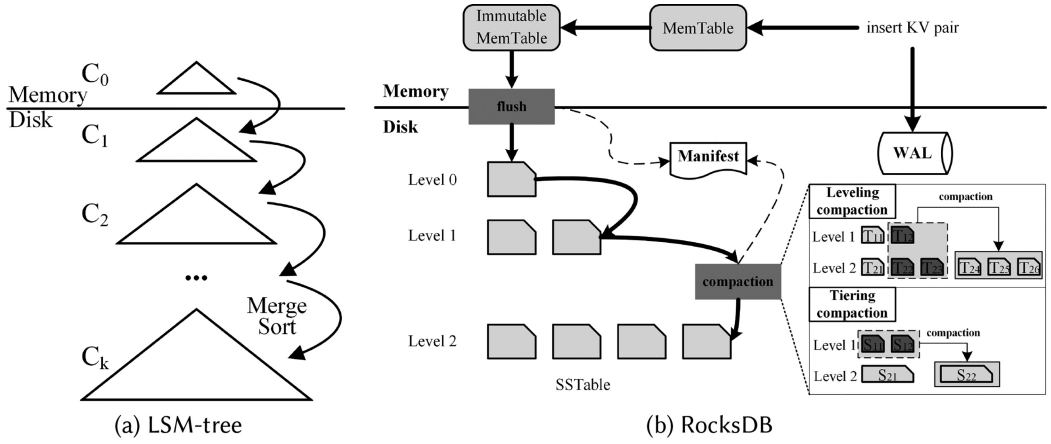


Fig. 1. Architecture of the LSM-tree and RocksDB.

the write KV pairs in DRAM, and its indexing structure is implemented via a skip list—that can update the corresponding pointers to insert and delete data. When a MemTable size reaches a threshold, the MemTable will transform into an Immutable. Then, the Immutable is flushed to the disk, generating a sorted string table or SSTable or *SSTable*. SSTables are managed in disks in a hierarchical-level manner, where data at each level is sorted, and the data volume proportionally increases with each level up. When data volume at a level hits its threshold, a major compaction procedure will be invoked to transfer the new SSTable to a high level. The LSM-tree favors insert, update, delete, and scan operations. The compaction procedure sorts data and collects invalid data; however, frequent compaction operations result in high write amplification.

Generally, two factors affect the data volume involved in a compaction procedure. First, the height H of an LSM-tree is closely related to the capacity size ratio R of the two adjacent levels, and the data is rewritten at each level in compaction. If the total amount of data written to the LSM-tree is D , and the data volume at the lowest level is A , H is equal to

$$\left\lceil \log_R \left(\frac{D}{A} \times (R - 1) + 1 \right) \right\rceil.$$

Second, a compaction strategy affects the number of rewrites within the same level. Compaction strategies include leveling compaction and tiering compaction. A leveling compaction strategy achieves complete data sorted at the same level: data movement involves two levels. For example, when a compaction operation is triggered, SSTables at level L_i and SSTables with overlapping key ranges at level L_{i+1} are selected to perform sorting. Then, new SSTables are written back into level L_{i+1} . In this case, the data is rewritten R times at the same level before being moved to the highest level, where the data is written $O(H \times R)$ times—a phenomenon that is referred to as write amplification. Given the values of D and A , the write amplification is a function of the capacity size ratio R , and it has a relative minimum at some value of R . In an LSM-tree with leveling compaction, the write amplification cannot be decreased below a certain level. The tiering compaction strategy allows overlapping sorted runs at the same level [16, 41], where merge operations usually occur. When compaction is triggered, all the sorted runs at level L_i will be sorted. Then, new SSTables are appended to level L_{i+1} . The data are rewritten only once at the same level, and write amplification reduces to $O(H)$. During a read, up to $O(H \times R)$ sorted runs may need to be read. In the worst case, each sorted run needs to be checked once, which enlarges the read overhead. The data volume

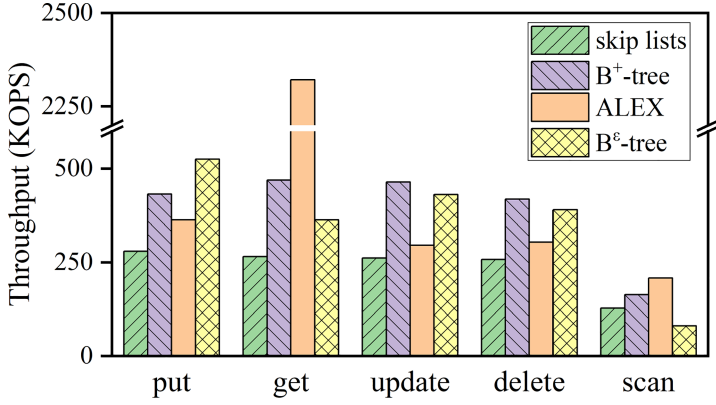


Fig. 2. In-memory throughput of skip lists, B⁺-tree, ALEX and B^ε-tree.

involved in a compaction procedure is too large, which leads to disk I/O bottlenecks and impairs write performance.

2.2 Index Structures for Key-value Pairs

The in-memory index structure deployed in large-scale KV stores affects query performance. We need to focus on the performance of basic operations and memory overhead in the design of in-memory index structures. These basic operations supported by KV stores are put, get, update, and delete. A large amount of indexed data induces memory overhead of indexes and impacts the performance of basic operations. Then, it is challenging to use skip lists as suitable index structures for large-scale KV stores due to high memory overhead [54]. Besides, scan operations are increasingly becoming popular in KV store applications. For example, e-commerce companies analyze user behaviors by querying all sales data [31]. The machine learning algorithms with the k-NN [28] and k-means [30] algorithms frequently read all entries in a KV store and calculate their distances to each other. The index structure, a vital technique in the KV store, should favor scan operations. Existing index structures include skip lists [43], B⁺-tree [12], B^ε-tree [7], learned index [29] and hash [3]. However, hash indices fall short of supporting scan operations.

The B^ε-tree is a data structure optimized for write-intensive application scenarios. It improves write performance by introducing buffers and delayed write techniques into the nodes of a B-tree. Compared to B⁺-tree, the B^ε-tree is more suitable for scenarios that require handling bulk write operations. However, the cost of faster writes is the additional overhead incurred by queries to process the buffer, leading to slower query performance. Learned index is an emerging database indexing technique that applies machine learning models to database indexing to replace or enhance traditional data structures. The core idea of learned indexes is to leverage data distribution characteristics and train a model to predict the location of data, thereby accelerating lookups and access. If the data distribution follows a clear pattern (such as time-series data, incrementing IDs), learned indexes can efficiently capture these patterns and thus speed up lookups. Since new inserts in learned indexes can affect the accuracy of the trained model, learned indexes are best suited for scenarios where data is not frequently updated. If data is frequently updated, retraining the model may be necessary, which introduces significant time overhead. To support better updates, XIndex [48] introduces buffers to absorb new writes, FINEdex [34] adds fine-grained extensible data structures at insertion points, and ALEX [19] leaves empty slots in its gap array to absorb writes. Furthermore, we perform micro-benchmarks on skip lists, B^ε-tree, ALEX, and B⁺-tree to quantitatively evaluate their efficiency under a wide range of operations. Figure 2 shows the in-memory

throughput results with 10 million KV pairs. B^+ -tree delivers 1.54 $\times$, 1.77 $\times$, 1.77 $\times$, 1.62 $\times$, and 1.28 $\times$ higher throughput than skip lists for put, get, update, delete, and scan operations, respectively. These results confirm that B^+ -tree has higher performance over skip lists in these operations. In addition, we count the footprint of skip lists and B^+ -tree under the micro-benchmarks, discovering that there is a slight difference between the two counterparts. The footprint of an index structure is comprised of the cost of storing keys and location information. In skip lists, a node stores a single KV pair, whereas B^+ -tree saves memory cost by storing multiple KV pairs in a node. ALEX achieves significantly higher throughput than the B^+ -tree for both get and scan operations, with improvements of 4.9 $\times$ and 1.26 $\times$, respectively. However, B^+ -tree outperforms ALEX in put, update, and delete throughput, by 1.19 $\times$, 1.57 $\times$, and 1.37 $\times$, respectively. ALEX enhances query performance by leveraging a learned model to predict data locations, which reduces the lookup path length. However, since ALEX reserves free space to absorb new inserts, its space overhead is larger, with memory consumption nearly double that of the B^+ tree. In comparison, the B^+ -tree also demonstrates higher throughput than the B^ϵ -tree for get, update, delete, and scan operations, with respective improvements of 1.29 $\times$, 1.07 $\times$, 1.07 $\times$, and 2.0 $\times$. On the other hand, the B^ϵ -tree benefits from its buffer, offering 1.21 $\times$ higher put throughput than the B^+ -tree. However, this buffer also introduces additional query overhead for the B^ϵ -tree compared to the B^+ -tree. In short, ALEX is best suited for read-heavy workloads, while the B^ϵ -tree excels in batch insertion scenarios. The B^+ -tree strikes a balance, delivering relatively strong performance across various scenarios.

3 Motivation

To evaluate the drawbacks of the tiering-based KV stores, we deploy RocksDB with the tiering compaction strategy on a DRAM-SSD structure. In this study, we configure a workload with a 100-GB dataset, 16-B keys, and 1-KB values. The experimental environment and parameters are outlined in Section 5. We obtained three intriguing findings as follows.

Finding 1: Excessive levels in the LSM-tree lead to high read/write amplification. The tiering-based KV stores allow overlapping key ranges at the same level, which lessens data rewrite, but the read operation must perform a linear search of all overlapping key ranges in each level to find the target KV pair. Moreover, the number of levels in the LSM-tree is an amplification factor: increasing the number of levels is a culprit of a high get/scan cost. In addition, data movement between levels still rewrites data, leading to write amplification.

In RocksDB, we set up various numbers of levels to study its performance under the 100-GB random-write workloads. We set the number of levels of the LSM-tree to at least four to store similar data volume in the highest level of the LSM-tree. Figure 3 illustrates that the write, get, and scan throughput increases as the number of levels decreases. The throughput of four-level RocksDB is 7.3%, 11%, and 30% higher than those of the 7-level RocksDB with respect to put, get, and scan, respectively. The more levels RocksDB has, the worse its write, read, and scan performance becomes, especially for scan operations.

To mitigate the issue of the excessive number of levels, we propose a BLog structure to curb the number of levels. The BLog uses an in-memory B^+ -tree to record the index of each KV pair—and the log in external memory is applied to buffer writes, accumulating written data to the LSM-tree.

Finding 2: A handful of tiering-based KV stores omit a key partitioning scheme, thereby giving rise to a high write tail latency. RocksDB employs a tiering compaction strategy to merge small-sized sorted runs into large-sized ones rather than furnishing a key partitioning scheme. The results show that the lack of a key partitioning scheme is a leading factor behind high write tail latency and unstable write performance. Table 1 unveils the average, 75th, 90th, 99th, and 99.9th percentile write latency (us) of RocksDB under the workloads: the average write latency is 24.079 us, but the 99th percentile latency is 43.03 times the average latency—the results entail that the

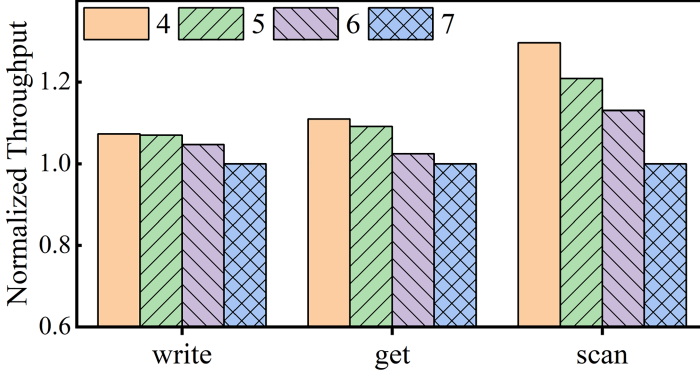


Fig. 3. Put, get, scan performance of RocksDB with tiering_compaction strategy. We configure different numbers of levels in RocksDB.

Table 1. Write Latency of RocksDB with Tiering Compaction under Workload with 100-GB Writes

Latency (μ s)	avg.	75%	90%	99%	99.9%
RocksDB-tiering	24.1	7.0	10.7	1,036	1,275.2

prolonged response time of a few write requests enlarges the overall response time and impairs the **quality of service (QoS)**.

Two primary causes contribute to extended write tail latency. The first one is the flush operation because the performance of writing data to memory is faster than that of flushing data from memory to disks. Situations can arise where the in-memory data buffer is full while the flush operation is still in progress—and user write requests are delayed, contributing to high write tail latency. The second one is the compaction operation. The tiering compaction strategy lacks a key partitioning scheme, resulting in a massive data volume in a single compaction procedure—that consumes excessive CPU cycles and I/O bandwidth, hindering flush performance with high write-tail latency.

We advocate for a bucket structure to devise an efficient key partitioning scheme in the tiering compaction strategy to trim data volume involved in a single compaction procedure. In addition, we design a dynamic allocation algorithm for BLog to alleviate the long-tailed latency by dynamically adjusting the BLog capacity when a flush operation blocks write requests.

Finding 3: KV stores powered by tiering with key range partitioning often overlook key distribution within key space, ignoring both long-term key-density stability and short-term key-density volatility across partitions. In this study, we configure four types of workloads: random-write workloads with Uniform, Zipfian, latest distribution, and sequential-write workloads in YCSB [51] with 100 million write requests. We divide the key space into 500 key-range partitions and register the key density of these partitions over time. We measure the key density every 5 seconds, including both the key density of the data written within those 5 seconds and the key density of the entire key space. Our experimental results show that the mean short-term key density of the written data across the 500 key partitions within 5 seconds ranges from 0.014 to 0.025, with a variance ranging from 3.1×10^{-7} to 9.2×10^{-7} . The long-term key density mean across the 500 key partitions within the entire key space is in the range of $[0.014, 0.025]$, with a variance between $[2.2 \times 10^{-9}, 8.7 \times 10^{-9}]$. The short-term key density mean is much more volatile than the long-term one, as evidenced by the variance. Figure 4 plots the key density fluctuations for four randomly selected partitions out of 500, which reveals similar key density variations regardless of

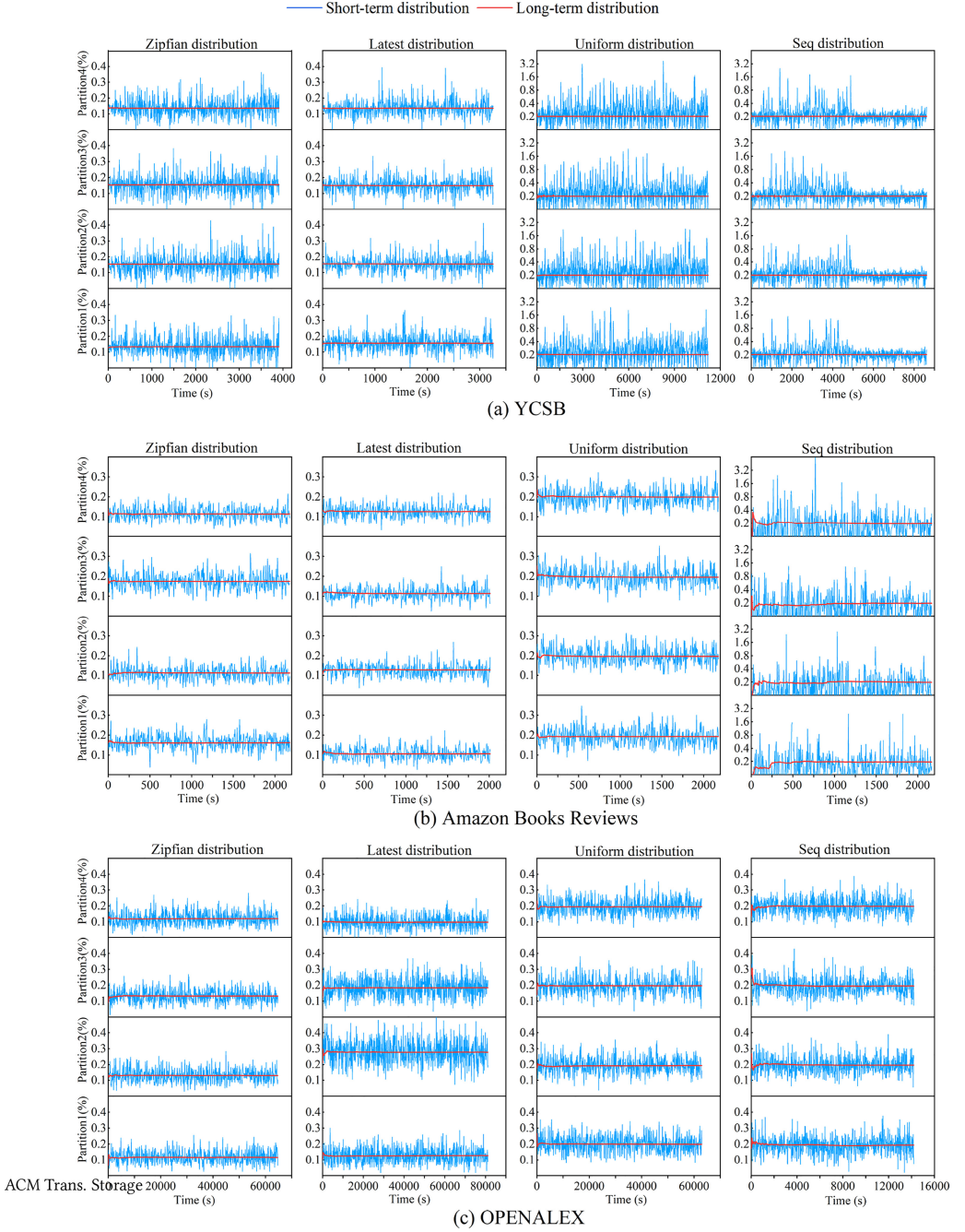


Fig. 4. The key distribution of RocksDB under four different key distributions: Zipfian, Latest, Uniform, and Sequential across various datasets. The blue line represents the key density measured over the short term (5 seconds for YCSB and Amazon Books Reviews, 20 seconds for OPENALEX), while the red line represents the average key density.

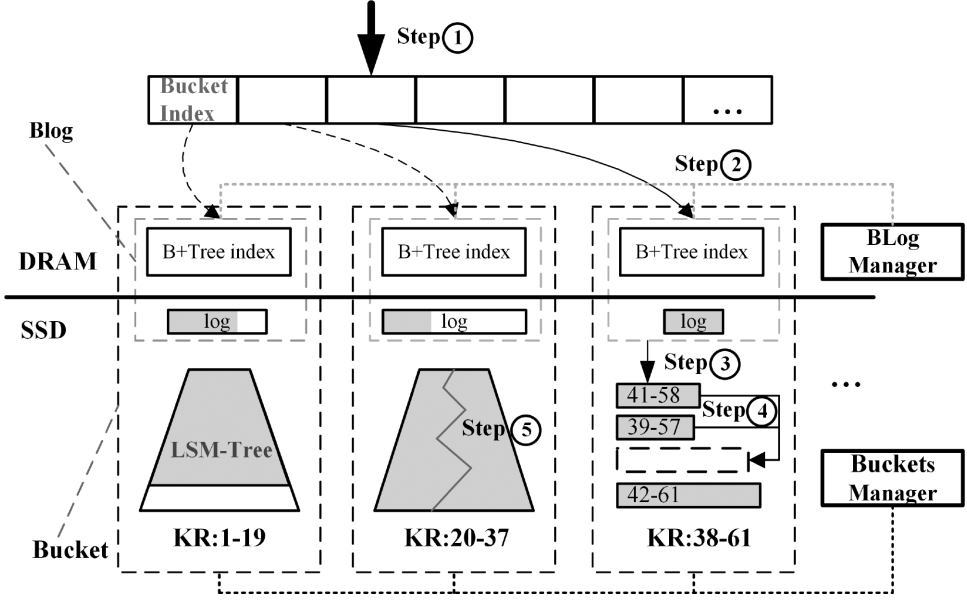


Fig. 5. The overview system of MTree. KR denotes a key range in which the keys of objects in a bucket fall.

the key distribution workload or partitions. We observe an oscillating pattern despite the average key density remaining stable. Short-term key-density volatility can lead to a lower-level key range containing a larger data volume than a higher level. Level-by-level compaction tends to exacerbate write amplification, a common issue often overlooked in existing KV stores. The long-term key-density stability, however, allows for effective regulation of data volume per bucket, preventing performance cliffs caused by data imbalance. We also apply a similar approach to examine the experimental results on datasets OPENALEX [4, 40] and Amazon Books Reviews [24], which exhibit data distribution patterns similar to those described above. Specifically, the OPENALEX dataset has a total size of 100.8 GB, containing 5.75 million records. Each record ranges from 638 B to 50,448 B, with an average size of 18,657 B. We execute 25 million write requests under Zipfian, Latest, and Uniform key distributions, and 5.75 million write requests under the Sequential distribution. The Amazon Books Reviews dataset is 20.1 GB, consisting of 29.5 million records, each ranging from 164 B to 57,922 B, with an average size of 715 B. We execute 25 million write requests under Zipfian, Latest, and Uniform key distributions, and 29.5 million write requests under the Sequential distribution. The key space of both datasets is divided into 512 key-range partitions.

We propose a bucket-splitting strategy, being aware of key distribution within the key space, to balance partitioned data by virtue of the long-term key-density stability. In addition, we design a size-adaptive BLog structure and a cross-level write method based on the LSM-tree to address short-term key-density volatility.

4 The Design of MTree

4.1 Overview System

In this section, we introduce our proposed MTree—a new hierarchical partitioned KV store based on the LSM-tree and BLog. Figure 5 sketches an overview system of MTree. The upper level is a BLog structure, which consists of a B⁺-tree index in memory and a log in the SSD for receiving newly written KV pairs. The lower level is an LSM-tree structure for receiving data accumulated by

the BLog. The bucket manager partitions the key space into multiple buckets based on long-term key-density stability. The BLog manager dynamically adjusts BLog capacity in each bucket according to short-term key-density volatility. MTree achieves high performance through the following four modules. The BLog structure reduces the number of levels in the LSM-tree (Section 4.2). The bucket structure and the BLog→LSM-tree operation (Section 4.3) facilitate key partitioning based on key ranges. The bucket splitting technique (Section 4.4) ensures that buckets store a similar data volume. The dynamic BLog technique (Section 4.5) allocates the capacity of BLog in each bucket.

MTree supports all basic KV store operations, including *put*, *delete*, *get*, and *scan*. Figure 5 illustrates that the *put* operation embraces four steps.

- Step ①: We locate the bucket with the corresponding key range of the inserted KV pair. The bucket index is an array that registers the smallest key and bucket location.
- Step ②: The BLog in this bucket is responsible for receiving new inserted KV pair.
- Step ③: When the BLog is filled, the BLog→LSM-tree operation is triggered to write the data into the LSM-tree as a sorted run.
- Step ④: When a level in the LSM-tree is filled, a compaction operation is invoked to merge all the data in that level and place it in the next level.
- Step ⑤: When the LSM-tree in this bucket is filled, a bucket-splitting operation is triggered.

A *delete* operation marks a removed KV pair with a null value. A *get* operation first locates the bucket containing the target item's key, then searches within the BLog of the bucket. If the target item is not found in the BLog, MTree searches all sorted runs at the level. Bloom filters within each sub-level help avoid most, if not all, unnecessary access to files that do not contain the requested key. Similarly, the Bloom filter in each sorted run prevents accessing most files that lack the target key. If the target item is not found at the level, MTree searches the next level until the target item is located or the last level of the LSM-tree in the bucket. A *scan* operation necessitates accessing all BLogs and sorted runs in the specified range.

4.2 BLog

In the LSM-tree, a small number of levels can mitigate write amplification. Two methods are commonly used to achieve this goal: (1) increasing MemTable size and (2) adopting KV pairs rather than blocks as the indexing granularity for the key space. Both methods, however, incur high memory overhead. To balance the memory overhead and write amplification, we adopt a BLog structure atop an LSM-tree. The BLog uses an indexing granularity of a KV pair and has a size approximately one-tenth that of an LSM-tree. This design decreases write amplification while reducing memory overhead. In addition, using key-value pairs as the indexing granularity ensures that dirty data indexes are not recorded. As a result, only clean data is written to the LSM-tree, preventing space amplification. Figure 6 depicts the BLog structure, which consists of a log holding data on the SSD and a B⁺-tree that stores in-memory index of each KV pair. The B⁺-tree index structure relies on a small amount of memory to locate data in real time, thereby offering efficient query operations. Aiming to compress the number of levels in the LSM-tree, the log structure accumulates data written to the LSM-tree. We elect B⁺-tree as the data structure for storing the indexes because B⁺-tree exhibits remarkable read and write performance (see Section 2.2). Moreover, as memory is expensive, we articulate a way of curbing the memory overhead of B⁺-tree in this section. The BLog facilitates *put*, *delete*, *get*, and *scan* operations.

Insert. As shown in Figure 6, inserting a KV pair embraces four steps.

- Step ①: The inserted KV pair is written to the **write-ahead log (WAL)**.
- Step ②: MTree first identifies the BLog to which the inserted KV pair belongs. Then, MTree writes the data into an in-memory buffer (e.g., a 4-KB block) and records the corresponding

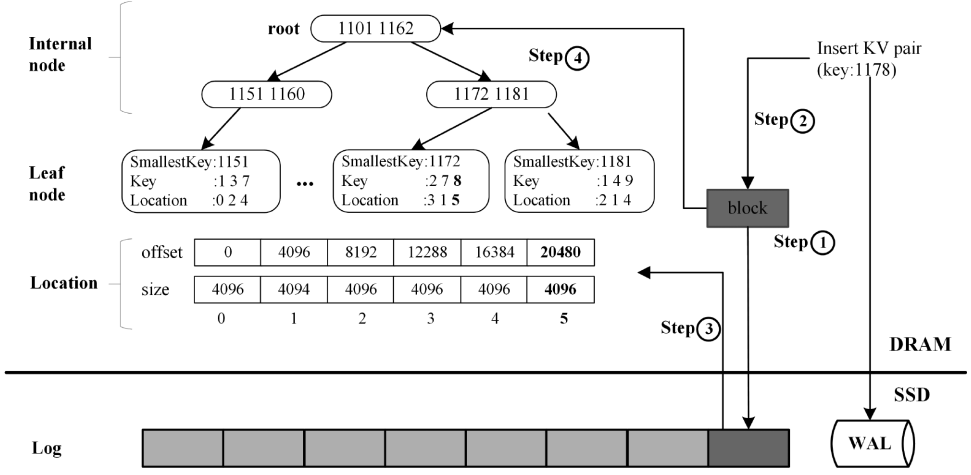


Fig. 6. Structure of BLog. Internal nodes store keys and pointers to leaf nodes, and leaf nodes store key and position information for KV pairs.

key. Please refer to Section 4.6 for an approach to handling crash recovery consistency of the buffer.

- Step ③: When data volume in the buffer reaches the threshold, the data is flushed to the log in the SSD—and we store location information.
- Step ④: The recorded key and location information are registered to the B⁺-tree through the following process. We locate the leaf node corresponding to the key and insert the recorded key and location information into an index array at the appropriate position.

The delete operation shares similar processing steps except for step 4. The two operations check whether the same key exists in the leaf node. If the answer is no, we add the recorded key and location information; otherwise, we modify the key's location information.

Get. Two steps are undertaken to get a KV pair. First, We initially pinpoint the BLog associated with the query KV pair, and then we check whether the targeted KV pair exists in the B⁺-tree index. If not found, a miss will be indicated. Otherwise, the value is retrieved from the log based on the location information. Second, if the value is empty, the query signifies a miss as the KV pair has been deleted. Otherwise, the value is the targeted KV pair.

Scan. Three steps are carried out to find all KV pairs in a key range. First, BLog provides an iterator that is applied to scan each KV pair in the log file. The iterator supports the Seek(), Prev(), Next(), and Value() functions. Second, Seek(key) locates the smallest key in the range of query keys in B⁺-tree index. Next() iterates the entire key range to retrieve corresponding location information. If the same location information is encountered multiple times, only one instance must be recorded. Third, the values of all query KV pairs are loaded from the log, where the KV pairs with non-empty values are the targeted query KV pairs.

Reducing memory usage in BLog. To curtail memory usage in BLog, we use prefix compression and shared location information to reduce indexing overhead. Each index entry contains key and location information. We implement a prefix compression method to reduce the size of a key, which identifies an index entry. Specifically, only the smallest key among these keys at the same leaf node is preserved entirely; in other keys, the portion of the suffix that does not overlap with the smallest key is preserved. In MTree, the key size is 16 B, and the prefix compression method can reduce the key size to around 14 B. The key size, denoted as K , is optimized to a window between

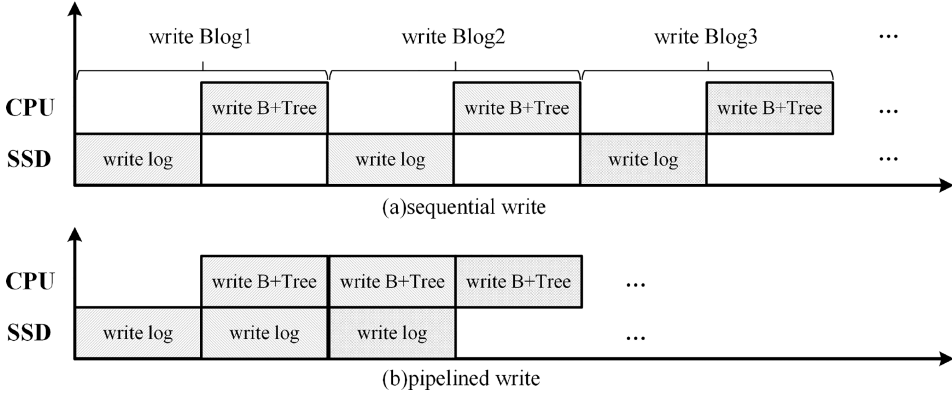


Fig. 7. Sequential and pipelined write.

1 B and 3 B in this study, where variables are defined in bytes. Location information registers the location of a block encompassing the KV pair. KV pairs in the same block share the location information. To record the location information of a 4 KB block, we need 4 B for the offset and 2 B for the size. V represents the size of a KV pair, and a 4 KB block contains $\frac{2^{12}}{V}$ KV pairs. Additionally determining which block corresponds to a key-value pair requires referencing 2-B records. After optimization, the size of the location information of each KV pair is translated to $(2B + 6B \cdot \frac{2^{12}}{V})$. In addition, the memory overhead of the internal nodes is evenly spread over each KV pair as E , which never exceeds 1 B. The footprint of indexes is substantially compressed through the aforementioned strategy. For example, given a KV pair with 16-B keys and 1024-B values, the indexing overhead per KV pair shrinks from 20 B to 6.27 B. When allocating a memory size of M for the BLog, the upper limit capacity of the BLog is approximated to be $BLog_{capacity}$. Then, we have

$$BLog_{capacity} = \frac{M \cdot V}{E + K + 2B + 6B \cdot \frac{2^{12}}{V}}, \quad (1)$$

where $BLog_{capacity}$ guides the allocation of BLog capacity in each bucket. We discuss in Section 7.1.1 the effect of different sizes of $BLog_{capacity}$ on write performance.

Pipelined Write Procedure. The time overhead of writing data to BLog is divided into two aspects. First, KV pairs are written into a log on the disk, which consumes I/O resources. Second, the keys and location information of KV pairs are stored to the B⁺-tree, which drains computational resources. The rapid development of storage devices has drastically shortened I/O time thanks to high throughput and low latency characteristics—and computation time is no longer negligible compared with I/O time. While writing data to the log to obtain its location information and then writing it to the B⁺-tree is necessary, this sequential process results in under-utilizing both computing and I/O resources over a period. A simple yet effective way of enhancing write performance is to parallelize the steps of a BLog write. For example, concurrently undertaking the write log step and the write B⁺-tree step shortens write time. Unfortunately, it is futile to parallelize a single BLog write due to data dependency. The good news is that no data dependencies exist among multiple batches of BLog writes—and we realize parallel processing by overlapping the write I/O time and computation time of multiple BLog writes as illustrated in Figure 7.

4.3 Buckets and BLog→LSM-tree

The $BLog \rightarrow LSM\text{-tree}$ procedure is slated to write data residing in the BLog to the LSM-tree. MTree employs the tiering compaction strategy to reduce data rewriting at the same level. To

cut back the data volume processed during a compaction operation, we divide an LSM-tree into multiple partitions based on key ranges. In addition, MTree accumulates a large amount of written data for the LSM-tree when a BLog is adopted. The BLog size may reach up to several gigabytes, which is expected to reduce the number of levels in the LSM-tree. However, writing a large volume of BLog data to the LSM-tree at once can cause a high write stall. To reduce the data volume involved in the BLog→LSM-tree procedure, we partition BLogs based on key ranges, aligning with the partitioning of the LSM-tree: LSM-tree and BLog in the same partition share the same key ranges. Recall that we refer to a partition in MTree as a bucket, encompassing a BLog and an LSM-tree. MTree offers three benefits: ① The memory overhead of logging location information is reduced due to the small-sized log. ② The log data of the same key range is centralized, leading to high scan performance. ③ The log **garbage collection (GC)** is easily accomplished following the BLog→LSM-tree procedure.

Tiering compaction. We limit the number of levels in an LSM-tree to a maximum of L and restrict the number of overlapping sorted runs at each level to a maximum of T . Then, there are at most $L \times T$ runs in the LSM-tree when fully populated. Each bucket has an upper bound on the LSM-tree capacity (e.g., $L = 2$ or $T = 8$). The upper limit on the number of sorted runs favors the lookup performance. When the LSM-tree is filled, the total write amplification caused by tiering is capped at two. The process of tiering is usually straightforward. Except for the last level, a merge operation is initiated when the number of overlapping sorted runs within a level reaches the threshold T – and this operation involves all sorted runs at that level. A larger-sized sorted run is generated after sorting, and the run is stored in the adjacent higher level.

BLog→LSM-tree. The BLog in each bucket has a maximum capacity. Once the amount of write data reaches the threshold, the BLog→LSM-tree operation is kicked off. The process is summarized as follows. ① Before writing data into the LSM-tree, a new BLog is generated to store the write from this bucket, preventing user writes from being hindered. ② An iterator is built for the BLog to iterate through each KV pair in the BLog to yield an SSTable. ③ According to the SSTable size, we identify a level with a matching data volume and deposit data at the level within the LSM-tree.

Buckets in MTree abate the data volume involved in the BLog→LSM-tree procedure and facilitate tiering of small-sized data volume during compaction. Leveraging buckets enhances compaction concurrency, thereby improving system scalability.

Discussion. We perform a set of experiments (see Sections 6.1.2 and 6.1.3) to evaluate if the KV separation in level 0 and the tiering structure improve the read and write performance. Since we set the data volume in our experiment to 100 GB, with the number of buckets in MTree set to a typical range of several hundred, we employ an array to manage these buckets. As the data volume in MTree increases, the number of buckets also grows, with each bucket containing a BLog. For a data volume of 100 TB, the number of buckets could potentially reach 1 million. It can be impractical to manage millions of buckets using an array in MTree. Therefore, we plan to construct a global B^+ -tree to manage these millions of buckets in our future work. This B^+ -tree will store the minimum key and the address of each bucket.

In addition, when there is only one BLog, two methods transfer the data in the BLog into buckets once the BLog is full. The first method is that all the data from the BLog are written into different buckets simultaneously. The GC can be performed by simply deleting the BLog's log file. However, this method involves a large amount of data transfer during the BLog→bucket operation, which can block write requests and cause long-tail write latency. While the second one is that MTree only writes the data corresponding to the key range of a bucket from the BLog into that bucket. This approach avoids long-tail write latency, with only a portion of data in the BLog log marked as invalid, increasing the GC cost. Additionally, if there is only one BLog, the values of adjacent keys in the BLog's log are stored in scattered locations, leading to high scan cost.

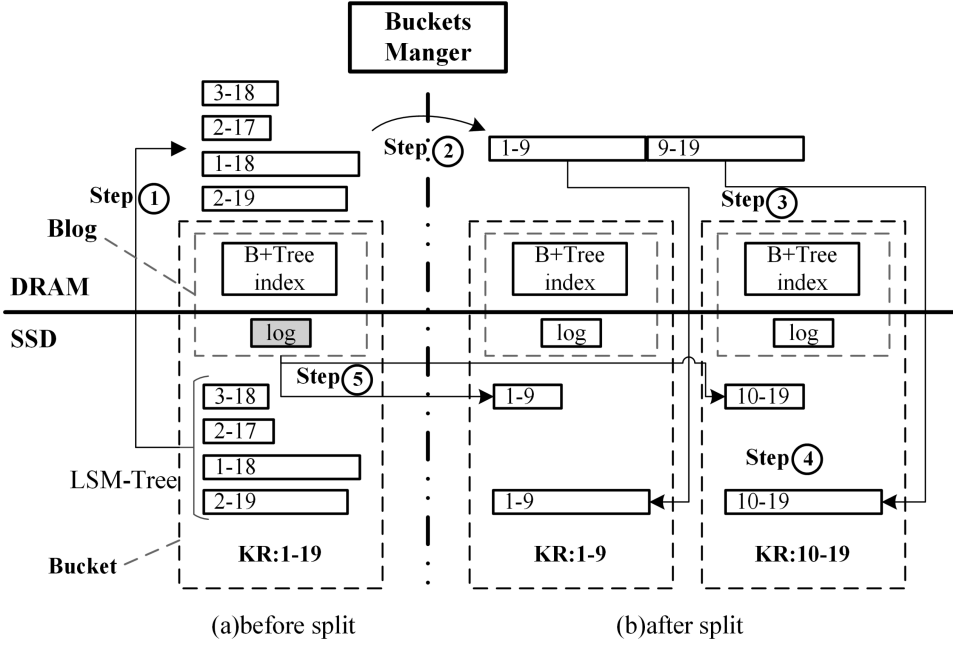


Fig. 8. The splitting procedure of a bucket.

4.4 Splitting Buckets

In line with the long-term key-density stability in workloads, the key range of each bucket is adjusted to enable each bucket to store a similar data volume. Upon initializing the system, MTree only allocates a small number of buckets due to the unknown key distribution in the workload at this stage. To boost read performance, the capacity of a bucket should not be set too large; therefore, these initial buckets will be fully filled. If the data volume in a bucket reaches a predetermined threshold, the bucket-splitting operation will be sparked. The key range of each bucket is dynamically adjusted to align with the key distribution within the key space through splitting operations.

For each bucket, we set the number of levels of the LSM-tree to a maximum of L , and the number of overlapping sorted runs in each level to a maximum of T . When the number of overlapping runs at the highest level reaches T , the data volume in the bucket crosses a threshold. A bucket, containing a BLog and an LSM-tree, is split into N small-sized buckets according to the data volume—and the BLog and LSM-tree in the bucket are partitioned into N smaller BLogs and LSM-trees. To mitigate the impact of prolonged splitting operations, we initiate a concurrent splitting process in the LSM-tree while the BLog receives incoming write data. There is no upper limit on the BLog size to prevent write requests from being impeded by the splitting operations.

Figure 8 sketches the splitting process.

- Step ①: All sorted runs in each level are retrieved.
- Step ②: A global iterator is constructed to sort all sorted runs.
- Step ③: After dividing sorted runs into equal-sized partitions, the runs are written back to the SSD.
- Step ④: N new buckets are forged one by one according to the N sorted runs. Each new bucket encompasses a new BLog and a new LSM-tree, with the sorted runs stored at the highest level of the LSM-tree.

- Step ⑤: The original BLog is split into N parts in accordance with the range of the newly generated N buckets, and the N part are stored into the LSM-tree as sorted runs.

During the splitting operation, MTree reorders data in the LSM-tree where it is inevitable to encounter high write amplification—that concern is negligible as the write amplification is bounded by $\frac{N+1}{N}$. A bucket containing a BLog and an LSM-tree is split into N new buckets according to the amount of data. Let us assume that the size of a full bucket is 1. Then, the size of each new bucket will be $\frac{1}{N}$ at the bucket splitting; thus, it will take $(1 - \frac{1}{N})$ new data to fill the new bucket. The ratio of the amount of data in the bucket 1 to the amount of newly written data $(1 - \frac{1}{N})$ is $\frac{N}{N-1}$, which is the write amplification caused by the bucket splitting. In simpler terms, the splitting operation introduces marginal I/O overhead, resulting in a minimal impact on the average write latency.

4.5 Dynamic BLog

Because short-term key density continuously fluctuates, it is unlikely to be an optimal solution to dividing BLogs equally among various buckets. The ever-changing write frequency of these buckets causes the fluctuation of key density in different intervals. Thus, we allocate BLogs according to the write frequency.

More BLog resources are designated to a bucket with a high write frequency and vice versa for buckets exhibiting low write frequency—and most data are stored in a large-sized BLog. In doing so, a majority of data can go through fewer LSM-tree buffer levels¹ because BLog→LSM-tree selects the level in agreement with the BLog size. Fewer LSM-tree buffer levels entail a low write amplification that fully utilizes resources. We face, however, a problem that the write speed of buckets constantly changes—and it is unreasonable to reserve a BLog size in the future based on past write speed. The goal of allocating BLog, aiming to meet the write speed in the future rather than the past speed, is to conserve BLog resources. In this study, we use past write speed to predict future ones through time series forecasting [6] guiding BLog-resource: the allocated BLog size is expected to match the changing write speed. BLog allocation—a frequent operator—requires a low overhead for predicting future write speed. As future write speed constantly changes, we intend to model the future write speed as a function of time. Because the data dimension of past write speeds is small, we obtain the write speed function through the least squares curve fitting.

Table 2 lists the notations and their description in this paper. The allocation metrics for BLog capacity in each bucket can be calculated using the write speed function for each bucket and average time (Δt). Δt represents the time spent in fully utilizing a BLog according to the write speed function. Given the i^{th} bucket, we apply the least squares to obtain the allocation metric through the three phases below.

First, MTree calculates the write speed of the i^{th} bucket at time t_j . We are unable to acquire the specific write speed at a particular moment, but we estimate the i^{th} bucket's write speed at the time t_j by computing the average write speed of $[t_j, t_j + \Delta t_{N_i}^j]$. Thus, we have $v_i^j = \frac{N_i^j}{\Delta t_{N_i}^j}$.

Second, MTree uses the least squares curve fitting (i.e., $v_i(t)$) to fit the write speed (t_j, v_i^j) ($1 \leq j \leq n$) at a sequence of moments, thereby predicting the write-speed change of BLog in the i^{th} bucket. $v_i(t)$ is composed of m linearly independent functions $v_{i_0}(t) = 1, v_{i_1}(t) = \sin t, v_{i_2}(t) = \sin 2t, \dots, v_{i_{m-1}}(t) = \sin(m-1)t$.

$$v(t) = a_0 + \sum_{h=1}^{m-1} a_{h-1} \sin(h \cdot t) \quad (m < n). \quad (2)$$

¹Herein, LSM-tree buffer levels refer to levels rather than the highest level.

Table 2. Notation and Description Used in This Article

Notation	Description
K	The number of buckets and there is $K \in \mathbb{N}$
$v_i(t)$	BLog's write speed function corresponds to i^{th} bucket ($1 \leq i \leq K$, and $i \in \mathbb{N}$)
v_i^j	The write speed of the BLog corresponding to the i^{th} bucket at time t_j ($1 \leq j \leq n$, and $j, n \in \mathbb{N}$)
N_i^j	The size of the buffer occupied by i^{th} bucket corresponding to BLog from time t_j
$\Delta t_{N_i^j}$	The time cost by i^{th} bucket corresponding to the BLog to write N_i^j -sized data from time t_j
B_i^*	The i^{th} bucket corresponds to the BLog's allocation index
B_i	The actual BLog allocated to i^{th} bucket
M	Total BLog size allocated in memory
$M_0 = M - B_1 - B_2 - \dots - B_K$	The size of BLog that is not used

We input (t_j, v_i^j) ($1 \leq j \leq n$) into $v_i(t)$ to derive a contradictory equation $Ka = v$, which is expressed as

$$\begin{bmatrix} 1 & \sin t_1 & \dots & \sin(m-1)t_1 \\ 1 & \sin t_2 & \dots & \sin(m-1)t_2 \\ \dots & & & \\ 1 & \sin t_n & \dots & \sin(m-1)t_n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \dots \\ a_{m-1} \end{bmatrix} = \begin{bmatrix} v_i^1 \\ v_i^2 \\ \dots \\ v_i^n \end{bmatrix}. \quad (3)$$

Third, an approximate solution to this contradictory system of equations can be found using the method of least squares:

$$Q = \sum_{j=1}^N \left[\sum_{h=0}^{m-1} \left(a_0 + \sin(h \cdot t_j) \cdot a_h - v_i^j \right) \right]^2, \quad (4)$$

where Q is a quadratic function of m independent variables a_h and continuous. There is a set of numbers $a_0, a_1, a_2, \dots, a_{m-1}$ minimizing Q . That is, we obtain the extreme value of the Q problem. Let $\frac{\partial Q}{\partial a_h} = 0$, and we have

$$K^\top K a = K^\top v. \quad (5)$$

Then, we bring forth the value of a that is the fitting function $v_i(t)$ for the write speed of the i^{th} bucket. Given k buckets in the MTree, the above model derives the k buckets' write speed function denoted as $v_1(t), v_2(t), \dots, v_k(t)$. Then, we extract the average time Δt spent in filling up the BLogs as

$$\int_t^{t+\Delta t} [v_1(t) + v_2(t) + \dots + v_k(t)] dt = B. \quad (6)$$

Next, given the average time Δt and the fitting function $v(t)$ for the write speed of the i th bucket, we assess the BLog allocation metric for the i th bucket as

$$B_i^* = \int_t^{t+\Delta t} v_i(t) dt. \quad (7)$$

After estimating the BLog allocation measures, MTree puts together the allocation metrics, the total BLog capacity, and the decision on performing a flush operation to determine the allocated BLog capacity of the bucket. Initially, each bucket's allocated BLog capacity has upper and lower

ALGORITHM 1: BLog allocation algorithm based on the i^{th} bucket

Input: The allocation metrics of the BLog (B_i^*) corresponding to the i^{th} bucket. The BLog size is not used (M_0). Whether it is in the process of splitting (*splitting*). Whether it is in the process of BLog→LSM-tree (*flushing*). Whether the allocated BLog has been written to full (*full*).

Output: BLog(B_i) allocated for the i^{th} bucket.

```

1: if full then
2:   if  $B_i^* \geq B_i$  and  $M_0 \geq 2MB$  then
3:      $B_i \leftarrow B_i + 2MB$ 
4:      $M_0 \leftarrow M_0 - 2MB$ 
5:   else
6:     if splitting and  $M_0 \geq 2MB$  then
7:        $B_i \leftarrow B_i + 2MB$ 
8:        $M_0 \leftarrow M_0 - 2MB$ 
9:     else
10:      if flushing and  $M_0 \geq 2MB$  then
11:         $B_i \leftarrow B_i + 2MB$ 
12:         $M_0 \leftarrow M_0 - 2MB$ 
13:      else
14:         $M_0 \leftarrow M_0 + B_i - 2MB$ 
15:         $B_i \leftarrow 2MB$ 
16:      end if
17:    end if
18:  end if
19: end if
20: return  $B_i$ .
```

limits, which guide the choice of an appropriate level when data in a BLog is flushed to the LSM-tree. When a bucket is originated, the allocated BLog capacity is set to the lower limit. When the allocated BLog capacity of that bucket is fully utilized, Algorithm 1 is designed to determine the allocated BLog capacity of the bucket, see Algorithm 1.

- Step 1: MTree judges whether the BLog capacity is smaller than the allocation metric and whether the unused BLog in the memory is larger than 2 MB—and if both conditions are met, MTree assigns another 2-MB BLog to that bucket. (Lines 1~5).
- Step 2: If the former condition is not satisfied, MTree judges whether a split operation or a flush operation is being performed at this moment—and if this condition is met, MTree assigns another 2 MB of BLog to that bucket. (Lines 6~12).
- Step 3: For the other cases, MTree flushes BLog data into the LSM-tree. (Lines 13~20).

4.6 Crash Recovery

It must be noted that a crash may occur when MTree is loaded with data. Both the BLog block buffer and B⁺-tree index structures reside in memory, so the recovery procedure is required after MTree crashes. MTree restores block buffer contents by replaying WAL, while B⁺-tree indexes are reconstructed through reprocessing the logs of BLog.

For BLog's block buffer, recall that we introduce in Section 4.2 a method to use a block to buffer KV pairs in BLog as disks are addressed on a page-by-page. After filling a block, we append it to the log to avoid high write amplification. MTree ensures crash consistency for data in the block by appending, providing WAL for the contents of a write batch to be appended along with a sequence number. The WAL size is limited and needs to be replaced periodically. If the maximum sequence

number of a WAL is smaller than the minimal sequence number of a block in each BLog, all KV pairs in that WAL can be deleted because the pairs have been written to disks.

When data is written to the BLog, it is appended to an in-disk log. The index entries for each KV pair in the log are maintained in DRAM through a B⁺-tree structure. To ensure crash recovery for B⁺-tree index, the indexes can be reconstructed by replaying the log of the BLog. Thus, the easiest crash recovery approach is traversing the log in the BLog once to reconstruct the B⁺-tree index. However, the amount of data in the BLog is relatively large, and traversing the log leads to a high overhead. Therefore, MTree registers a copy of the index information of all KV pairs in BLog as a data segment in the log. Consequently, only a small amount of log data needs to be accessed during crash recovery to reconstruct the B⁺-tree index: a data segment—being added between every two blocks in the log—includes the key, the locations of all KV pairs in the previous block, and the offset and size of the subsequent data segment. During the system crash recovery, MTree retrieves the initial data segment from log offset 0. Using the offset and size metadata in the current segment, all subsequent data segments are sequentially read. Leveraging the index metadata within these segments, the complete B⁺-tree index structure is reconstructed.

5 Experimental Setup

We conducted experiments on a Dell EMC PowerEdge T550 with an Intel Xeon Platinum 8380 processor and 32-GB DRAM. We employ Ubuntu 22.04 LTS with a Linux 5.15 kernel. In this study, we validate our proposed MTree on different SSDs; then, we deploy a SATA SSD (Samsung 870), a PCIe SSD (Optane SSD 900p), and a four-Ultrastar DC SN640 NVMe SSDs-based RAID. We compare MTree against state-of-the-art KV stores: HyperLevelDB, RocksDB_leveling, RocksDB_tiering, and PebblesDB.² We implement MTree by extending HyperLevelDB in this development. RocksDB_leveling is a baseline. RocksDB_tiering and PebblesDB are tiering KV stores. To maintain consistency in memory management, we establish an upper memory usage limit of 128 MB for indexes in BLog. HyperLevelDB, RocksDB, and PebblesDB use two 64-MB memtables in memory. We configure the parameters L and T to 2 and 8. Unless otherwise stated, MTree is pre-allocated with 50 buckets, each containing a BLog with an upper limited capacity of 32 MB.

In the experiments, we measure the performance of MTree by answering the following research questions:

- △ What is the write amplification of MTree? (Experiment 1)
- △ What is the performance of MTree under DB_Bench with random-write and random-read workloads? (Experiment 2)
- △ Does MTree solve the write long-tail latency problem? (Experiment 3 and 4)
- △ What is the performance of MTree under real-world YCSB-C workloads? (Experiment 5)
- △ What is the performance of MTree under different update workloads? (Experiment 6)
- △ What is the performance of MTree when handling variable-sized values? (Experiment 7)
- △ How does total BLog capacity size affect write performance? (Experiment 8)
- △ How does the pipelined write procedure in BLog affect write performance? (Experiment 9)
- △ Can the splitting buckets respond to long-term key-density stability? (Experiment 10)
- △ What is the performance gain from a dynamic BLog? (Experiment 10)
- △ What is the performance of MTree under real-world workloads? (Experiment 11)
- △ What is the performance of MTree when handling small-sized values? (Experiment 12)
- △ What is the performance and power consumption of SplinterDB and MTree under YCSB-C and DB_Bench? (Experiment 13)

²RocksDB_leveling and RocksDB_tiering represent RocksDB with the leveling and tiering compaction strategy, respectively.

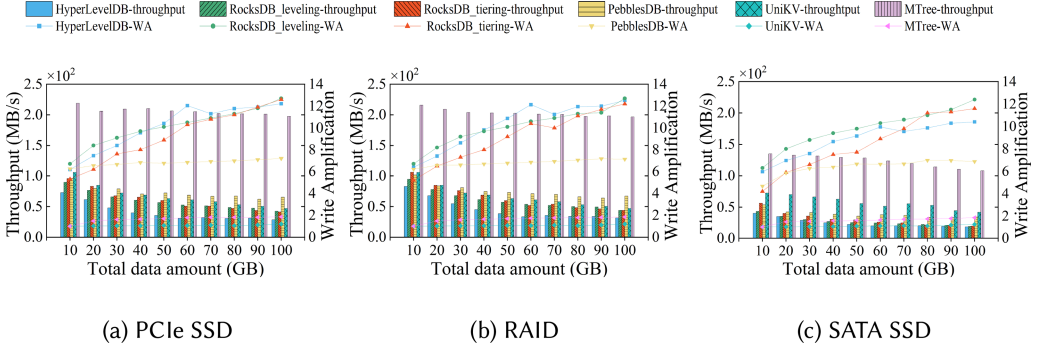


Fig. 9. Write amplification and throughput of the KV stores with various data volumes.

6 Performance Evaluation

6.1 Performance under DB_Bench

We replay DB_Bench—a micro-benchmark with key uniform distribution—to study the performance of KV stores.

6.1.1 Write Amplification—WA (Experiment 1). Figure 9 shows the WA measures of KV stores under the random-write workloads with different amounts of load data, 16-B keys, and 1-KB values.

MTree achieves the lowest write amplification and the highest write throughput across the entire spectrum of storage types, including PCIe SSD, RAID, and SATA SSD. PebblesDB's WA is higher than that of MTree by factors of 3.9, 3.8, and 3.7 on the PCIe SSD, RAID, and SATA SSD, respectively. The results unveil that the storage devices have little effect on WA; therefore, we delve into the performance of the KV stores running on PCIe SSD. When data volume reaches 100 GB, the WA of HyperLevelDB, RocksDB_leveling, RocksDB_tiering, and PebblesDB are 6.6 $\times$, 6.8 $\times$, 6.8 $\times$, and 3.9 $\times$ that of MTree, respectively. However, the WA of MTree is 1.5 $\times$ that of UniKV. MTree embraces BLog to reduce the number of levels in the LSM-tree closely to one; hence, a small amount of data is involved in compaction. BLog stores data in disk. To be fair, we consider both BLog and LSM-tree when measuring the WA of MTree. HyperLevelDB and RocksDB_leveling do not apply the tiering compaction strategy.

Consequently, the data involved in compaction includes all SSTables within the range of overlapping keys between two adjacent levels, resulting in significant WA. RocksDB_tiering and PebblesDB utilize tiering but have more LSM-tree levels. While tiering eliminates the need to rewrite data within the same level, it still relies on compaction when data is moved between levels. As a result, both show improved WA measures, though the improvements remain limited. UniKV applies KV partial separation mechanism, which stores KV pairs in multiple log files and records the pointers of keys in the LSM-tree levels, thereby reducing write amplification for large-sized KV pairs. Furthermore, MTree's WA fluctuates slightly from 1 to 1.86 when the data amount increases from 10 GB to 100 GB. MTree's write throughput remains consistent between 197 MB/s and 219 MB/s. On the contrary, except UniKV, the WA of the other KV stores shows a substantial increment as the amount of loaded data grows. For example, the WA of RocksDB_leveling climbs from 6.7 to 12.7; meanwhile, write throughput drops from 89.5 MB/s to 42.5 MB/s. Similar to MTree, UniKV's WA fluctuates slightly from 1.01 to 1.23 when the data amount increases from 10 GB to 100 GB; however, its write throughput drops from 109 MB/s to 47.9 MB/s.

6.1.2 Write Performance (Experiment 2). Figure 10 plots the throughput of KV stores under a random-write workload with 100 GB data and various value sizes across PCIe SSD, RAID, and SATA SSD. We analyze the performance of the KV stores on PCIe SSD.

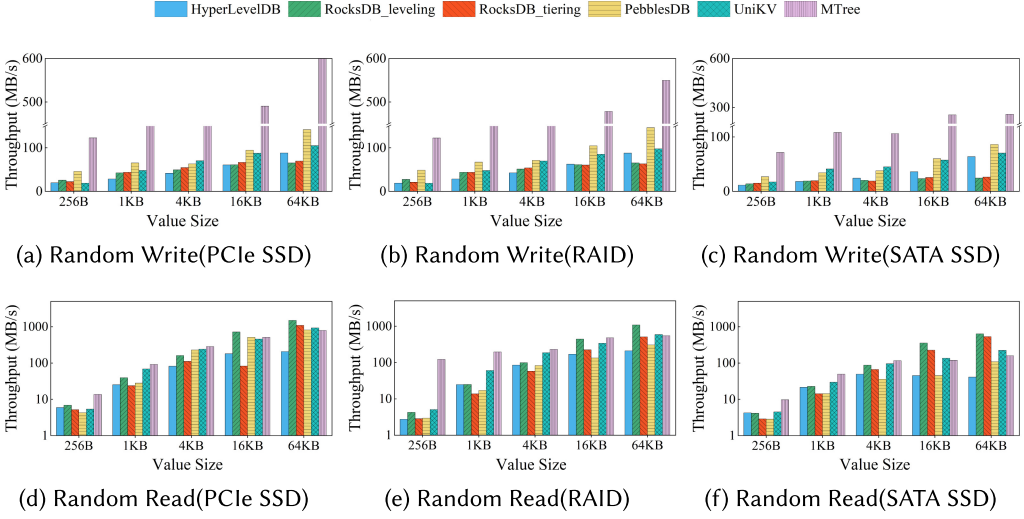


Fig. 10. Throughput of the KV stores with various value sizes.

On average, MTree’s random write throughput is $5.6\times$ to $8.1\times$, $4.7\times$ to $9.8\times$, $4.3\times$ to $9.2\times$, $2.7\times$ to $5.2\times$ and $3.3\times$ to $6.5\times$ higher than those of HyperLevelDB, RocksDB_leveling, RocksDB_tiering, PebblesDB and UniKV, respectively. HyperLevelDB and RocksDB_leveling frequently initiate compaction during data loading, interfering with flushing operations and degrading write performance. RocksDB_tiering and PebblesDB adopt the tiering compaction strategy to reduce the volume of data processed in a compaction procedure, thereby optimizing write performance. UniKV also employs a technique similar to tiering, introducing a dynamic range partitioning method. By mapping different ranges of KV pairs to independently managed partitions, each partition has its own UnsortedStore and SortedStore to manage data. MTree reduces the number of levels in the LSM-tree and employs tiering, which rarely triggers compaction. Thus, MTree delivers superior random write throughput compared to the other KV stores.

In Figure 10(c), MTree has a larger ratio of write throughput to RocksDB than those plotted in Figure 10(a) and (b). RocksDB undergoes more compaction than MTree, and the impact of compaction on write performance is more pronounced on the slow SSDs. Thus, MTree exhibits better write performance on SATA SSDs than the alternative KV stores. Compared with the tiering systems, PebblesDB, UniKV, and MTree share similar write performance on the tested SSDs.

6.1.3 Read Performance (Experiment 2). We gauge the random read performance of each KV store on PCIe SSD, RAID, and SATA SSD. Before replaying the random read trace, we execute random writes of 100 GB to create the database. Figure 10(d) shows the throughput of each KV store under the random-read workload with 100-GB data and different value sizes on PCIe SSD. MTree has a similar random read throughput. MTree has few LSM-tree levels, and the overlapping key ranges in each level become sorted after read-triggered compaction. The read-triggered compaction in MTree involves almost all data, which consumes CPU time and disk I/Os. With small-sized values, the total search time is long—and MTree has a long period to benefit from the small number of sorted runs after the compaction. MTree outperforms RocksDB_leveling in terms of random read performance. The total search time is short for the case of large-sized values, and MTree only enjoys a small period in which there is a benefit from a small number of sorted runs following the compaction. In addition, MTree is subject to interference with read amid compaction. Then, MTree’s random read performance is not on par with RocksDB_leveling.

Moreover, MTree has a clear read-performance advantage over PebblesDB by a factor of up to 1.94, 2.84, and 2.82 on PCIe SSD, RAID, and SATA SSD, respectively. Compared to UniKV, when the value size is below 4KB, the read performance advantage is up to 1.12, 1.23, and 1.68 on PCIe SSD, RAID, and SATA SSD, respectively. However, when the value size exceeds 4 KB, this advantage gradually diminishes and may even fall slightly below that of UniKV. This is because UniKV maintains a hash table in memory to index key-value pairs in the unsortedstore and employs key-value separation for large-sized key-value pairs. This allows more key address entries to be stored in the SSTable, enabling faster data location through index information during reads. MTree has an LSM-tree tier number close to 1, which shows an advantage on low-speed storage devices. Comparing with RocksDB_leveling and MTree on RAID and SATA SSD, we demonstrate a tradeoff in performance: given small-sized values, MTree is superior to RocksDB_leveling by a factor of 2.7; when it comes to large-sized values, MTree's performance lags RocksDB_leveling by up to 25%.

6.1.4 Write-tail Latency (Experiments 3 and 4). In this study, we use write latency as a metric to measure the duration between when a write request invokes the write operation interface and when the write operation status is returned. In this group of experiments, we simulate realistic production workloads and ensure that the blocking in a client does not affect other requests. In particular, we model the arrival process of user requests using a Poisson distribution. Request arrivals are considered random, with inter-arrival times following an exponential distribution. This entails that arrivals are independent and occur at a constant average rate. The Poisson distribution generates random inter-arrival times by setting the average arrival rate (λ), effectively simulating realistic request arrival patterns. Additionally, we study tail latency under various workloads to evaluate the maximum throughput of KV stores by limiting the request arrival rate to 20%, 40%, 60%, 80%, and 100% of their maximum throughput.

Figure 11 presents the average latency and the 50th, 90th, 99th, and 99.9th percentage write latency of KV stores under a 100-GB random-write workload, with respect to different ratios of maximum throughput. The results show that as the maximum throughput increases, the tail latency decreases across all KV stores, with MTree consistently achieving lower write latency. Under open-loop conditions, where maximum throughput ranges from 20% to 80%, the 99th percentage latency of MTree is 54%, 95.8%, 95.7%, 44.5%, and 23.8% lower than that of HyperLevelDB, RocksDB_leveling, RocksDB_tiering, PebblesDB, and UniKV, respectively. HyperLevelDB and RocksDB_Leveling experience frequent compaction operations, which block flush operations and impact foreground write rates. Consequently, even though their write latency decreases, their performance remains inferior to KV stores that utilize a tiering compaction strategy. MTree utilizes a tiering compaction strategy to reduce the data volume rewritten in a compaction procedure and further decreases the number of the LSM-tree levels. MTree reduces write latency and delivers superior performance over other KV stores.

When it comes to closed-loop conditions, the maximum throughput reaches 100%, and the 99th percentile latency of MTree is 76.5%, 97.9%, 97.9%, and 76.0% lower than that of HyperLevelDB, RocksDB_leveling, RocksDB_tiering, and PebblesDB, respectively. The latency of UniKV is nearly the same as that of MTree. RocksDB_tiering suffers from high write tail latency due to the lack of effective key partitioning; thus, it processes a large volume of data during compaction. PebblesDB curtails data volume involved in compaction using a guard structure, thereby significantly reducing write tail latency. UniKV reduces compaction granularity through dynamic key range partitioning, which helps avoid frequent compaction operations and achieve lower write tail latency, outperforming the average write tail latency. MTree further optimizes write tail latency by using a bucket structure to reduce the data volume involved in compaction. Additionally, the untimely transfer of data from BLog to the LSM-tree is addressed through the BLog allocation algorithm.

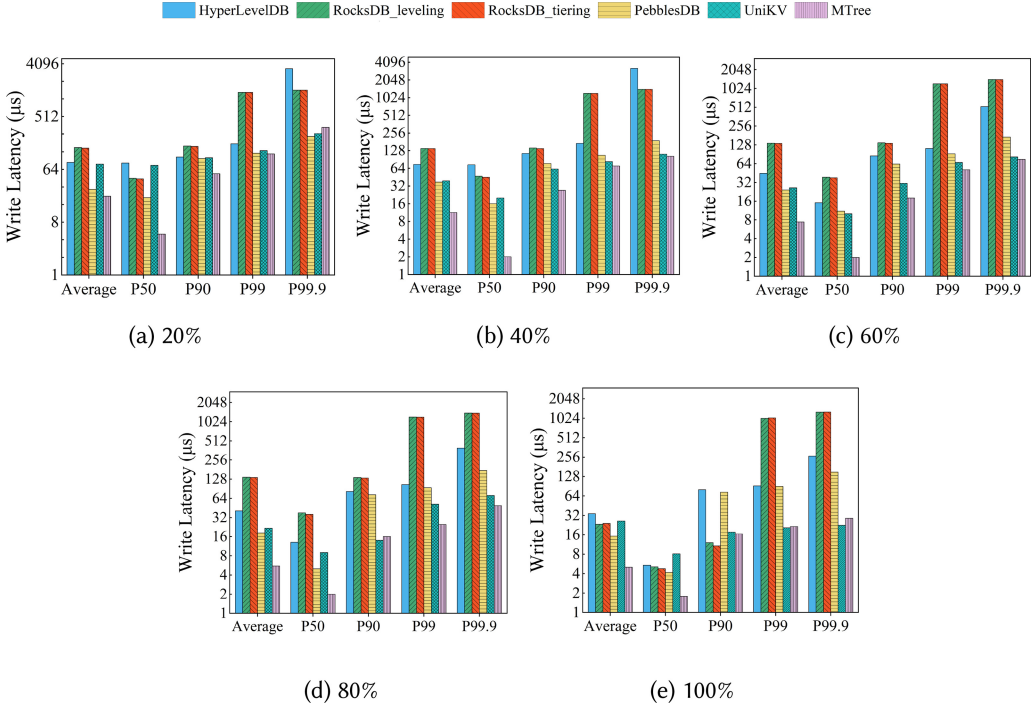


Fig. 11. Write tail latency distribution of MTree. The result shows the average, 50th, 90th, 99th, and 99.9th percentile write latency (in microseconds, μ s) for KV stores under various throughput levels (20% to 100% of maximum value).

Figure 12 records the throughput of each KV store per second during a 100 GB random write to visualize the benefits of reduced write-tail latency. MTree completes the 100-GB random writes more rapidly than other KV stores thanks to its superior random write throughput. We conclude that MTree delivers the most consistent performance among its peers.

Notably, when a write request, being about to invoke a write operation, encounters activities like compaction, it can block this request, preventing new requests from being generated until the block is cleared. In a production system, the time spent between generating a new request and invoking the write operation interface might include blocking, which means the tail latency is lower than that observed in production systems. Write latency measured in this article includes the time spent blocking the write latency, which allows us to study the write tail latency of KV stores. In future work, we need to collect real data on new request generation from production systems to study the write latency, aiming to obtain accurate write latency in production systems.

6.2 Performance under the Yahoo Cloud Serving Benchmark (Experiment 5)

Now, we study the performance of MTree under YCSB-C, features of which are listed in Table 3. Initially, we load 100-GB data into the KV store with uniform key distribution under YCSB-C. Zipfian key distribution is configured in all workloads except workload *D*. We also set the KV pair size to 16-B keys and 4096-B values. Figure 13(a) sketches the performance of KV stores on PCIe SSD under YCSB-C. In the insert phase, MTree's throughput is 6.5 times, 4.8 times, 4.0 times, 5.3 times, and 2.7 times higher than that of HyperLevelDB, RocksDB_leveling, RocksDB_tiering, PebblesDB, and UniKV, respectively. The performance enhancement is attributed to MTree's

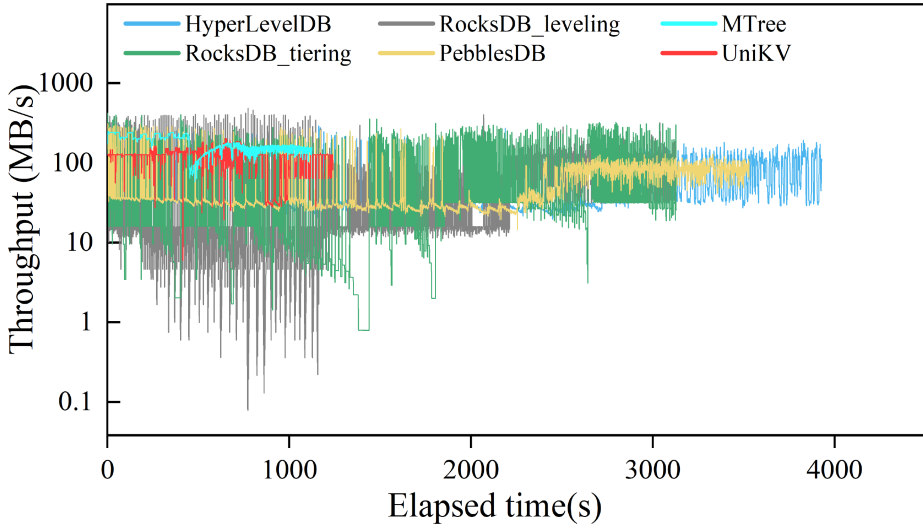


Fig. 12. Throughput fluctuations due to write stalls. The random write performance fluctuates where the troughs on curves signify the occurrences of possible write stalls.

Table 3. Features of Workloads in YCSB-C

Type	Feature	Distribution
Load	100% Writes	Uniform
A	50% Reads, 50% Updates	Zipfian
B	95% Reads, 5% Updates	Zipfian
C	100% Reads	Zipfian
D	95% Reads, 5% Inserts	Latest
E	5% Inserts, 95% Range Queries	Zipfian
F	50% Reads, 50% Read-Modify-Writes	Zipfian
User Define	Updates Workloads	Zipfian

HyperLevelDB RocksDB_leveling RocksDB_tiering PebblesDB UniKV MTree

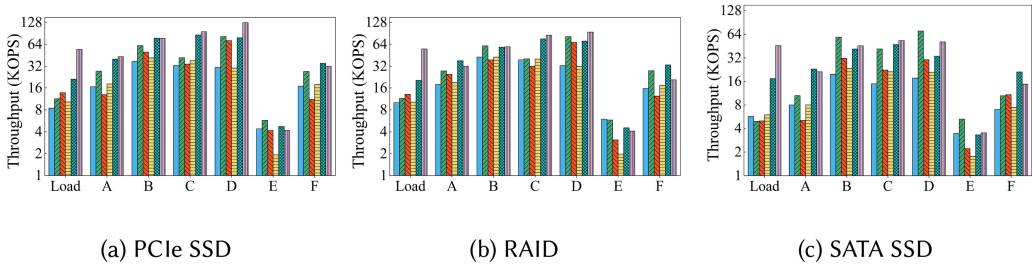


Fig. 13. Throughput of MTree under the YCSB-C workloads.

level being pushed close to 1, accompanied by low write amplification. For mixed read and write workloads, such as workloads A and F, MTree also outperforms the other KV stores. For read-intensive workloads, such as workloads B, C, and D, MTree achieves optimal performance

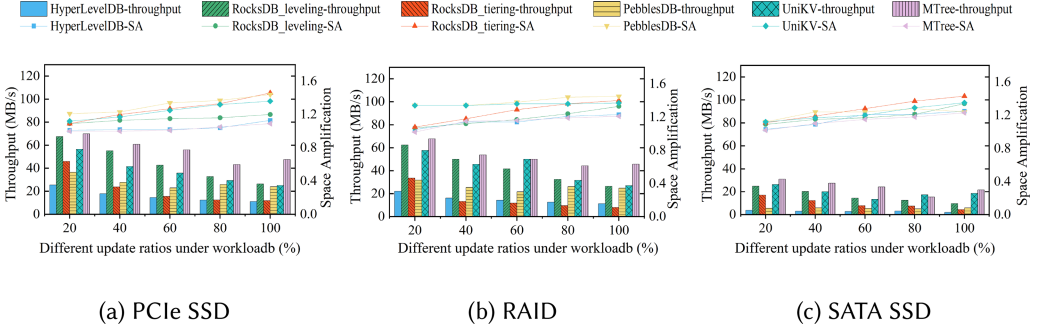


Fig. 14. Throughput and Space amplification of different update ratio under the YCSB-C workload B.

by maintaining minimal levels in the LSM-tree. UniKV employs specific strategies for optimizing scan performance within the UnsortedStore and SortedStore in each partition. As a result, for the scan-dominant workload E, UniKV holds a slight advantage over MTree.

Figure 13(b) and (c) shows the performance of the tested KV stores on RAID and SATA SSD. As the read/write performance of the tested devices declines, the performance of MTree, HyperLevelDB, PebblesDB, and UniKV worsens, but RocksDB's performance remains unaffected. Notably, RocksDB, a production system, provides well-rounded optimization. While MTree, HyperLevelDB, PebblesDB, and UniKV are experimental prototypes, RocksDB demonstrates robust read performance.

6.2.1 Performance under YCSB-C Workload with Different Update Ratios (Experiments 6). To evaluate MTree's write performance and space amplification under workloads with frequent updates, we modify the update ratios in YCSB workload B to 20%, 40%, 60%, 80%, and 100%. The key-value pair size is set to a default of 16 B for keys and 4 KB for values. We begin this group of experiments by loading 100-GB data into the key-value store. To examine space amplification, we gauge the ratio of the final amount of data written to disk to the original data loaded in disk. Figure 14 shows the performance of various key-value stores under five different read-update workloads. On PCIe SSD or RAID SSD, MTree's average throughput is 2.5x, 1.3x, 2.6x, 2.2x, and 1.5x that of HyperLevelDB, RocksDB_leveling, RocksDB_tiering, PebblesDB, and UniKV, respectively. Likewise, on SATA SSD, MTree improves the average throughput of these systems by 9.2x, 1.7x, 3.4x, 4.8x, and 1.8x, respectively. As the update ratio increases, the performance of all systems generally declines, though MTree experiences a more gradual decrease. This is because a single update operation essentially acts as an insertion, and as the update frequency increases, outdated data in disk also accumulates. Consequently, more frequent compaction operations are triggered, which reduces performance. Such effects become more obvious in HyperLevelDB and RocksDB_leveling. MTree alleviates the performance decline through (1) tiering to reduce the number of the LSM-tree levels and (2) a tiering-compaction strategy to reduce the data involved in compaction operations.

When it comes to space amplification, MTree generally maintains a superior level. HyperLevelDB and RocksDB_leveling do not utilize tiering. Thus, their compaction operations involve all overlapping SSTables in adjacent levels, which enables them to promptly discard outdated data to reduce space amplification, compared to RocksDB_tiering. PebblesDB leverages tiering to avoid data rewriting within the same level; however, the excessive number of LSM-tree levels and overlapping SSTables between levels that are not merged in a timely manner leads to higher space amplification. UniKV creates multiple partitions using dynamic range partitioning and

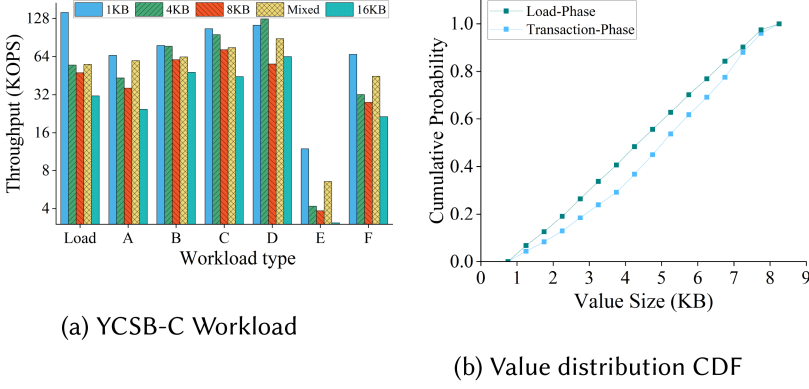


Fig. 15. The performance of different value sizes under YCSB-C and the distribution of value sizes under a specific workload. (a) shows the performance with value sizes of 1 KB, 4 KB, 8 KB, 16 KB, and mixed sizes. (b) illustrates the distribution of randomly generated value sizes during the load and transaction phases in workload B.

performs compaction and GC operations on each partition. Unfortunately, since these operations are performed in a sequential manner, outdated value records in vlogs or outdated index entries in SSTables may not be cleared promptly. Even worse, outdated data in the UnsortedStore may remain uneliminated, which results in significant space amplification. Unlike other schemes, MTree, with its LSM-tree levels close to one, experiences minimal impacts from tiering-compaction. Moreover, MTree employs a BLog structure to buffer newly written data and accumulate updates before transferring valid data to the LSM-tree. This novel design effectively avoids the drawbacks of tiering compaction, thereby lowering space amplification. The experimental results show that MTree has space amplification comparable to RocksDB_leveling, PebblesDB thanks to BLog's in-place updates.

6.2.2 Performance with Variable Value Sizes under YCSB-C (Experiments 7). In addition, we configure a mixed KV size workload to study the performance of MTree. The experimental results show that MTree also has stable performance under workloads with mixed KV sizes. We conduct two types of experiments based on YCSB-C to evaluate MTree under real-world workloads with varying value sizes. First, we set the key size to 16 B, with value sizes set to 1 KB, 4 KB, 8 KB, and 16 KB, totaling 100 GB of data. Second, due to the lack of tools for generating varying value sizes, we develop a load generator capable of producing values ranging from 1 KB to 8 KB, totaling approximately 100 GB of data comprised of 25,000,000 KV pairs. These setups enable us to measure the throughput of MTree under YCSB-C workloads. Moreover, to gain deeper insights into the density distribution of randomly generated value sizes, we analyze the distributions of varying value sizes under workload B. During the load and transaction phases, we collect the information on value size 1,200 times. To ensure randomness in the generation process, we sample the value size every 0.5 seconds, which ensures the reliability of the statistical results. Figure 15(a) depicts the performance of MTree under YCSB-C workloads with varying mixed KV sizes. It shows a noticeable decline in throughput as the value size increases during the load phase, indicating that larger values impose greater strain on the system. The throughput for mixed value sizes is approximately equivalent to that of the 4 KB workload case. This is due to the relatively uniform distribution of the randomly generated value sizes. Specifically, as indicated in the load phase (see Figure 15(b)), when the value size is approximately 4 KB, the cumulative frequency reaches 0.5, suggesting a uniform distribution of value sizes. Consequently, the generated data volume does not significantly deviate

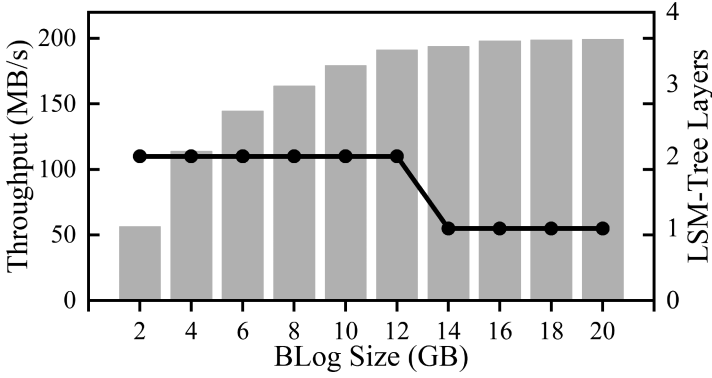


Fig. 16. Write throughput and the number of levels for MTree at different BLog sizes.

from the original fixed data volume, thereby resulting in consistent system performance under the same number of operations. In the transaction phase, an analysis of the cumulative distribution on the workload B (Figure 15(b)) reveals that when the value size is around 4 KB, the cumulative frequency is approximately 0.35. This indicates that the data distribution density predominantly falls within the range of 4 KB to 8 KB. Such a distribution leads to a generated data volume exceeding 100 GB, thereby reducing throughput compared to the 4 KB workload case. These comparison results in Figure 15(a) are expected. Moreover, mixed value sizes with MTree demonstrate better stability across most workloads, particularly in cases with significant load variation, where MTree provides superior performance. These results demonstrate MTree's strong adaptability in diverse real-world load scenarios.

7 Extended Study

7.1 Performance Gain Analysis

Let us investigate MTree's performance gain under write-intensive workloads. Because MTree achieves similar performance across the three tested storage devices, we study the performance gain on the PCIe SSD.

7.1.1 The Size of BLog (Experiment 8). To evaluate the impact of BLog size on performance, we adjust the total BLog size without limiting the memory size under a 100-GB random-write workload with 16-B keys and 1024-B values. Figure 16 unfolds the write throughput of MTree in case of different total BLog sizes. The write throughput of MTree rises as the total size of the BLog goes up to 20 GB from 2 GB. The write throughput is 56.5 MB/s for a total BLog size of 2 GB and 199.2 MB/s for a total BLog size of 20 GB. This trend is expected because large-sized BLogs decrease the number of levels in the LSM-tree, which curbs compaction data volume and achieves a high write throughput. When the total size of BLog is 14 GB, the write throughput of MTree slightly changes because the compaction ceases to occur when the LSM-tree levels reach a single one. The capacity of BLogs in each bucket is constrained by the fine-grained BLog→LSM-tree operation, which restricts the maximum capacity.

7.1.2 Pipelined Write Procedure (Experiment 9). We quantify the impact of the pipelined write procedure technique on the system performance under a random-write workload with a 10-GB dataset, 16-B keys, and 1-KB values. In this test, BLog accommodates 100% of the dataset to validate the pipelined write procedure. Figure 17 shows the random-write throughput of MTree governing the pipelined write procedure. The experimental results unravel that the random write throughput

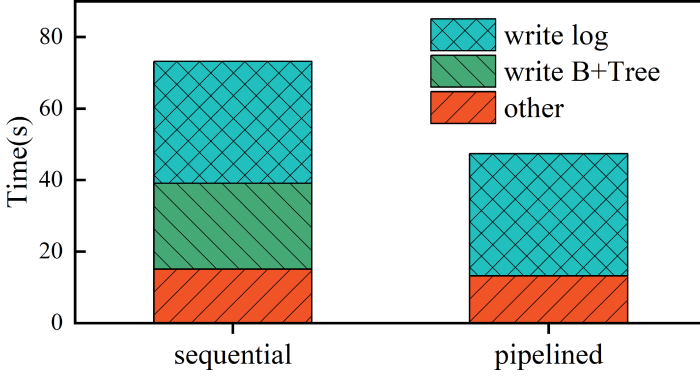


Fig. 17. The time overhead of MTree with a sequential and a pipelined write procedures.

of MTree without the pipelined write procedure is 139.9 MB/s. The write time for log and B⁺-tree is 34.11 s and 23.95 s because the low-latency storage device drastically slashes the I/O time—and the computation time is no longer negligible. The random write throughput of MTree with the pipelined write procedure is 216.1 MB/s, which improves the write throughput of BLog by a factor of 1.54.

7.1.3 Bucket Splitting. Because real-world workloads are mostly skewed, the data volume in some buckets in MTree grows faster with more sorted runs than others. However, the long-term key density of each bucket is relatively stable; then, we adapt to the changes of key distribution in MTree by virtue of bucket splitting to balance data volume in each bucket. We validate whether the splitting bucket technique can handle key distribution changes and measure throughput loss caused by the splitting process. We configure a 100-GB dataset with 16-B keys and 1-KB values under the workloads with a uniform key distribution and skewed keys. Figure 18 reveals the write throughput and data volume in each bucket in the context of initializing one bucket or pre-allocating 400 buckets in MTree. Under the uniform-key write workload, the write amplification of MTree is 1.86 in the case of 400 pre-allocated buckets and 2.19 in the case of one initialized bucket. Besides, all the buckets have a balanced amount of data with similar write throughput in both cases because splitting buckets does not block user write requests. Under the key-skewed write workload, the data in each bucket of MTree, using the bucket split technique, is more balanced, as this method can adapt to dynamic changes in key distribution.

7.1.4 Dynamic BLog (Experiment 10). To evaluate the dynamic BLog technique, we employ a 100-GB random-write workload with 16-B keys and 256-B values. The dynamic BLog technique reduces WA by utilizing the key distribution in MTree, reducing the number of levels in the LSM-tree. In addition, MTree alleviates the problem of the BLog→LSM-tree operation blocking the write requests. The WA of MTree is 2.54 and 2.69 with and without the dynamic BLog technique, respectively. Moreover, the 90th percentile write latency is 18.37 us and 21.75 us, respectively. We reckon that the dynamic BLog technique incurs a small overhead in calculating the distribution metrics for each bucket: the fitting function calculation for each bucket only entails a few matrix calculations—and its overhead is smaller than the time spent in loading data. Thus, we conclude that the overhead has negligible adverse impact on write throughput.

7.2 Performance of MTree under Real-world Workloads (Experiment 11)

Figure 19 shows the write amplification, throughput, and write latency of MTree under the Amazon books dataset. We can see from Figure 19 that except UniKV, the write amplification

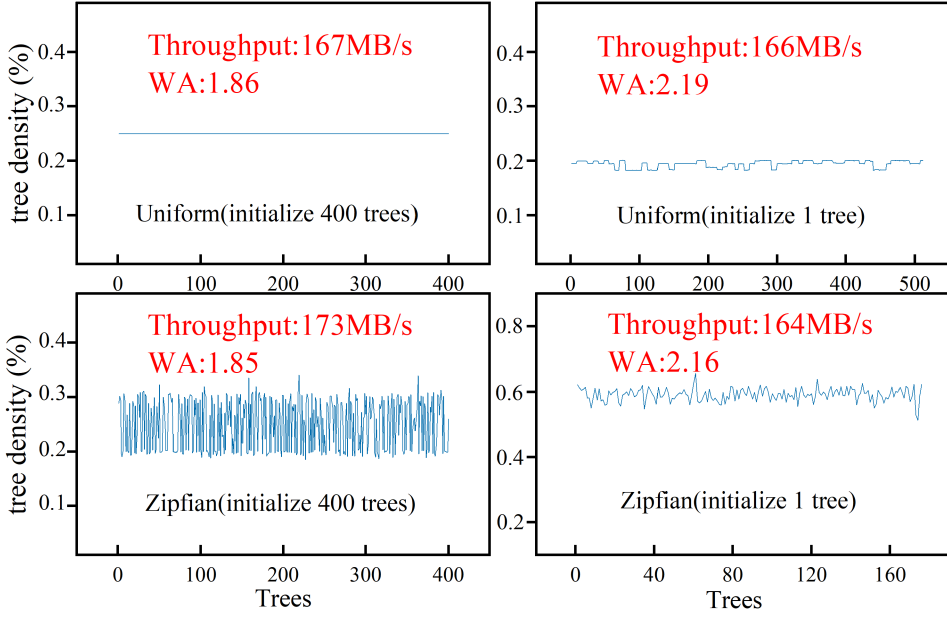


Fig. 18. The bucket' key density of MTree in case of the split of bucket technique under write workloads with uniform and zipfian key distributions.

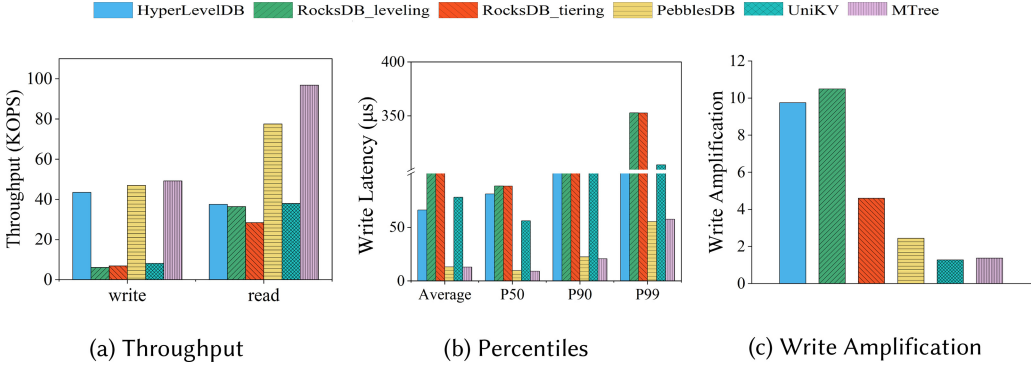


Fig. 19. Throughput, percentiles and write amplification of MTree under real-world datasets.

(WA) for HyperLevelDB, RocksDB_leveling, RocksDB_tiering, and PebblesDB is 7.1 $\times$, 7.7 $\times$, 3.3 $\times$, and 1.8 $\times$ that of MTree, respectively. MTree's write throughput is higher than HyperLevelDB, RocksDB_leveling, RocksDB_tiering, PebblesDB, and UniKV by 15%, 708%, 708%, 5%, and 509%, respectively. MTree's read throughput is higher than HyperLevelDB, RocksDB_leveling, RocksDB_tiering, PebblesDB, and UniKV by 157%, 165%, 239%, 25%, and 155%, respectively. MTree's 90th percentile latency is lower than HyperLevelDB, RocksDB_leveling, RocksDB_tiering, PebblesDB, and UniKV by 81.2%, 87.4%, 87.4%, 8%, and 83.5%, respectively. These results are expected. HyperLevelDB and RocksDB_leveling use the leveling compaction strategy, which frequently triggers compaction operations during data loading. As a result, they block write requests, leading to higher average write latency. Frequent compaction operations continuously rewrite data, leading to high write amplification and low write performance. One benefit of the

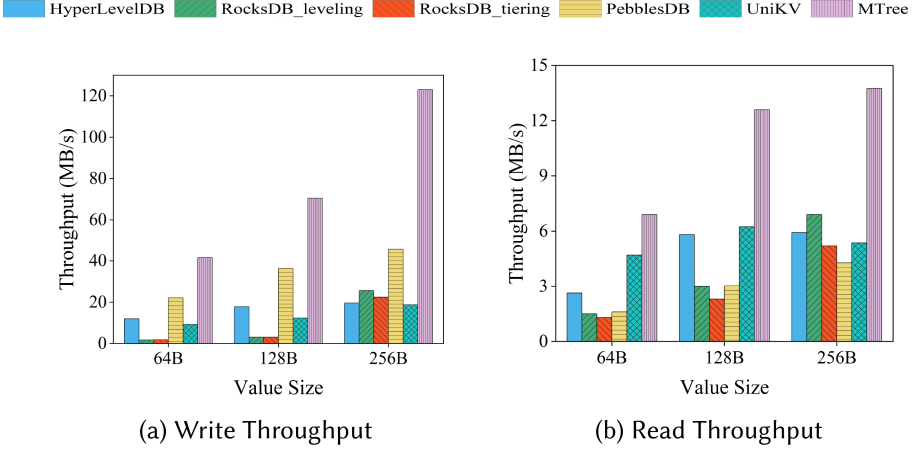


Fig. 20. Random write/read throughput of the KV stores with various small-sized values.

leveling compaction strategy is that the fully ordered structure of each level in the LSM-tree provides good read performance. RocksDB_tiering and PebblesDB use the tiering compaction to reduce compaction frequency, lowering write amplification and improving write performance. RocksDB_tiering involves a larger amount of data during compaction, which blocks write requests and causes long-tail write latency. In contrast, PebblesDB uses guards to control the data volume involved in compaction to avoid long-tail write latency. UniKV employs a strategy that combines a tiering strategy with key-value separation, aiming to reduce write amplification. Unluckily, under real-world datasets, compaction and GC affect system performance and thus yield lower write performance. On the other hand, UniKV maintains a hash-index table in memory to index a portion of on-disk data and reduces I/Os during queries through dynamic partitioning, thereby achieving high read performance. MTree leverages the tiering to reduce the number of levels in the LSM-tree, leading to infrequent compaction operations. By using buckets to manage the data volume for each compaction, MTree achieves low write amplification and high write performance, effectively avoiding long-tail write latency.

7.3 Performance of MTree under Workloads with Various Small-sized Values (Experiment 12)

As mentioned in prior work [8], approximately 80% of the data in large-scale production environments, such as ZippyDB and UP2X—two popular use cases of RocksDB at Facebook—consists of small values (from 32 B to 256 B). To study the throughput of MTree under workloads with small-sized values, we configure three small-sized values: 32 B, 64 B, and 128 B in random read and write workloads in our experiments. Figure 20 shows the throughput of MTree on PCIe SSDs.

Under random-write workloads, MTree achieves higher write throughput than HyperLevelDB, RocksDB_leveling, RocksDB_tiering, PebblesDB, and UniKV by 3.3×, 24×, 25×, 1.8×, and 4.9×, respectively. Both PebblesDB and RocksDB_tiering employ tiered compaction strategies to prevent data rewrites at the same level. PebblesDB enhances performance by utilizing its guard structure to decrease the written data volume during compaction, thereby outperforming RocksDB_tiering. In contrast, MTree significantly improves write performance because the BLog structure reduces the number of levels in the LSM-tree, and the tiering compaction strategy reduces rewritten data volume in compaction. Thus, MTree outperforms other KV stores.

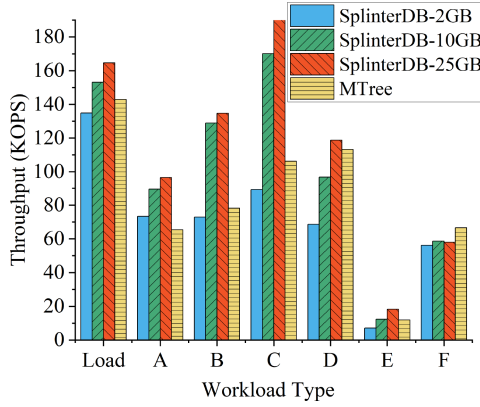


Fig. 21. Comparison of throughput of MTree and SplinterDB with different cache capacity configurations under YCSB-C.

Under random-read workloads, MTree achieves 185%, 340%, 450%, 430%, and 97% higher throughput compared to HyperLevelDB, RocksDB_leveling, RocksDB_tiering, PebblesDB, and UniKV, respectively. In addition to MTree, UniKV also exhibits high read performance. UniKV achieves this by not performing key-value separation for small values, reducing I/Os overhead. UniKV also employs an in-memory hash table for fast data retrieval. Its range partitioning strategy directly locates the target partition, thereby reducing extra data reads. In comparison to other KV stores, MTree has fewer LSM-tree levels. Besides, after read-triggered compaction, the overlapping key ranges within each layer are sorted. When values are small, MTree achieves high read performance because the small number of ordered segments are generated after compaction. It is noteworthy that such an optimization becomes more pronounced as the value size decreases.

7.4 Performance and Power Consumption of SplinterDB and MTree under YCSB-C and DB_Bench (Experiment 13)

7.4.1 Performance of MTree and SplinterDB. We evaluate the performance of MTree and SplinterDB with different cache capacities (2 GB, 10 GB, and 25 GB) under the YCSB-C. The key size is set to 16 B, with a 1-KB value. MTree consumes less than 2-GB memory. In the load phase, we write 100 million key-value pairs. SplinterDB's memory usage is categorized into two types: (1) a pre-configured fixed-size clock cache, which stores trunk nodes and branches and occupies a large portion of memory; and (2) a dynamically allocated cache for background threads, which occupies a smaller portion. In practice, SplinterDB can adjust memory usage by pre-configuring the clock cache capacity, enabling flexible deployment on servers with various memory resources. However, under small cache capacity configurations, SplinterDB's performance is adversely affected by the cache hit rates of the quotient maplets and branches, resulting in degraded performance.

As shown in Figure 21, in the load phase, the throughput of MTree is similar to that of SplinterDB-2 GB. However, SplinterDB-2GB exhibits a write amplification of 3.87, while MTree's write amplification is 2.33, delivering a reduction of 39%. In phases B, C, D, E, and F, MTree's throughput increases by 8%, 19%, 66%, 70%, and 17%, respectively, compared to SplinterDB. In phase A, MTree's throughput is slightly lower than that of SplinterDB-2GB, with a 10% decrement.

When memory resources become abundant, SplinterDB shows significant performance improvements across various YCSB-C workloads. We analyze the performance of SplinterDB with a cache capacity of 25 GB during load and C phases. In the load phase, SplinterDB-25 GB outperforms SplinterDB-2 GB in terms of flush and compaction times. Specifically, the flush time is decreased

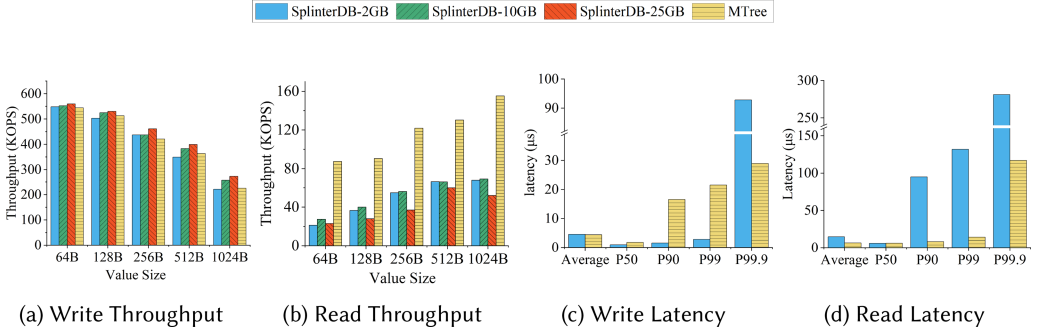


Fig. 22. Throughput and latency of MTree and SplinterDB under DB_Bench with random-write/read workloads.

by as much as 43%, and the compaction time is reduced by up to 11%. In the C phase, when using a Zipfian distribution for lookups, SplinterDB-2 GB yields a branch cache hit rate of 94%, while SplinterDB-25 GB reaches 99%. This results in a 73% reduction in I/O overhead from cache misses, leading to a 129% increment in lookup performance. In other YCSB-C phases, SplinterDB-25 GB also benefits from the larger memory, outpacing environments with limited memory. Thus, in memory-abundant environments, SplinterDB is a superior choice to MTree.

When it comes to a 100 GB random write/read workload in DB_Bench, we set the key size to 16 B, with the value size ranging from 64 B to 1 KB. Figure 22 presents the performance comparison between SplinterDB and MTree. It shows that, when SplinterDB uses a 2-GB cache, the write performance of MTree and SplinterDB is nearly the same across different value-size cases. However, in terms of random read performance, MTree clearly outperforms SplinterDB. Specifically, MTree's random-read performance is up to 310% better than SplinterDB's, across various value-size cases. This performance gap is mainly attributed to MTree's bucket index, which quickly locates the bucket containing the key. Within a bucket, the B⁺-tree facilitates direct lookup of KV pairs in disk, enhancing query efficiency. MTree's internal tree depth is lower than that of SplinterDB, and its smaller key range contributes to faster lookups. For example, under workloads with 1-KB value and 100-GB random writes, the average LSM-tree depth of MTree's bucket is 2, while SplinterDB's tree depth is 4.

Under random read workloads, SplinterDB with 10 GB and 25 GB cache capacities does not exhibit the same performance improvement as observed in the load phase. In some cases, higher cache capacities even lead to a decline in read performance. This is because the larger cache sizes, while improving cache hit rates, also increase replacement overhead and memory management costs. For example, with a 1-KB value in random reads, SplinterDB-2 GB has a branch cache hit rate of 74%, while SplinterDB-25 GB achieves 89%. However, the I/O overhead from cache misses goes up by 13%, leading to a 22% reduction in performance.

Figure 22(c) and (d) shows the tail latency for random writes and reads with a value size of 1 KB. It shows that MTree's average write latency is comparable to SplinterDB, but its P99.9 tail latency is 68% lower than that of SplinterDB. For 50th, 90th, and 99th latencies, SplinterDB exhibits lower latencies compared to MTree, with latency reductions of up to 86%. In random-read scenarios, MTree reduces the average latency of SplinterDB by 56%. Moreover, MTree outperforms SplinterDB in terms of all tail latency percentiles (except 50th), with the maximum latency reduction reaching 91%. In other scenarios, the tail latency behavior follows the same pattern, with SplinterDB performing better in terms of write tail latency, and MTree exhibiting superior read tail latency.

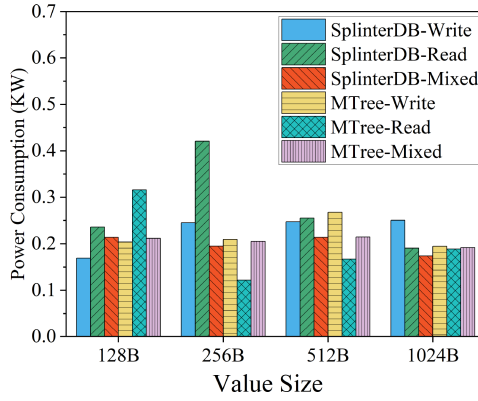


Fig. 23. Comparison of power consumption between SplinterDB and MTree.

In a server, the physical memory capacity is typically large, ranging from tens to hundreds of gigabytes. However, this does not imply that an application can exclusively occupy a significant portion of memory resources. In practice, there are existing multi-tenant and multi-application environments; therefore, it is necessary to design efficient memory resource allocation methods to dynamically allocate space on-demand for applications. Consequently, each application is constrained to a limited portion of memory resources. For servers with limited memory, under large-data-volume workloads, a small-sized clock cache can reduce the cache hit rates of the quotient maplet and branch cache in SplinterDB, leading to degraded system performance. Then, MTree outperforms SplinterDB in terms of performance. Conversely, under sufficient memory and workloads with data skew (hot spots), SplinterDB's size-tiered B^e structure delivers high throughput and low write amplification, surpassing MTree. However, considering available memory space, workload distribution, and multi-tenant contention, MTree remains a feasible solution for some applications.

7.4.2 Power Consumption of MTree and SplinterDB. To study the power consumption of MTree and SplinterDB with a 2 GB cache capacity under DB_Bench with 100 GB read, write, and mixed-read-write workloads, we employ the following methodology. First, we monitor the system's total energy consumption (in joules) every second using system commands (`cat /sys/class/powercap/intel-rapl:*/energy_uj`). At the end of each test, we calculate the difference in total energy consumption between the start and end points. The ratio of this difference to the total runtime represents the average power consumption. Figure 23 depicts the total power consumption of KV stores under 100 GB random-write/read workloads with various value sizes. The results indicate that MTree and SplinterDB exhibit similar power consumption under random-write workloads. Under random-read workloads, MTree demonstrates a 71% and 36% reduction in power consumption compared to SplinterDB for value sizes of 256 B and 512 B, respectively. However, for a value of 128 B, MTree exhibits 30% higher power consumption than SplinterDB, while at 1 KB, both KV stores demonstrate similar power consumption. The memory usage in MTree is primarily attributed to the index of BLog and Bloom filters. Under random-write workloads, both MTree and SplinterDB employ a tiered-compaction strategy, where data is first read into memory, sorted, and then written back to disk. This process increases memory usage, leading to similar power consumption in both KV stores. Under read workloads, MTree searches target keys faster than SplinterDB due to its fewer levels in the LSM-tree within a bucket and its in-memory BLog index. This design enables MTree to reduce the frequency of memory and disk accesses, thereby lowering overall power consumption.

Table 4. Tiering Compaction Strategies

	Reducing numbers of levels	Enabled scan	Reducing write tail latency	monetary cost	Using long-term key-density stability	Using short-term key-density volatility
RocksDB-tiering	✗	✓	✗	✓✓✓	✓✓✓	✗
PebblesDB [44]	✗	✓	✓✓	✓✓	✗	✓✓
LSM-Trie [49]	✗	✗	✓✓	✓✓✓	✓✓✓	✗
WipDB [57]	✓	✓	✓	✗	✓✓✓	✗
ChameleonDB [56]	✓✓✓	✗	✓✓✓	✗	✓✓✓	✗
LWC-store [52]	✗	✓	✓	✓✓✓	✓✓✓	✗
Dostoevsky [16]	✗	✓	✗	✓✓✓	✓✓✓	✗
LSM-Bush [38]	✗	✓	✗	✓✓✓	✓✓✓	✗
Spooky [17]	✗	✓	✓	✓✓✓	✓✓✓	✗
SifrDB [37]	✗	✓	✗	✓✓✓	✗	✓
ALDC [9]	✗	✓	✓✓	✓✓✓	✓	✗
dCompaction [42]	✗	✓	✗	✓✓✓	✓✓✓	✗
LightKV [23]	✓✓	✓	✓✓	✗	✗	✗
MTree	✓✓✓	✓	✓✓✓	✓✓✓	✓✓✓	✓✓✓

8 Related Works

LSM-tree attracts attention from academia and industry thanks to its high throughput and low latency. Compared with the traditional B^+ -tree, LSM-tree offers better write performance using a write buffer and a background compaction. However, the data rewrites caused by compaction trigger high write amplification. The write amplification of an SSD degrades performance and incurs wear-out problems; thus, it is essential to mitigate the write amplification issue of the LSM-tree. In this section, we focus on the related work to reduce the write amplification in KV stores.

Tiering. Table 4 reveals the factors considered by the state-of-the-art tiering compaction strategies. ✓ indicates that the work incorporates this factor, and its number represents the degree of optimization on the metric. ✗ indicates that the study does not address this concern. PebblesDB [44] provides a partitioned management of SSTables using well-sorted guards, in which SSTables can be unsorted. Since guards are generated based on a randomized algorithm, there is no guarantee that the amount of data in each guard remains consistent in the long term. In PebblesDB, queries are also triggered in the Memtable. If the target item is not found in the query, it searches each level of the LSM-tree. In a level, PebblesDB locates the guard associated with the item’s key and searches through each SSTable in the guard. PebblesDB employs a Bloom filter to check whether an SSTable contains the target item, avoiding unnecessary search cost. If the target item is not found at the level, PebblesDB searches for the next level until the target item is located or the last level in the LSM-tree. So, many levels in the PebblesDB result in low read performance.

SplinterDB [14] introduces a hybrid index strategy called STB^ϵ -tree that combines the advantages of the LSM-tree and the B^ϵ -tree. The trunk node in B^ϵ -tree allows the branches to exist overlapping key ranges, reducing the frequency of rewrites in a level and reducing write amplification. In SplinterDB, queries are first searched in the Memtable. If the target item is not found, the search continues down the B^ϵ -tree. When searching in a trunk node t , SplinterDB uses the quotient filter to check if the corresponding branch contains the target key. A quotient filter hit prompts a search within the corresponding branch. If the quotient filter misses, SplinterDB searches the correct child node until the target key is found or until a node without valid child nodes is reached. SplinterDB uses a tiering strategy in the trunk node, requiring multiple quotient filter reads for lookups within the node. The authors introduced the quotient maplet structure in Mapped SplinterDB [13] to accelerate query performance and reduce quotient filter size. This structure replaces multiple quotient filters in the same trunk node with a single quotient maplet lookup, re-

ducing the overall filter size and improving performance, even on machines with limited memory resources.

The query in MTree, PebblesDB, and SplinterDB involves searching overlapping key ranges at each level and using filters to check if the query key range contains the target item, thereby reducing search costs. Mapped SplinterDB reduces the number of filter lookups with maplets, which reduces query costs and the overall size of the filters. However, compared to PebblesDB and SplinterDB, MTree has fewer levels in the LSM-tree, achieving the lowest query cost.

In addition, many levels in the LSM-tree result in low read performance. LSM-Trie [49] incorporates partitioning to facilitate tiering by virtue of a prefix bucket structure. The hash-based partitioning strategy ensures that each partition shares the same data size, but the system does not support range queries. WipDB [57] utilizes bucket splitting and merging to furnish key division for tiering. Each bucket is set up with a MemTable, which decreases the number of levels in the LSM-tree and lowers read/write amplification. However, WipDB causes a high monetary cost. ChameleonDB [56] utilizes a hash-based key partitioning scheme, where values are stored in the log—and the index information of values and keys are stored in the PM. The compaction strategy employs a lazy-leveling approach, where levels beyond the last one are sorted using a tiering compaction strategy.

Likewise, LWC-store [52] employs a key partitioning scheme for tiering and a workload-specific balancing strategy—the technique that alleviates high write amplification and improves the write performance. Dostoevsky [16] uses a lazy-leveling scheme coupled with tiering except for the last level, reducing the write/read amplification and space amplification. The lazy-leveling technique, however, cannot furnish key partitioning—and a high write tail latency becomes inevitable. Spooky [17], dividing the largest level of data into the same-sized files, embraces a key partitioning scheme that hinges on the key ranges of those files to support small-grained merge operations. SifrDB [37] employs a tiering compaction strategy and merges overlapping sub-buckets, reducing space reservation for compaction. Jungle [1] combines an LSM-tree and a copy-on-write B⁺-tree to reduce update costs without compromising lookup performance. By adjusting the compaction threshold of its copy-on-write B⁺-tree, Jungle achieves a better balance between write amplification and space amplification. ALDC [9] uses the lower-level triggering compaction scheme, and the key range of SSTable facilitates a key partitioning scheme. The key compaction threshold is dynamically adjusted by adding an adaptive policy to adapt to changes in the ratio of real-time read/write operations in workloads. Unfortunately, many levels in the LSM-tree lead to low read performance. dCompaction [42] further improves the write throughput by delaying compaction operations to accumulate data in compaction, but a large-sized data volume in a single compaction procedure leads to a high write tail latency. LightKV [23] utilizes PM devices to perform key range partitioning based on the key distribution in the key space in the prior period. Nonetheless, the partitioning pays no attention to long-term key-density stability, causing imbalanced data sizes among partitions.

The aforementioned work using tiering compaction strategies more or less suffers from having one or more of the following shortcomings: (1) an excessive number of levels that leads to higher read/write amplification (2) no support for scans (3) a large write tail latency (4) higher monetary cost, and (5) failure to account for key distribution in the key space. MTree is adapted to tiering compaction strategy.

Key-value separation. KV stores that separate keys from values adjust the indexing granularity to key-value pairs. This approach allows for direct data lookup in memory using keys, thereby eliminating the need for compaction to reorder data on the disk. Consequently, write amplification is significantly reduced. Key-value separation stores also encounter several inherent challenges: 1. The values corresponding to adjacent keys are stored in scattered locations on the disk, leading

to high scanning cost. 2. To prevent excessive space amplification, GC must be periodically performed to remove invalid values. Keys in the key-value separation store can be stored in memory or on the disk. For instance, Wisckey [35] opts to store keys in disk, which introduces additional disk I/Os during read operations, thereby increasing read cost.

Wisckey [35] separates keys and values—and the keys are managed using the LSM-tree structure, whereas the values are managed using the log structure. In this way, compaction operations only require keys to be involved, reducing write amplification. Nevertheless, values in the log need GC to avoid high amplification, and Wisckey increases the overhead of range queries and the write performance that benefits from large-sized values. HashKV [10] deploys hash-based data grouping anchored on key-value separation, alleviating a high GC overhead. In addition, HashKV employs selective key-value separation and thermal awareness to boost GC efficiency, both of which can reduce I/O amplification. However, the hash-based data grouping method impairs range query performance. Parallax [50] employs different placement strategies for key-value pairs of varying sizes and establishes a B^+ -tree index for each level to enhance query performance. Specifically, small key-value pairs are placed in the leaf nodes of its B^+ -tree, while large ones are always stored in a value log. Medium-sized key-value pairs are stored in a value log, except at the last level, where they are placed in leaf nodes to delay GC until they reach the final level. This approach trades space amplification for lower GC costs and reduced write amplification. FenceKV [47] improves range query performance by storing values in a range based on key-value separation using a partitioning idea of a fence for the values. UniKV [55] introduces a partial-key-value separation-based dynamic range partitioning method. This approach reduces the frequency of compaction operations, lowering write amplification while delivering scan performance comparable to LevelDB. MTree curbs high write amplification by a two-folded approach, namely, (1) reducing the number of levels in the LSM-tree and (2) avoiding the effect of size ratios between adjacent levels without creating any indirection.

Hardware Support. NoveLSM [27] mitigates the load during burst writes by deploying a larger MemTable on PM, which lowers the number of levels in the LSM-tree and write amplification. MatrixKV [53] alleviates write stalls using PM to manage a larger level 0—and fine-grained compaction is employed in levels 0 and 1. In addition, the large data volume in level 0 reduces the LSM-tree’s depth, lowering write amplification. MTree aims to curtail write amplification without increasing monetary cost, thereby achieving notable performance gains.

Index Structure. SLM-DB [26] uses an index structure of B^+ -tree while managing large data sets, and SLM-DB applies B^+ -tree on PM to build an index for each object in disks. SLM-DB makes it possible to keep the LSM-tree as one level, avoiding intensive compaction overhead. BerkeleyDB [39] adopts a B-tree to store data in storage devices, whereas internal nodes reside in the memory, allowing read and write operations to be performed with a single I/O. MTree’s index structure employs B^+ -tree to manage large logs, ensuring high read/write efficiency and low memory cost.

9 Conclusion

In this article, we proposed a hierarchical data management approach referred to as *MTree* catered for tiering-based KV stores. MTree curtails the number of levels in the LSM-tree by forming a hierarchical structure that seamlessly integrates the log with LSM-tree: the log structure accumulates data written to the LSM-tree. In addition, we devised a long- and short-term key density distribution-aware partitioning scheme for the key range, thereby dynamically resizing buckets and BLogs to reduce the number of layers in the LSM-tree. The above approach offers the advantage of reducing the number of levels in the LSM-tree to almost a singular layer. Our comprehensive results offer compelling evidence that MTree possesses a clear edge over state-of-the-art techniques in terms of write amplification reduction and system performance enhancement.

References

- [1] Jung-Sang Ahn, Mohiuddin Abdul Qader, Woon-Hak Kang, Hieu Nguyen, Guogen Zhang, and Sami Ben-Romdhane. 2019. Jungle: Towards dynamically adjustable {Key-Value} store by combining {LSM-Tree} and {Copy-On-Write} {B+-Tree}. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'19)*.
- [2] Reed Allman. 2014. Rock Solid Queues @ Iron.io. Retrieved June 4, 2025 from <https://www.youtube.com/watch?v=HTjt6oj-RL4>
- [3] Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. 2009. Hash, displace, and compress. In *Proceedings of the European Symposium on Algorithms*. Springer, 682–693.
- [4] Betterbq. 2024. YCSB-OPENALEX. GitHub Repository. Retrieved June 4, 2025 from <https://github.com/betterbq/YCSB-OPENALEX>
- [5] Burton H. Bloom. 1970. Space/time tradeoffs in hash coding with allowable errors. *Communications of the ACM* 13, 7 (1970), 422–426.
- [6] Gianluca Bontempi, Souhaib Ben Taieb, and Yann-Aël Le Borgne. 2013. Machine learning strategies for time series forecasting. *Business Intelligence: Second European Summer School, eBISS 2012, Brussels, Belgium, July 15–21, 2012, Tutorial Lectures* 138, 2 (2013), 62–77.
- [7] Gerth Stølting Brodal and Rolf Fagerberg. 2003. Lower bounds for external memory dictionaries.. In *Proceedings of the SODA*. 546–554.
- [8] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, modeling, and benchmarking {RocksDB}{Key-Value} workloads at facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*. 209–223.
- [9] Yunpeng Chai, Yanfeng Chai, Xin Wang, Haocheng Wei, and Yangyang Wang. 2020. Adaptive lower-level driven compaction to optimize LSM-tree key-value stores. *IEEE Transactions on Knowledge and Data Engineering* 34, 6 (2020), 2595–2609.
- [10] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. 2018. Hashkv: Enabling efficient updates in {KV} storage via hashing. In *Proceedings of the 2018 {USENIX} Annual Technical Conference ({USENIX}{ATC}'18)*. 1007–1019.
- [11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems* 26, 2 (2008), 1–26.
- [12] Douglas Comer. 1979. Ubiquitous B-tree. *ACM Computing Surveys* 11, 2 (1979), 121–137.
- [13] Alex Conway, Martín Farach-Colton, and Rob Johnson. 2023. SplinterDB and maplets: Improving the tradeoffs in key-value store compaction policy. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.
- [14] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. 2020. {SplinterDB}: closing the bandwidth gap for {NVMe}{Key-Value} stores. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC'20)*. 49–63.
- [15] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems* 31, 3 (2013), 1–22.
- [16] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better space-time tradeoffs for LSM-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*. 505–520.
- [17] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. 2022. Spooky: Granulating LSM-tree compactions correctly. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3071–3084.
- [18] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 205–220.
- [19] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. 2020. ALEX: An updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 969–984.
- [20] Facebook. 2019. Rocksdb, a persistent key-value store for fast storage environments, 2019. [Online]. Available: <http://rocksdb.org/>
- [21] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. 2014. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. 75–88.
- [22] B. Fitzpatrick and A. Vorobey. 2011. Memcached: a distributed memory object caching system[EB/OL].(2011-1).
- [23] Shukai Han, Dejun Jiang, and Jin Xiong. 2020. Lightkv: A cross media key value store with persistent memory to cut long tail latency. In *Proceedings of the 36th International Conference on Massive Storage Systems and Technology (MSST'20)*.

- [24] Yupeng Hou, Jiacheng Li, Zhankui He, An Yan, Xiusi Chen, and Julian McAuley. 2024. Bridging language and items for retrieval and recommendation. *CoRR* (2024).
- [25] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, et al. 2015. BetrFS: Write-optimization in a kernel file system. *ACM Transactions on Storage* 11, 4 (2015), 1–29.
- [26] O. Kaiyrakhmet, S. Lee, B. Nam, et al. 2019. SLM-DB: Single-Level-Key-Value store with persistent memory. *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 191–205.
- [27] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for nonvolatile memory with NoveLSM. In *Proceedings of the 2018 {USENIX} Annual Technical Conference ({USENIX}{ATC}'18)*. 993–1005.
- [28] James M. Keller, Michael R. Gray, and James A. Givens. 1985. A fuzzy k-nearest neighbor algorithm. *IEEE Transactions on Systems, Man, and Cybernetics* SMC-15, 4 (1985), 580–585.
- [29] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*. 489–504.
- [30] K. Krishna and M. Narasimha Murty. 1999. Genetic K-means algorithm. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 29, 3 (1999), 433–439.
- [31] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. 2012. The vertica analytic database: C-store 7 years later. *Proceedings of the VLDB Endowment* 5, 12 (2012), 1790–1801.
- [32] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-performance concurrency control mechanisms for main-memory databases. *Proceedings of the VLDB Endowment* 5, 4 (2011), 327–338.
- [33] Leveldb. 2018. Fast and Lightweight Key/value Database Library by Google. Retrieved June 4, 2025 from <http://code.google.com/p/leveldb>
- [34] Pengfei Li, Yu Hua, Jingnan Jia, and Pengfei Zuo. 2021. FINEdex: A fine-grained learned index scheme for scalable and concurrent memory systems. *Proceedings of the VLDB Endowment* 15, 2 (2021), 321–334.
- [35] Lanyue Lu, Thanumalayan Sankaranarayanan Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage* 13, 1 (2017), 1–28.
- [36] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. 2021. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 243–262.
- [37] Fei Mei, Qiang Cao, Hong Jiang, and Jingjun Li. 2018. SifrDB: A unified solution for write-optimized key-value stores in large datacenter. In *Proceedings of the ACM Symposium on Cloud Computing*. 477–489.
- [38] N. Dayan and S. Idreos. 2019. The log-structured merge-bush and the wacky continuum. In *Proceedings of the 2019 International Conference on Management of Data*. 449–466.
- [39] Michael A. Olson, Keith Bostic, and Margo I. Seltzer. 1999. Berkeley DB. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*. 183–191.
- [40] OpenAlex. 2024. OpenAlex: An Open Index of Scholarly Works. Retrieved June 4, 2025 from <https://openalex.org>
- [41] Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 4, 33 (1996), 351–385.
- [42] Fengfeng Pan, Yinliang Yue, and Jin Xiong. 2017. dCompaction: Delayed compaction for the LSM-tree. *International Journal of Parallel Programming* 6, 45 (2017), 1310–1325.
- [43] William Pugh. 1990. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM* 33, 6 (1990), 668–676.
- [44] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 497–514.
- [45] Kai Ren and Garth Gibson. 2013. {TABLEFS}: Enhancing metadata efficiency in the local file system. In *Proceedings of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC}'13)*. 145–156.
- [46] Hui Sun, Guanzhong Chen, Yinliang Yue, and Xiao Qin. 2023. Improving LSM-Tree based key-value stores with fine-grained compaction mechanism. *IEEE Transactions on Cloud Computing* 11, 4 (2023), 3778–3796.
- [47] Chenlei Tang, Jiguang Wan, and Changsheng Xie. 2022. Fencekv: Enabling efficient range query for key-value separation. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 3375–3386.
- [48] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. 2020. XIndex: A scalable learned index for multicore data storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 308–320.

- [49] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based ultra-large key-value store for small data. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*. USENIX Association, 71–82.
- [50] Giorgos Xanthakis, Giorgos Saloustros, Nikos Batsaras, Anastasios Papagiannis, and Angelos Bilas. 2021. Parallax: Hybrid key-value placement in lsm-based key-value stores. In *Proceedings of the ACM Symposium on Cloud Computing*. 305–318.
- [51] Yahoo!. 2019. YCSB-C, a C++ Version of YCSB. Retrieved June 4, 2025 from <https://github.com/brianfrankcooper/YCSB>
- [52] Ting Yao, Jiguang Wan, Ping Huang, Xubin He, Qingxin Gui, Fei Wu, and Changsheng Xie. 2017. A light-weight compaction tree to reduce I/O amplification toward efficient key-value stores. In *Proceedings of the 33rd International Conference on Massive Storage Systems and Technologies (MSST'17)*. 1–13.
- [53] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with a matrix container in NVM. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. 17–31.
- [54] Yinliang Yue, Bingsheng He, Yuzhe Li, and Weiping Wang. 2016. Building an efficient put-intensive key-value store with skip-tree. *IEEE Transactions on Parallel and Distributed Systems* 28, 4 (2016), 961–973.
- [55] Qiang Zhang, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. 2023. The design and implementation of UniKV for mixed key-value storage workloads. *IEEE Transactions on Knowledge and Data Engineering* 35, 11 (2023), 11935–11949.
- [56] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. 2021. ChameleonDB: A key-value store for optane persistent memory. In *Proceedings of the 16th European Conference on Computer Systems*. 194–209.
- [57] Xingsheng Zhao, Song Jiang, and Xingbo Wu. 2021. WipDB: A write-in-place key-value store that mimics bucket sort. In *Proceedings of the 2021 IEEE 37th International Conference on Data Engineering (ICDE'21)*. IEEE, 1404–1415.

Received 12 June 2024; revised 30 April 2025; accepted 1 May 2025