The Journal of Supercomputing (2024) 80:23995–24025
https://doi.org/10.1007/s11227-024-06382-5

MTDB: an LSM-tree-based key-value store using a multi-tree structure to improve read performance

Xinwei Lin[1,2] · Yubiao Pan[1,2] · Wenjuan Feng[1] · Huizhen Zhang[1] · Mingwei Lin[3]

Accepted: 18 July 2024 / Published online: 1 August 2024

Abstract
Traditional LSM-tree-based key-value storage systems face significant read amplification issues due to the multi-level structure of LSM-tree, the unordered SSTable files in Level 0, and the lack of an in-memory index structure for key-value pairs. We observed that, under the influence of workloads with locality features, key-value pairs exhibit a range-specific access intensity. Addressing the three reasons for LSM-tree read amplification, we have utilized the range-specific access intensity of workload to propose a multi-tree structure consisting of a B+ tree, a single-level hot tree, and an LSM-tree with partition-based Level 0. This aims to enhance the read performance of LSM-tree-based key-value storage systems. We designed the prototype, MTDB, based on LevelDB. The experimental results show that MTDB's read performance is 1.62× to 2.02× that of LevelDB, and it approaches or exceeds the read performance of KVell and Bourbon while reducing memory overhead by 58.85%–86%.

Keywords Key-value store · Storage system · LSM-tree · Read amplification · Write amplification

1 Introduction

The Log-Structured Merge-tree (LSM-tree) [1] is a widely adopted data structure in current key-value storage systems, such as Google's BigTable [2] and LevelDB [3], Facebook's RocksDB [4] and Cassandra [5], and Apache's HBase [6].

The LSM-tree exhibits several features. Firstly, it turns random writes into sequential writes. Specifically, the LSM-tree initially caches key-value pairs from user's random writes in a MemTable. When the capacity of the MemTable reaches its limit, the MemTable is converted into an Immutable MemTable. The sorted data is then sequentially written to storage devices in the form of SSTables (Sorted String Tables). Secondly, the LSM-tree maintains a multi-level and ordered structure in

Extended author information available on the last page of the article external memory. Key-value pairs within each level are globally ordered (except for the Level 0), and the capacity of each level is limited and gradually increases (typically increased by a factor of 10, e.g., the capacity of Level i+1 is 10 times that of Level i). Lastly, key-value pairs always move from the upper levels to the lower levels of the LSM-tree. For instance, when a certain Level i reaches its capacity limit, the LSM-tree triggers a Major Compaction operation. This operation selects SSTable files from Level i and all SSTable files in Level i+1 that overlap in key ranges. These files are then read into memory, sorted, one or more new SSTable files is generated, written back to Level i+1, and the selected SSTable files are deleted.

Due to the characteristics of the LSM-tree, key-value storage systems based on LSM-tree suffer from a significant issue known as read amplification. This problem is primarily attributed to three reasons: (1) The multi-level structure of the LSM-tree necessitates the key-value storage system to traverse through levels from top to bottom to locate the required key-value pairs, continuing until the target pair is found or the absence of the specified key-value pair is confirmed. (2) Unlike other globally ordered levels, the SSTable files in Level 0 exhibit overlaps. Consequently, when searching for a target key-value pair in Level 0, the system needs to check all SSTable files in Level 0 (up to a maximum of 12 files), whereas other levels only require checking one SSTable file. (3) Since the key-value pairs in SSTable files lack in-memory indexes, checking SSTable files involves reading their index blocks, Bloom filter blocks, and data blocks to ascertain whether the desired key-value pairs can be located. Research indicates that LSM-tree-based key-value storage systems sometimes face read amplification as high as 300 [7, 8], leading to a decline in read performance. In real-world scenarios, the majority of database workloads involve more reads than writes. For example, the read-to-write ratio in AI/ML applications can reach 9:1, in OLTP applications it can be as high as 10:1, and in Twitter's Memcached workload, the ratio can reach 7:1 [9].

Therefore, addressing the issue of read amplification in key-value storage systems based on LSM-tree has become an interesting research issue in this field. In response to the aforementioned challenges, one category of research [10, 11] utilizes a global in-memory indexing structure, indexing all key-value pairs in the LSM-tree. Although this approach eliminates the efficiency gap between early and recent writes, it incurs high memory indexing costs for the large number of cold key-value pairs under skewed workloads, yielding minimal benefits. Another category of research [12] seeks to enhance the retrieval efficiency of lower-level data by moving frequently accessed SSTable files from lower levels to upper levels or into memory. However, since read operations typically require traversing all SSTable files in Level 0, relocating SSTable files to Level 0 can exacerbate read amplification. There is also a category of work [13] that addresses read amplification by implementing Bloom filters. While this approach helps mitigate read amplification, it still fails to resolve the challenges associated with multi-level access.

Research indicates that key-value storage system workloads commonly exhibit pronounced locality characteristics [14–16]. Due to the sequential arrangement of key-value pairs, under the influence of evident workload locality characteristics, there exists a range-specific access intensity for key-value pairs. In other words, certain ranges of key-value pairs within the workload experience higher access frequencies, while others within different ranges are accessed less frequently. Therefore, to reduce the memory overhead of global indexing, this study leverages the range-specific access intensity characteristics of key-value pairs in workloads, using in-memory indexes only for key-value pairs with range-specific hotness. This approach avoids the memory overhead caused by cold key-value pairs in global indexing. At the same time, it addresses the three causes of read amplification in LSM-tree by proposing a multi-tree structure to enhance the read performance of LSM-tree-based key-value storage systems. Specifically, our contributions in this paper are as follows.

- To minimize the overhead of LSM-tree's top-down level-by-level search for required key-value pairs in external memory, this paper introduces a multi-tree structure, namely an in-memory B+tree, an on-disk single-level hot tree, and an on-disk three-level LSM-tree. Newly entered key-value pairs, with frequently accessed feature, reside temporarily in the B+tree. As the access frequency decreases for certain key-value pairs, they are flushed to the on-disk LSM-tree. When the access frequency for key-value pairs within certain ranges increases in the LSM-tree, those with range-specific access intensity are moved to the on-disk hot tree. This multi-tree structure dynamically allocates key-value pairs to different structures based on workload variations, thereby enhancing read performance.
- To address the problem of traditional key-value storage systems having to check all SSTable files in Level 0, this paper designs a partition-based Level 0 for the three-level LSM-tree. Specifically, we adopt a partitioned design for Level 0 based on key ranges, where key-value pairs between different partitions are ordered, allowing overlapping key ranges within each partition. Additionally, each partition in Level 0 is limited to a maximum of 4 SSTable files. Consequently, when checking Level 0, the system only needs to examine up to 4 SSTable files.
- We do not create in-memory indexes for all key-value pairs. Instead, we create in-memory indexes only for key-value pairs with range-specific access intensity to improve their read performance. Specifically, we generate in-memory indexes for range-specific key-value pairs in the on-disk hot tree and record them in the B+tree. This allows for direct reading of these key-value pairs through the in-memory indexes stored in the B+tree, avoiding additional operations like reading index blocks, Bloom filter blocks, and data blocks. Furthermore, the transfer of range-specific key-value pairs from LSM-tree to the on-disk hot tree may introduce write amplification, so we design this movement as a logical operation.
- Finally, we design reasonable data flows in the multi-tree structure and implement the prototype system MTDB (Multi-Tree Database). We evaluate the design using standard database evaluation tools, Yahoo! Cloud Serving Benchmark (YCSB) [17], and Rocksdb-benchmark-Mixgraph (RBM) [4], comparing it with KVell [11], WiredTiger [26], Bourbon[27] and LevelDB [3] in terms of throughput, read amplification, CPU and memory usage, and access overhead. Experimental results validate the effectiveness of our design.

The remainder of this paper is structured as follows. Section 2 provides an introduction to the background and research motivation of this paper. Section 3 outlines the specific details of MTDB, the system we designed. Section 4 presents our experimental design and results. Section 5 discusses MTDB's challenges and future work. Section 6 discusses research work relevant to this paper. Finally, Sect. 7 offers a summary and conclusion of this paper.

2 Background and motivation

2.1 LSM-tree

The LSM-tree exhibits excellent write performance by turning random write operations into sequential write operations. LevelDB [3] is one of the most popular key-value storage databases based on LSM-tree. Figure 1 illustrates the overall architecture of LevelDB. In memory, LevelDB utilizes a SkipList to implement the MemTable and Immutable MemTable. On external storage (e.g., SSD), all SSTable files are organized in a multi-level structure, including seven levels represented by Level 0 to Level 6. Each SSTable contains several Data Blocks, an Index Block, and several Meta Blocks. The Data Block contains key-value pairs, the Index Block is used to index the Data Block, and the Meta Block includes extended features such as Bloom filters. The write process for new key-value pairs is as follows: the key-value pairs are first appended to the SSD's Log for crash recovery, then added to the MemTable, which is sorted by key. Once the MemTable is full, it is converted into an Immutable MemTable and flushed to Level 0 on SSD. When Level i is full,

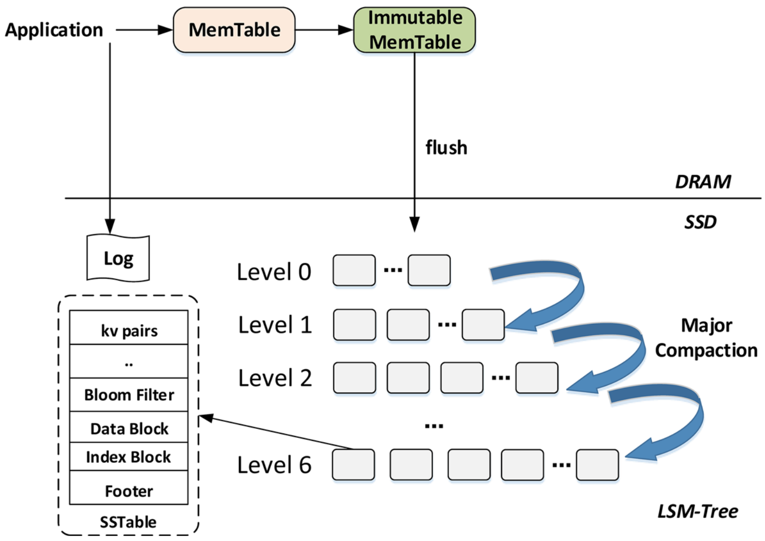


Fig. 1 The architecture of LevelDB

the key-value pairs in Level i undergo Major Compaction to merge into Level i+1. Specifically, the Major Compaction operation reads overlapping data from adjacent levels, merges them, generates a new SSTable file, and writes it back to Level i+1. Clearly, this process introduces additional I/O operations and leads to write amplification. The read process involves searching through multiple SSTables. Specifically, the read process for key-value pairs is as follows: the system first checks the in-memory structures, then searches all SSTables in Level 0 and one SSTable in each remaining level until it finds the data or reaches the bottommost level. Despite the use of Bloom filters [18], due to their false positives and limited memory for caching Bloom filters, each read still requires multiple I/O operations, resulting in read amplification.

The factors contributing to read amplification in LSM-tree can be mainly attributed to three aspects. Firstly, the hierarchical structure and search mechanism in LSM-tree reduce read efficiency. The multi-level structure of LSM-tree requires searching through multiple SSTable files from the first level to the last when looking for the required key-value pairs. As the number of levels increases, read amplification becomes more severe. Secondly, to ensure write performance, key-value pairs are flushed from memory directly to Level 0 in the form of SSTable files without triggering Major Compaction operations. This results in key range overlaps among SSTable files in Level 0, ultimately requiring the examination of all SSTable files in Level 0 during read operations. Thirdly, due to the lack of in-memory indexes for key-value pairs in SSTable files, examining SSTables involves reading their index blocks, Bloom filter blocks, and data blocks. Research indicates that, in the worst-case scenario, LevelDB and WiscKey may need to inspect up to 14 and 10 SSTable files, respectively [7]. In LevelDB, the number of SSTable files to be checked in Level 0 is approximately equal to the sum of the SSTables in all other levels.

2.2 Motivation

Current real-world workloads exhibit highly skewed characteristics. Research by Facebook shows that in their distributed key-value storage engine ZippyDB, approximately 1% of key-value pairs handle 50% of the read operations, with each of these key-value pairs being accessed more than 100 times. In contrast, about 73% of key-value pairs are accessed only once [15]. Observations in Alibaba's production environment indicate that 50% (in daily cases) to 90% (in extreme cases) of accesses touch only 1% of the total items [19].

In applications like AppsFlyer [20], Flurry [21], Google Firebase [22], and social networks such as Twitter [16], data items have key prefixes with varying access intensities. Common key prefixes result in higher access frequencies for key-value pairs within certain key ranges. Due to the sequential arrangement of key-value pairs, these applications exhibit range-specific access intensity in read operations.

To validate the existence of range-specific access intensity, we conducted experiments and performed statistical analysis on workloads generated by YCSB [17]. We generated a load of one million requests following a Zipf distribution with a Zipfian constant of 0.99, partitioned into 90 subspaces based on different key

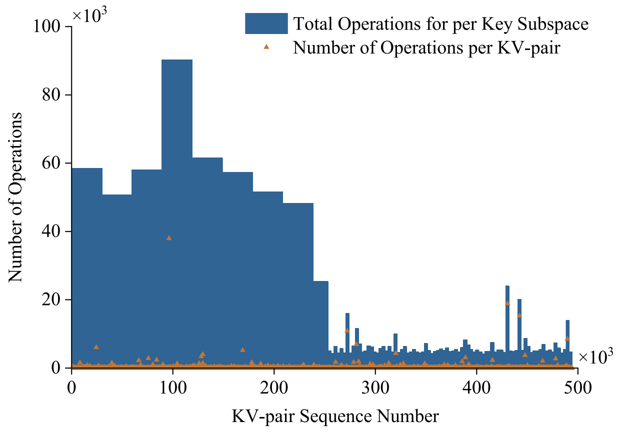


Fig. 2 The number of operations on KV Pairs and the distribution of operations in subspaces under YCSB workloads

prefixes. We then recorded the access frequency for each key-value pair and the access frequency for each subspace. Figure 2 presents the experimental results. We assigned numerical identifiers to key-value pairs in ascending order of character sequence. The orange triangles represent the access frequencies for each key-value pair, and each bar in the histogram represents the total access frequency for key-value pairs with the same key prefix. The results indicate that among the 90 subspaces, the first 9 subspaces encompass nearly half of the key-value pairs. This suggests that the workload exhibits spatial locality characteristics, where certain key prefix data have higher operational frequencies. Furthermore, examining the heatmaps of the subspaces reveals significant differences in the total access frequencies among different subspaces. Therefore, key-value pair access has range-specific access intensity, meaning that the frequency of accessing key-value pairs within certain ranges is higher, while the frequency is lower for key-value pairs within other ranges.

From the above discussion, we observe that optimizing read performance in LSM-tree by leveraging the range-specific access intensity feature of key-value pair accesses is an intriguing problem. Therefore, this paper proposes a solution to mitigate the read amplification caused by LSM-tree's top-down level-by-level access. The approach involves keeping the hottest data in memory and transferring key-value pairs with range-specific access intensity from the LSM-tree to a separate single-level hot tree. Additionally, the paper suggests designing a partition-based Level 0 where limits the number of SSTables within each partition to reduce the read amplification. Finally, the paper proposes creating in-memory indexes for key-value pairs with range-specific access intensity residing on the hot tree to further accelerate their read performance. The details of these design strategies will be elaborated in the next section.

3 MTDB design

In this section, we begin by introducing the system architecture of MTDB that we have designed. Subsequently, we provide a detailed description of the primary data structures and the data flow of MTDB. Finally, we discuss several issues in practical implementation.

3.1 Architecture overview of MTDB

Figure 3 illustrates the system architecture of MTDB that we have designed. The main concept behind the MTDB architecture is to leverage the range-specific access intensity. It involves designing a multi-tree structure to store key-value pairs with different hotness levels on separate trees. This approach aims to alleviate the performance degradation in reads caused by the traditional LSM-tree's multi-level searches, Level 0's SSTable file checks, and the lack of in-memory indexes.

Specifically, MTDB incorporates three crucial design principles: (1) MTDB treats key-value pairs with range hotness differently. The hottest range of key-value pairs is retained in the B+tree, the coldest ones are stored in the three-level LSM-tree, and dynamically changing key-value pairs within a moderately hot range are moved to the single-level hot tree. This differential treatment of key-value pairs with varying hotness levels helps alleviate the performance decline in reads caused by the multi-level searches in traditional LSM-tree. (2) MTDB designs a partition-based Level 0 for the three-level LSM-tree. In Level 0, MTDB introduces Guards to partition the KV pairs, and the B+tree flushes key-value pairs to the specified partition based on each partition's key range. Additionally, MTDB limits the number of SSTables in each partition to 4. Due to these designs, the partitions in Level 0

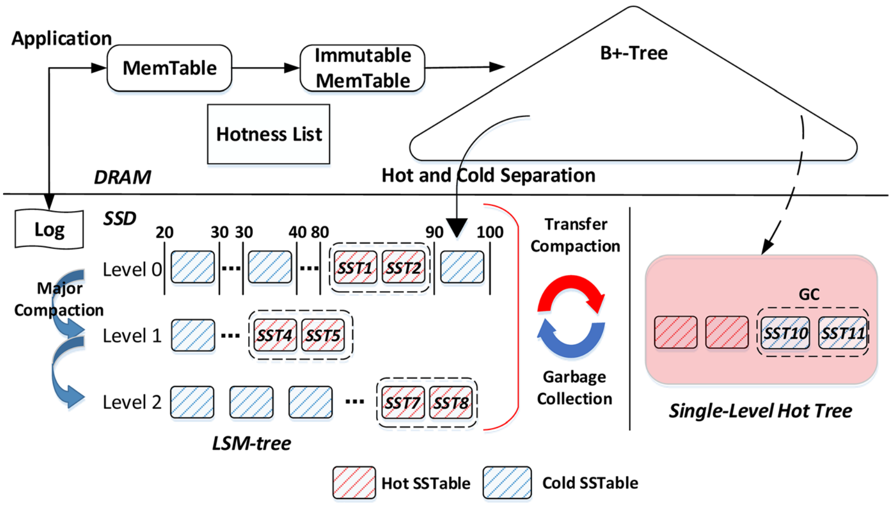


Fig. 3 The architecture of MTDB

are ordered, and the 4 SSTables within each partition allow for overlapping. Consequently, each search for key-value pairs in Level 0 is constrained by MTDB to a maximum of 4 SSTables in a specific partition, reducing the Level 0 search overhead in traditional LSM-tree. (3) As SSTables lack in-memory indexes, traditional LSM-tree checks for SSTables involve reading index blocks, bloom filter blocks, and data blocks. To boost the reading process of hot key-value pairs in SSTables, MTDB creates in-memory indexes for key-value pairs in the single-level hot tree through Transfer Compaction, storing them in the B+tree. MTDB achieves this by consuming a small amount of memory to accelerate the reading of frequently accessed hot key-value pairs. The single-level hot tree does not perform Compaction. Instead, it uses Garbage Collection to reclaim cold key-value pairs back to the LSM-tree.

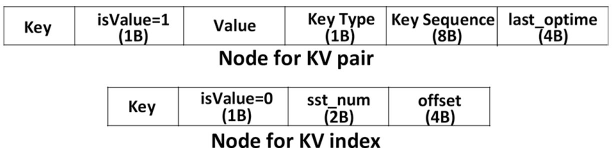
3.2 Data structure

To implement the aforementioned design principles, the system architecture of MTDB retains traditional components such as MemTable, Immutable MemTable, and Log from LSM-tree-based key-value storage systems. Additionally, MTDB introduces new components and related designs, including a B+tree, a single-level hot tree, and a three-level LSM-tree based on partition-based Level 0. In the following subsections, we will provide a detailed introduction to each of the new components and describe the data flow processes between them.

3.2.1 B+tree

Unlike LevelDB, which directly flushes Immutable MemTable to SSD, MTDB caches key-value pairs from Immutable MemTable in the B+tree first. Additionally, MTDB creates in-memory indexes for key-value pairs in the hot tree and stores them in the B+tree. Therefore, the B+tree has two types of leaf nodes: one type is called key-value nodes, which retain the latest and frequently operated key-value pairs, and the other type is called key-value index nodes, which provide indexes for key-value pairs in the hot tree.

Figure 4 illustrates the structure of these two types of leaf nodes. “key" and the identifier “isValue" are fixed fields in each leaf node. When the leaf node stores key-value pairs, MTDB allocates fields such as “value", “key type" (1 Byte), “key sequence" (8 Bytes), and “last_optime" (4 Bytes). The “key type" and “key sequence" are the elements required to form an internal key in LevelDB. “last_optime" is used to record the time of the most recent operation on the key-value pair. Therefore, when a key-value pair undergoes insert, read, or update operations,



Key	isValue=1 (1B)	Value	Key Type (1B)	Key Sequence (8B)	last_optime (4B)

Node for KV pair

Key	isValue=0 (1B)	sst_num (2B)	offset (4B)

Node for KV index

Fig. 4 Two different types of B+tree leaf nodes

its “last_optime" will be initialized or updated. When the leaf node stores indexes (pointers) for key-value pairs in the hot tree, it allocates “sst_num" and “offset" fields. “sst_num" takes up 2 Bytes, indicating the file number of the SSTable where the target key-value pair is located, and “offset" takes up 4 Bytes, indicating the offset of the target key-value pair's data block relative to the SSTable file. By using “sst_num" and “offset" it is possible to quickly locate the data block corresponding to the target key-value pair. Additionally, we allocate separate memory spaces for the leaf nodes storing key-value pairs in the B+tree and the leaf nodes storing key-value indexes. This makes it easier to control the additional memory overhead caused by the introduction of the B+tree.

3.2.2 Single-level hot tree

We have implemented a single-level hot tree in MTDB, which receives key-value pairs from the LSM-tree. When MTDB identifies certain ranges of key-value pairs within the workload experience higher access frequencies, it isolates all SSTables belonging to that key range across the levels of the LSM-tree into the hot tree. It's important to note that this isolation is logical, meaning all SSTables isolated to the hot tree are not rewritten. Instead, MTDB system records the metadata of these SSTables in the hot tree. Specifically, MTDB writes the metadata information of these SSTables from the LSM-tree's Manifest into the hot tree's Manifest and removes them from the LSM-tree, achieving logical isolation of SSTables into the hot tree. The total space of this single-level hot tree is determined by a configurable parameter, Ratio $_{\text{hot\_tree}}$, which represents the percentage of the total database size that the hot tree occupies. When the size of this single-level hot tree is determined, the size of the leaf nodes in the B+tree that stores indexes for key-value pairs in the hot tree is also determined.

3.2.3 Three-level LSM-tree with Partition-based Level 0

We have designed a three-level LSM-tree with partition-based Level 0. Specifically, we leverage the Guard concept from SkipList [23, 24] to partition the entire key space in Level 0 of the LSM-tree into multiple subspaces. Two adjacent Guards constrain the key range of a subspace in Level 0, and using Guards to organize data ensures that each subspace only contains key-value pairs within a specific key range. Furthermore, we set a maximum limit of 4 SSTables for each subspace in Level 0, which is fewer than the original limit of 12 in Level 0. By reducing the number of SSTables that need to be searched in Level 0, we aim to decrease the read amplification in Level 0. Initially, MTDB sets several Guards (default is 100), and when a subspace performs one Compaction operation, it indicates that the workload is concentrated in this subspace, exhibiting a certain range-specific access intensity. After the completion of one Compaction in the subspace, MTDB adds a new Guard to evenly partition this key range, obtaining smaller granularity subspaces. For applications in key-value storage systems, the character composition of their key-value pairs has fixed characteristics. For example, in SQL database storage engine UDB, the keys mostly consist of the

4-byte MySQL table index, two object IDs, the object type, and other information [15]. In blockchain systems, different types of key-value pairs have different fixed key prefixes [25]. By adding new Guards, the system can quickly adapt to the key-value composition characteristics in the application load and perform finer-grained partitioning of hot key subspaces. When the number of Guards reaches the upper limit (default is 500), no further changes will occur. Finally, we limit the LSM-tree to three levels to reduce the overhead of LSM-tree's level-by-level search.

3.2.4 Data flow among multiple trees

1)B+tree → LSM-tree:

When the B+tree's leaf nodes, which store key-value pairs, reach the capacity limit, the B+tree performs a flushing operation. At this point, MTDB scans all key-value nodes in the B+tree, using data sampling and hotness evaluation to filter out key-value pairs. The key-value pairs with lower hotness are then written into the LSM-tree.

Specifically, MTDB conducts several random accesses in the B+tree to sample key-value nodes and obtain the last_optime of these sampled nodes. The $\text{last\_optime}_{\text{average}}$ is calculated and used as the overall last_optime for the B+tree. Subsequently, MTDB traverses all key-value nodes in the B+tree. By comparing the last_optime of each key-value pair with the $\text{last\_optime}_{\text{average}}$, key-value pairs with a last_optime lower than the $\text{last\_optime}_{\text{average}}$ are flushed to Level 0 of the LSM-tree. It's worth noting that the above operation is passively terminated based on SSTable size limits or when crossing Guards to ensure generated SSTables do not cross two subspaces in Level 0.

During each scan of the B+tree, approximately half of the key-value nodes are flushed to Level 0, leaving enough buffer space to accommodate new write operations.

2)LSM-tree → Hot tree:

To logically isolate SSTables with range-specific access intensity to a single-level hot tree, we designed the Transfer Compaction operation. First, we set up a 70KB Hotness List in memory, which records the access frequency of each subspace defined in Level 0. When triggering one Transfer Compaction, we calculate the range hotness of each subspace, defined as the access frequency of that subspace divided by its key range.

We have two different triggering mechanisms for Transfer Compactions based on mixed read-write workloads and read-intensive workloads:

- For mixed read-write workloads, when a subspace in Level 0 reaches the SSTable quantity limit, MTDB utilizes the Hotness List and calculates the range hotness for each subspace. If the range hotness of the subspace that needs Compaction ranks in the bottom percentage in terms of hotness among all subspaces and is less than the $\text{Ratio}_{\text{hot\_tree}}$, MTDB triggers a Transfer Compaction, logically isolating SSTables within this key range to the single-level hot tree. Otherwise, MTDB triggers the traditional Major Compactions, moving SSTables from Level 0 to Level 1.

- For read-intensive workloads, MTDB calculates the read-to-write ratio of the workload over a certain period. When the read-to-write ratio is too high (e.g., exceeding 90%), the system calculates the range hotness for all subspaces every 1000 read operations and selects the subspace with the highest range hotness for Transfer Compactions.

The participants in Transfer Compaction are all data within a selected subspace in the LSM-tree. Specifically, all SSTables in Level 0 corresponding to the subspace will be selected. In Level 1 and Level 2, all SSTables related to the key range of this subspace will be selected. Subsequently, one Transfer Compaction copies the metadata of these SSTables from the LSM-tree's Manifest to the single-level hot tree's Manifest and finally removes them from the LSM-tree's Manifest. Therefore, the Manifest in the hot tree records SSTable number, Minkey, and Maxkey similarly to the LSM-tree. Finally, the Transfer Compaction creates memory indexes for all key-values in the hot tree and stores them in B+tree key-value index nodes. Since SSTables from different levels in the LSM-tree contribute to these steps, effective handling of new and old versions is required. Specifically, we read these SSTables into memory, merge them, and then use SSTable file numbers and data block offsets to create and maintain indexes in the B+tree, ensuring that only the latest version of key-value pairs are indexed.

3)Hot tree → LSM-tree:

We control the size of the single-level hot tree and, consequently, limit the overhead of memory indexes by setting the $\text{Ratio}_{\text{hot\_tree}}$. When the total capacity of SSTables logically isolated to the hot tree reaches the specified limit, MTDB triggers a garbage collection (GC) operation to move the data from the hot tree back to the LSM-tree.

Specifically, MTDB performs a range hotness calculation based on the Hotness List for all subspaces contained in the hot tree when triggering GC. It selects the coldest subspace range, releases all key-value index nodes corresponding to this range in the B+tree, and rewrites the key-value pairs of this range back to Level 0 of the LSM-tree. Since the leaf nodes of the B+tree are ordered, performing a range scan for this subspace in the B+tree allows for a quick completion of the GC operation. Once the coldest key range is identified, MTDB traverses the key-value index nodes within this range in the B+tree. It retrieves the key-value pairs through the index, generates SSTables, and writes them into the corresponding partition of Level 0 in the LSM-tree. Finally, MTDB delete the corresponding SSTables.

3.2.5 Reading and writing process

Figure 5 illustrates the detailed example of the write and read processes in MTDB.

1)Writing process:

Taking Fig. 5 as an example, let's discuss the write process in MTDB. Please refer to Algorithm 1 for the specific write process.

STEP ① Write to the Log.

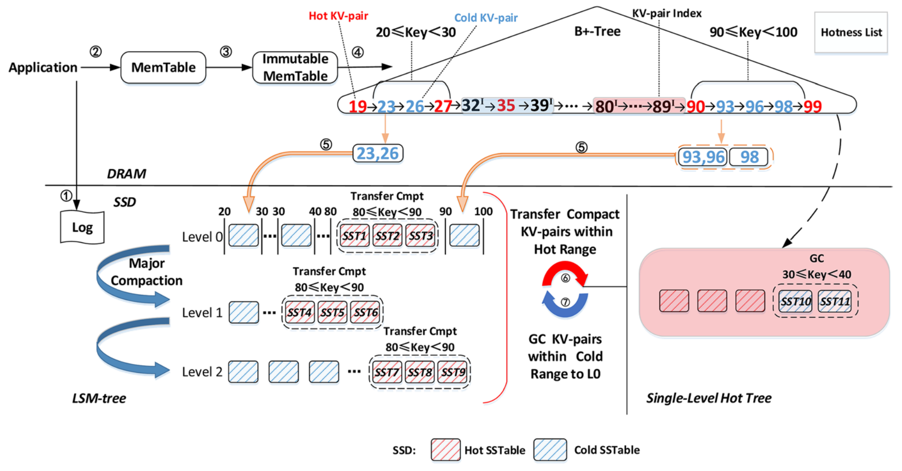


Fig. 5 Write and read examples in MTDB

STEP ② Batch write to MemTable.

STEP ③ MemTable transforms into Immutable MemTable.

STEP ④ Immutable MemTable moves to B+tree. For update requests, the new version of the value directly replaces the old version in the B+tree, achieving in-place updates. For delete requests, the tombstone marker for the key is recorded in the B+tree, and then written into LSM-tree.

STEP ⑤ Traverse key-value nodes in the B+tree and use a sampling strategy to flush cold data to the corresponding partition (subspace) in Level 0. As shown in Fig. 5, if there exists a subspace with $20 \leq \text{Key} < 30$ in Level 0, this range scan in the B+tree will start from the smallest key greater than or equal to 20, i.e., 23. Through hotness calculation, the key-value pair 23 with lower hotness, its last_optime is less than the last_optime$_{\text{average}}$, will be added to the SSTable waiting to be written to SSD. Similarly, when the SSTable is not at its size limit, key 26 will also be added to the SSTable. Key 27 with high hotness will continue to be retained in the B+tree. The termination conditions for the range scan include whether the SSTable has reached its size limit. The next flush operation will continue from the interrupted position after the thread is awakened and will scan the entire key range partition, which can refer to the example when handling the subspace with $90 \leq \text{Key} < 100$.

STEP ⑥ When the number of SSTables in a partition (subspace) in Level 0 of LSM-tree reaches the limit, range hotness calculation will be performed to determine the specific Compaction strategy. If the subspace has high hotness, the Transfer Compaction will be triggered. One Transfer Compaction creates B+tree indexes for hot SSTables, and logically isolates hot SSTables into the hot tree. As shown in Fig. 5, when the number of SSTables in the range $80 \leq \text{Key} < 90$ in LSM-tree Level 0 reaches the limit, a range hotness calculation is performed, and the hotness for the range $80 \leq \text{Key} < 90$ reaches the requirement for logical isolation. MTDB triggers the Transfer Compaction operation for the subspace $80 \leq \text{Key} < 90$, selecting SST1, SST2, and SST3 in Level 0, and SST4–SST9 in Level 1 and Level 2. Next, these nine SSTables undergo key-value pair merging. For Key 80, the generated index 80^{I} is saved in the B+tree. The indexes of key-value pairs in the remaining range are also saved in the B+tree, using a flag to distinguish between key-value pairs and index structures. Finally, these SST1–SST9 are logically isolated from LSM-tree into the hot tree. If the subspace has low hotness, Major Compaction is triggered. Major Compaction searches Level 1 for SSTables overlapping with the subspace key range. It merges the key-value pairs from the subspace in Level 0 into Level 1.

STEP ⑦ When the hot tree space ratio reaches the limit, $\text{Ratio}_{\text{hot\_tree}}$, GC is triggered. A coldest subspace is selected for GC, freeing the B+tree indexes corresponding to key-value pairs in this coldest range and rewriting the key-value pairs back to LSM-tree Level 0. As shown in Fig. 5, when the range hotness for the range $30 \leq \text{Key} < 40$ in the hot tree is the lowest, GC is performed for SST10 and SST11 in this range. Since the B+tree can perform in-place update operations on leaf nodes marked with an index, there may be expired data in SST10 and SST11. For example, key 35 may have been an index in SST10 or SST11, and after the in-place update, the index for key 35 is overwritten by the new key-value pair. To ensure data correctness during GC and reduce GC overhead, we only need to perform a range scan on the B+tree for key-value index nodes. Through a range scan of $30 \leq \text{Key}^{I} < 40$, we can identify leaf nodes with index markers in the range 30 to 40. These indexes point to valid key-value pairs in SST10 and SST11 and represent the latest version in MTDB. After reading the corresponding key-value pairs from the data block based on the index, SSTable is generated and written into Level 0 within the partition $30 \leq \text{Key} < 40$.

The process described above illustrates the write and compaction operations in MTDB, showcasing how it effectively manages data through various stages.

Algorithm 1 Write operation

```
1:  Input: $KVs$
2:  STEP ①$KVs \rightarrow Log$
3:  STEP ②$KVs \rightarrow$ MemTable
4:  STEP ③MemTable $\rightarrow$ Immutable MemTable
5:  STEP ④Immutable MemTable $\rightarrow B + tree$
        STEP ⑤
6:  if $B$+$tree$ size reaches limit then
7:      while $node.key \neq Guard$ and $SST.size \leq$ Limit do
8:          for each key-value node in $B$+$tree$ do
9:              if $last\_optime \leq last\_optime_{\mathrm{average}}$ then
10:                 Flush cold $KVs$ to Level 0
11:             else
12:                 Keep hot $KVs$ in $B$+$tree$
13:             end if
14:         end for
15:     end while
16: end if
        STEP ⑥
17: if $SSTables$ in $subspace_i$ in Level 0 $\geq$ 4 then
18:     for each $subspace$ in MTDB do
19:         Calculate range hotness
20:     end for
21:     if hotness of $subspace_i \leq Ratio_{\mathrm{hot\_tree}}$ then
22:         for $SSTables$ in $subspace_i$ do
23:             Remove $sst\_info$ from LSM-tree
24:             Add $sst\_info$ to Hot Tree
25:             for $KVs$ in SSTable do
26:                 Insert $KV$ index into $B + tree$
27:             end for
28:         end for
29:     else
30:         Major Compaction for $subspace_i$
31:     end if
32: end if
        STEP ⑦
33: if Hot Tree size $\geq Ratio_{\mathrm{hot\_tree}}$ then
34:     for each $subspace$ in MTDB do
35:         Calculate range hotness
36:     end for
37:     Choose the subspace with lowest hotness
38:     for $KVs$ in chosen subspace do
39:         Read $KV$ by index in $B + tree$
40:         Flush $KV$ to Level 0
41:     end for
42: end if
```

2)Reading process:

The read operation is relatively straightforward, and the specific process is illustrated in Algorithm 2. The read request proceeds by sequentially searching through the Memtable, Immutable MemTable, B+tree, and LSM-tree. Upon hitting a leaf node in the B+tree, the algorithm first checks the type of the leaf node. If it stores a key-value pair, the result is directly returned. If it stores an index for key-value pairs in the single-level Hot Tree, the algorithm reads the target data block from the hot tree based on the SSTable Number and data block offset and performs a binary search to locate the value, then returns the result. In case of a miss in the B+tree, the algorithm proceeds to search in the LSM-tree from top to bottom. The LSM-tree searches each level for SSTables that might contain the target key-value pair. It checks up to four SSTables in the partitioned Level 0 and up to one SSTable in the remaining levels. The search is conducted by reading and searching the Index Block and Data Block of the SSTables to determine if they contain the target key-value pair. If the requested key-value pair is not found in the last layer of the LSM-tree, it returns a “not found" result.

Algorithm 2 Read operation

```
1:  Input: Key
2:  if key is in MemTable then
3:      Get value from MemTable
4:  end if
5:  if key is in Immutable MemTable then
6:      Get value from Immutable MemTable
7:  end if
8:  if key is in B+tree then
9:      if $node.isValue = 1$ then
10:         Get value from node
11:     else if $node.isValue = 0$ then
12:         Get value by index
13:     end if
14: else
15:     for each level in LSM-tree do
16:         if key is found in level then
17:             Get value from SSTable
18:         end if
19:     end for
20: end if
```

3.3 Implementation issues

3.3.1 Serialization and deserialization of B+tree

When the database is closed, all node information in the B+tree is serialized to SSD. Upon reopening the database, the in-memory B+tree structure is restored by deserializing those information. In our experiments, we allocated 150MB of memory space for the B+tree, and both the serialization and deserialization times were less than 1 s.

3.3.2 Crash consistency

To ensure the robustness of MTDB, a crash recovery mechanism is essential for the B+tree to maintain consistency. The B+tree comprises two types of leaf nodes: key-value nodes and key-value index nodes.

For key-value nodes, MTDB employs the MemTable Log and our designed Read Log to achieve crash recovery. During a round of transferring B+tree data to the LSM-tree, MTDB traverses all key-value nodes from beginning to end, compares their last_optime with the $\text{last\_optime}_{\text{average}}$, and flushes key-value pairs with last_optime lower than the $\text{last\_optime}_{\text{average}}$ to Level 0 of LSM-tree.

MTDB delays the deletion of MemTable Log. The deletion mechanism used is that after a round of data transfer is completed, the Memtable Log with a creation time less than $\text{last\_optime}_{\text{average}}$ can be safely deleted. The key-value pairs in these Memtable Log may have undergone three types of operations in the B+tree: (1) Since the last_optime of the key-value pairs is less than $\text{last\_optime}_{\text{average}}$, they have already been transferred to the LSM-tree. (2)The key-value pairs have undergone in-place updates. The updated key-value pairs are written to new Memtable Log, making the key-value pairs in the original Memtable Log invalid. (3)Due to read operations, the last_optime of the key-value pairs is greater than $\text{last\_optime}_{\text{average}}$, so they are retained in the B+tree. To safely delete the Memtable Log and reduce space amplification and crash recovery costs, we designed a Read Log to record key-value pairs and their corresponding read operation times. The same, after a round of data transfer is completed, the records in the Read Log with operation times less than $\text{last\_optime}_{\text{average}}$ can also be safely deleted.

Therefore, only recording the $\text{last\_optime}_{\text{average}}$ for each flush is necessary for B+tree recovery. When restoring B+tree key-value nodes, the MemTable Log, used for write requests, and Read Log is scanned, timestamps are compared, and data with timestamps greater than or equal to the recorded $\text{last\_optime}_{\text{average}}$ are recovered.

For key-value index nodes, we can reconstruct the index based on the metadata information of the hot tree. By traversing all SSTables in the hot tree and filtering the latest versions of key-value pairs, the index can be rebuilt.

Table 1 YCSB Core workloads

Workload	A	B	C	D	E	F
Operations	R: 50%	R: 95%	R: 100%	R: 95%	S: 95%	R: 50%
	U: 50%	U: 5%		I: 5%	I: 5%	M: 50%
Req. Dist	Zipfian	Zipfian	Zipfian	Latest	Zipfian	Zipfian

R: Read, U: Update, I: Insert, S: Scan, M: Read-Modify-Write

In summary, during crash recovery, MTDB performs a merge operation on the hot tree and all log files, including MemTable Log and Read Log. This ensures MTDB's crash consistency and data persistence.

4 Evaluation

In this section, we conducted benchmark tests on MTDB using the Yahoo! Cloud Serving Benchmark (YCSB) tool to simulate real-world workloads and the Rocksdb-benchmark-Mixgraph (RBM) workload which simulate realistic social-graph workloads as developed by Facebook. The system evaluation was performed on hardware with an Intel i7-11700 processor, 32GB DRAM, and a 512GB SAMSUNG 980 PRO SSD, running Ubuntu 20.04 LTS.

4.1 Experimental setup

We conducted a comparative analysis between MTDB and four prominent key-value storage systems: Google's LevelDB [3], which is based on LSM-tree; WiredTiger [26], which uses a disk-based B+tree; KVell [11], which employs a global in-memory B+tree index; and Bourbon [27], which utilizes a key-value separation with a learned index. To implement MTDB, we added approximately 3500 lines of code on top of Google's LevelDB [1], creating a prototype. In MTDB, we reserved 50MB for key-value nodes of the B+tree as a buffer for segregating newly written hot and cold key-value pairs. The $Ratio_{hot\_tree}$ was set to 30%, with a 70MB reservation for hot tree indexing. Additionally, considering the internal structure overhead of the B+tree, we limited the overall memory usage of the B+tree to 150MB, with an 8MB cache, and configured the size of Level 1 to 1GB. For LevelDB,Bourbon and WiredTiger, we configured the Memtable size to 32MB and the cache to 128MB to balance memory overhead. The other parameters in the four systems use default configurations. KVell's memory overhead, including indexes and page cache, was minimized. In workload testing, KVell needed to scan the entire database for memory

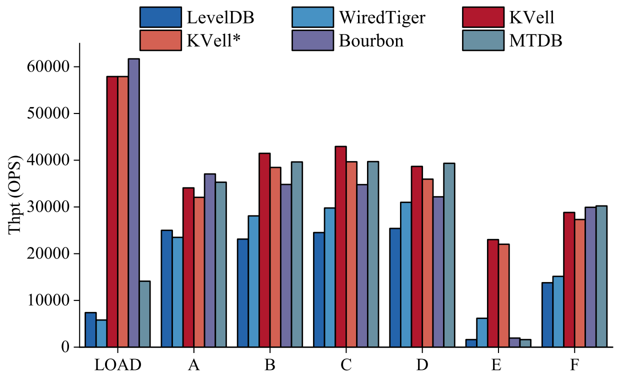


Fig. 6 YCSB(1KB) performance

index reconstruction each time it opened the database. We denote KVell, including the time for index reconstruction, as KVell*. To make a fair comparison, all the databases use one foreground thread and one background thread, and the Linux page cache is disabled.

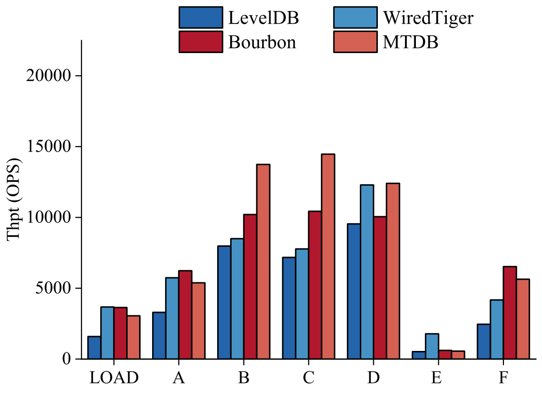
4.2 Experiments under YCSB workloads

The Yahoo! Cloud Serving Benchmark (YCSB) [17] is commonly employed to evaluate the performance of key-value storage systems under real-world workloads. It provides a framework and a standard set of six workloads for assessing the performance of key-value stores. The detailed information about the workloads is presented in Table 1. We utilized YCSB based on the Zipf distribution with a Zipfian constant of 0.99 (the default in YCSB) to load databases of two different value sizes, 1KB and 16KB, each with a total size of 10 GB. There are 5 M operations in the each workload with 1KB value sizes, while there are 312K operations in the each workload with 16KB value sizes.

4.2.1 Experiments under YCSB workloads

Figure 6 presents the performance evaluation of three systems using YCSB with a key-value size of 1KB. Overall, MTDB outperforms LevelDB by 1.52−2.19 times in throughput and achieves 1.27−2.43 times the throughput of WiredTiger, except for the E workload. In most cases, MTDB's performance is close to or better than KVell and Bourbon, while reducing memory overhead (Sect. 4.4). During the Load phase, MTDB exhibits a 1.7× improvement over LevelDB, primarily due to its design of partition-based Level 0, which accommodates more key-value pairs, reduces the frequency of Compaction triggers, and enhances the efficiency of each Compaction. KVell, utilizing a global memory index structure and avoiding Compaction, and Bourbon, employing a key-value separation structure to avoid rewriting values during LSM-tree Compaction, both achieve notable write performance. WiredTiger, which uses a disk-based B+tree, performs the worst under the Load workload. Under Workload A, MTDB's performance is second only to Bourbon. For update

Fig. 7 YCSB(16KB) performance



workloads, MTDB supports in-place updates. KVell achieves in-place updates within storage device pages but requires block erasure and rewriting, reducing its performance. Bourbon performs best under Workload A, mainly due to its key-value separation structure's advantages in update-heavy workloads. In Workload B (95% read) and C (100% read), MTDB's design of temporarily storing hot key-value pairs in a hot tree index results in read-intensive workload performance close to that of the globally memory-indexed KVell. Bourbon's learned index reduces the read and lookup overhead of Index Blocks and Data Blocks but cannot avoid the cost of reading Filter Blocks and retrieving values from the Vlog. Bourbon's read throughput is better than LevelDB and WiredTiger but lower than the memory-indexed MTDB and KVell.WiredTiger's disk-based B+tree performs better than LevelDB under read-intensive workloads (Workload B, C, D). In the Latest distribution of Workload D, MTDB's B+tree caches recently written new data, and the partition design increases Level 0 capacity, resulting in superior performance compared to other systems. In Workload E, KVell excels in range lookups as it only needs to traverse the memory B+tree index. WiredTiger's disk-based B+tree has advantages in range lookups over other LSM-tree-based systems. MTDB, Bourbon, and LevelDB perform similarly; MTDB's B+tree indexes only the hot tree data, requiring an iterator on the LSM-tree for range lookups. Additionally, MTDB does not perform Compaction in the hot tree, leading to a scattered distribution of key-value pairs. During range scans, MTDB requires more data blocks than LevelDB, but its memory index reduces the overhead of reading index blocks. In Workload F, KVell achieves in-place updates within storage device pages but still requires I/O operations. In Read-Modify-Write, some key-value pairs in MTDB's B+tree create an index, and in-place updates reduce the amount of data flushed, resulting in better performance than other systems. After accounting for the index rebuild time, the performance of KVell* shows a significant decrease. Except for Workload E, MTDB outperforms KVell*.

Figure 7 illustrates the experimental results of evaluating LevelDB, WiredTiger, Bourbon, and MTDB using YCSB with a key-value size of 16KB, observing

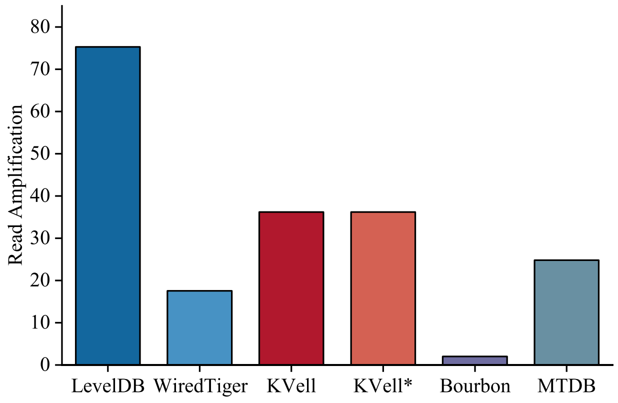


Fig. 8 Read amplification analysis under YCSB-C

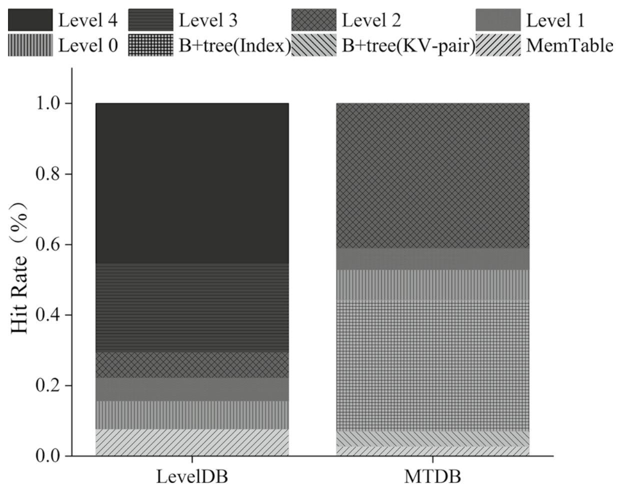


Fig. 9 Access analysis of each component for LevelDB and MTDB

the performance of each system as the key-value pair size increases. KVell cannot store key-value pairs larger than the specified page size, so KVell and KVell* are not included in the experimental results. With the same database size, the reduced number of key-value pairs results in fewer node splits and merges in WiredTiger's B+tree, making its write performance superior to the other LSM-tree-based systems. With the same database size and index memory overhead, MTDB's index structure can more effectively cover the data in the SSD. This reduces the number of data transfers from the hot tree to the LSM-tree when the memory space is insufficient. MTDB's relative performance in read-intensive workloads (Workload B, C, D) is

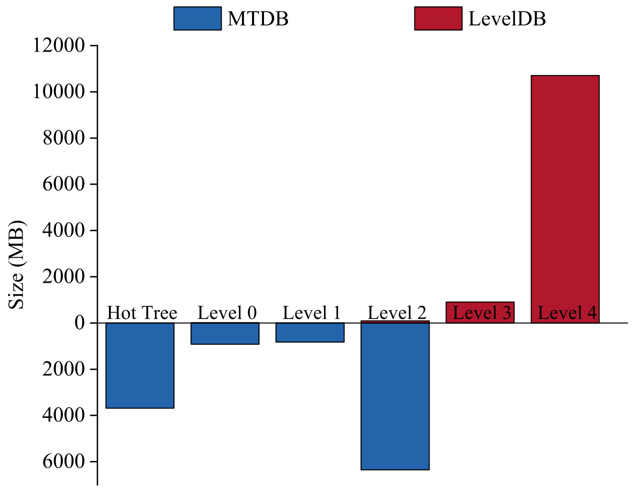


Fig. 10 Data distribution of each component in LevelDB and MTDB

further improved and outperforms the other compared systems. In workloads that include write operations (Workload A, F), the key-value separation system Bourbon and the B+tree-based storage system WiredTiger have advantages when handling 16KB key-value pairs. In contrast, the Compaction rewrite overhead is more pronounced in the LSM-tree-based MTDB and LevelDB. However, MTDB, with its Level 0 partition design, reduces write amplification compared to LevelDB. Consequently, MTDB still outperforms LevelDB in Workload A and F.

4.2.2 Read amplification under YCSB-C

Figure 8 illustrates the read amplification during Workload C for various systems. MTDB exhibits lower read amplification compared to other systems by reducing the number of accesses to Level 0 files through partitioning and creating indexes for the hot tree. Additionally, compared to LevelDB, MTDB has fewer LSM-tree levels for the same database size, reducing the number of SSTables to search. WiredTiger's B+tree has an advantage in read amplification, outperforming the LSM-tree-based LevelDB and MTDB. KVell incurs higher read amplification due to reading a significant amount of data during the reconstruction of the database index and subsequently reading data during the read load phase. Overall, the read amplification of KVell is higher than that of MTDB. Bourbon uses a key-value separation structure, resulting in a smaller overall size of the LSM-tree. This reduces lookup overhead within the LSM-tree. Additionally, Bourbon employs learned indexes to predict the location of data, reducing the number of accesses during lookup operations.

4.2.3 Access analysis of each component

In order to better analyze the read performance in the LSM-tree-based key-value stores, LevelDB and MTDB, we conducted a statistical analysis of the hit probabilities for various components in both systems, as shown in Fig. 9. We defined eight

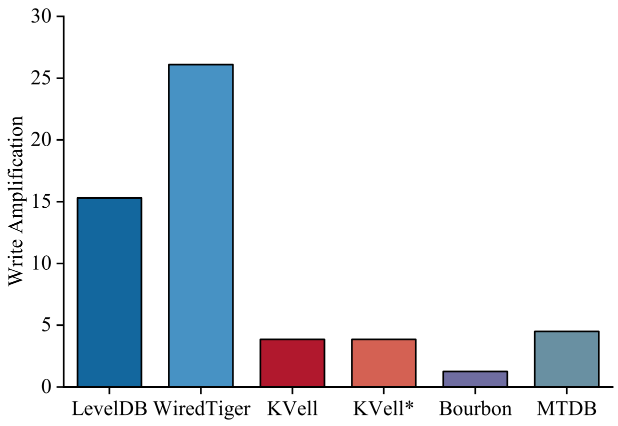


Fig. 11 Write amplification analysis

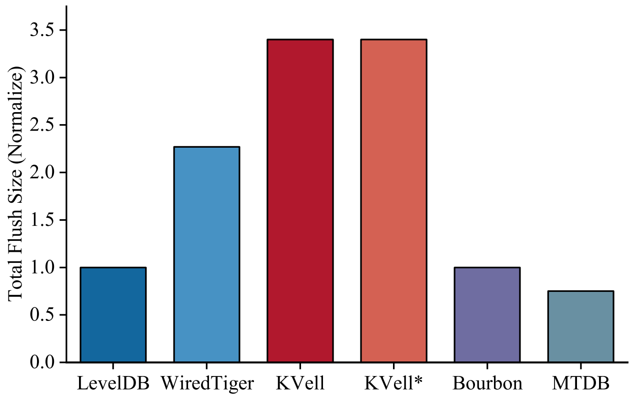


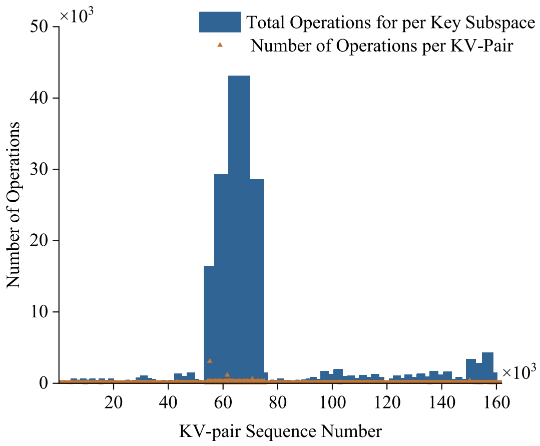
Fig. 12 Total size of flushed data

categories of components, including various memory structures and different levels of the LSM-tree. The read operation flows through these components from memory to SSD, ultimately reaching the highest level of the LSM-tree. The experiment did not include cache in the statistical analysis. The color intensity in the components represents the system overhead when a read operation hits that component, with darker colors indicating higher overhead. In MTDB, the system overhead for read operations is relatively low, and the overall color in the figure is lighter compared to LevelDB. Read operations in LevelDB concentrate on Level 3 and Level 4, while in MTDB, read operations are focused on the B+tree index and Level 2.

4.2.4 Data distribution of each component

After executing the WorkLoad, we compared the data distribution in various components of the LSM-tree-based key-value stores, LevelDB and MTDB, as

Fig. 13 The number of operations on KV Pairs and the distribution of operations in sub-spaces under RBM workloads



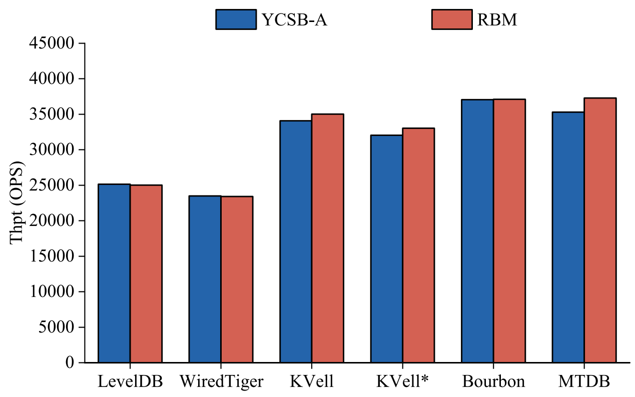


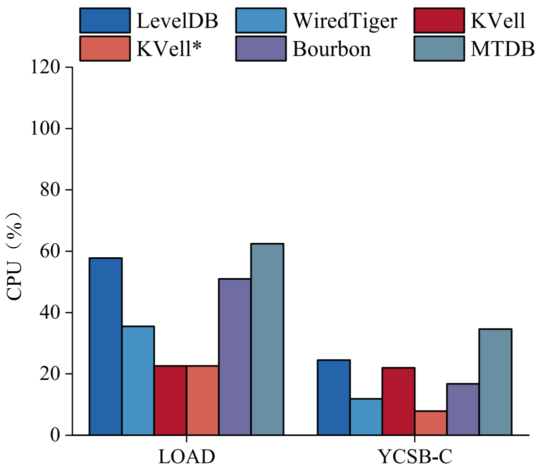
Fig. 14 Performance comparison under RBM workloads

illustrated in Fig. 10. In MTDB, the data is primarily concentrated in the hot tree and Level 2, with approximately 30% of the data residing in the Hot tree, occupying about 145MB of B+tree memory space dedicated to indexing key-value pairs in the hot tree. On the other hand, in LevelDB, 90% of the data is located in Level 4, which is a significant factor causing the concentration of data hits in Level 4 and consequently leading to a decline in read performance.

4.2.5 Write amplification

Figure 11 depicts the write amplification statistics during the Load phase for various systems. In the Load phase, MTDB retains some data in Level 0 partitions, reducing the number of Compaction triggers. Additionally, MTDB performs Level 0-Level 1 Compaction based on key ranges, selecting only SSTables in Level 1 that match the key range. This strategy reduces the involvement of Level 1 SSTables in

Fig. 15 CPU usage under LOAD and YCSB-C



Level 0-Level 1 Compaction, preventing rewriting of Level 1 SSTables. The write amplification in MTDB is reduced by 70% compared to LevelDB. KVell, utilizing a memory index design, avoids write amplification by not executing Compaction but incurs higher space amplification and memory usage as a trade-off. Bourbon uses a key-value separation structure, has the smallest write amplification during the Load phase.

4.2.6 Total amount of persistent data

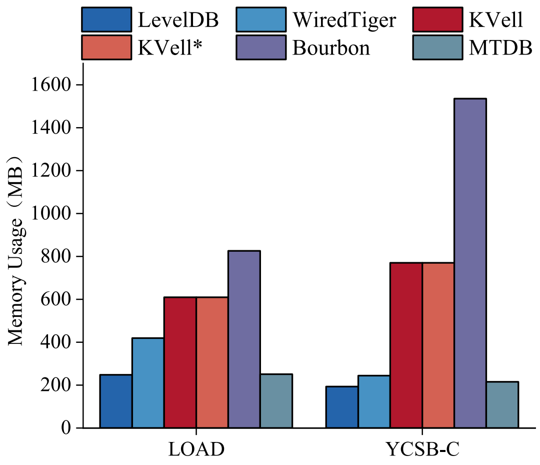
Figure 12 illustrates the number of key-value pairs flushed during WorkLoad F. WorkLoad F includes read operations and read-modify-write operations, allowing the B+tree to retain some hot key-value pairs to absorb update operations. This strategy effectively reduces the amount of writes to SSD under update loads. MTDB shows a 24% reduction in the number of flushed key-value pairs compared to LevelDB in WorkLoad F.

KVell necessitates a substantially higher volume of data to be flushed during update operations. This is because KVell uses an in-place update strategy. When the size of the key-value pair (1KB) is smaller than the page size of the SSD (4KB), KVell's in-place updates can cause frequent read-write-modify operations within the SSD, ultimately resulting in KVell's data write volume far exceeding that of other systems.

4.3 Experiments under RBM workloads

We conducted an analysis of spatial locality characteristics within YCSB workloads, as discussed in 2.2. Some subspaces in YCSB exhibit relatively high range hotness. RBM [4] emulates real-world workloads of key-value stores, characterized by their distribution of hotness and temporal patterns. Thus, RBM workloads can generate more pronounced spatial locality features, and we analyzed one million data points to understand these characteristics. The results, shown in Fig. 13, reveal that certain

Fig. 16 Memory usage under LOAD and YCSB-C



subspaces have significantly higher hotness compared to others, and there are no keys with extremely high operation frequencies. Using this workload and simulating the operation ratio of YCSB Workload A, the experimental results in Fig. 14 demonstrate a 10% performance improvement for MTDB compared to using the standard YCSB workload. This improvement is attributed to the more accurate identification of hot subspaces by the hotness strategy, leveraging the spatial locality features.

4.4 CPU and memory usage

Figure 15 shows the CPU usage of various systems during the write-intensive LOAD phase and the read-intensive YCSB-C phase. During the LOAD phase, the CPU overhead of LevelDB and MTDB, which use LSM-tree, is relatively high, primarily due to the frequent triggering of compaction operations. Although MTDB mitigates the triggering of Level 0 to Level 1 compactions, the in-memory B+tree and Transfer Compaction add extra CPU overhead, which is 8% higher than LevelDB. KVell, with its in-memory B+tree index and disk append-write design, incurs lower CPU overhead. Bourbon, using a key-value separation structure, has lower CPU overhead when writing data, with the main CPU consumption occurring during the learning index model building after the LOAD phase.

To avoid interference from Size Compaction in LevelDB, Bourbon, and MTDB during the CPU overhead testing of read workloads, the YCSB-C workload was executed when the database was in a stable state and no Size Compaction would be triggered. In MTDB, background threads perform Transfer Compaction and hot tree garbage collection to improve the access efficiency of hot key-value pairs, resulting in CPU overhead 1.4 times that of LevelDB but with minimal impact on foreground threads. Additionally, MTDB disables Seek Compaction, while in LevelDB, Seek Compaction triggered by read operations causes CPU overhead. KVell has high CPU overhead during the index rebuilding phase but low CPU overhead during read operations. Bourbon avoids data block retrieval, reducing CPU usage.

Figure 16 shows the memory usage of various systems during the write-intensive LOAD phase and the read-intensive YCSB-C phase. The memory usage of LevelDB and MTDB, both using LSM-tree, is similar and lower than the other compared systems. KVell maintains a global B+tree index in memory, while Bourbon keeps the model in memory, resulting in higher memory usage compared to the other systems. MTDB's memory overhead is reduced by 58.85−72.08% compared to KVell and by 69.61−86.00% compared to Bourbon.

5 Discussion

(1)Worst-case Scenario with Rapid Changes in Key Range Hotness. During drastic changes in key range hotness, all hot tree data needs to be GC into the LSM-tree. The experiment simulated this situation. To avoid the impact of compaction operations in the LSM-tree, the experiment was conducted after all Size Compactions were completed and Level 0 was empty. During GC, data is sequentially written to

Level 0 by traversing B+ tree index nodes. As GC data increases, Level 0-Level 1 compactions slow down throughput, taking about 105 s for 3 million 1KB key-value pairs. In extreme hotness changes, writing to Level 0 may not be optimal. Future work could assess key range hotness change frequency and choose a lower level for writing. The experiment also tested the overhead when large amounts of LSM-tree data are transferred to the hot tree. Since only index addresses are fetched and written to the B+ tree without rewriting SSTables, throughput is faster than hot tree garbage collection, taking about 24 s for Transfer Compaction on 3 million 1KB key-value pairs.

Under workloads with non-extreme key range hotness changes, both Transfer Compaction and GC in MTDB are executed at the key subspace granularity. Data within each key subspace ranged from 10MB to 40MB. Background threads perform Transfer Compaction and GC in short periods, locking only specific partitions and minimally impacting foreground threads.

(2)Handling Key Character Composition Changes. When handling regular workloads in MTDB, the number of Guards stabilizes once it reaches the upper limit. However, if the character composition of keys changes in the workload, the current Guard strategy (no deletion or reallocation) may result in skewed data distribution within key subspaces, potentially causing large data volumes during Transfer Compaction and GC. Additionally, in sequential load workloads, this could lead to the addition of ineffective Guards. Future work should consider operations for deleting and reallocating Guards to adapt to these special workloads.

3)Uniform Workloads with Insignificant Hotness Features. In uniform workloads where hotness features are not significant, the benefits gained by MTDB cannot offset the overhead of separating and transferring hot and cold key-value pairs in the B+tree. In such workloads, the hot tree index in MTDB can only handle 30% of read requests. Most lookups hit the LSM-tree, increasing the lookup path compared to LevelDB as an additional lookup in the B+tree is required first. Future work should consider a dynamic multi-tree usage strategy, where data skips writing to the B+tree and directly writes to L0 when hotness features are not significant, and the execution of Transfer Compaction is turned off.

6 Related work

Read optimization Tidal-Tree-Mem [12] moves frequently accessed files from lower levels to higher positions to reduce the overall file lookup time. Bourbon [27] accelerated searches on SStables by constructing learned indexes from static data stored within the SStables. TridentKV [9] employs a space-efficient partition strategy to address the Read-After-Delete problem and incorporates an optimized learned index block structure for faster file reading. SineKV [28] utilizes the characteristics of SSDs [29, 30] to decouple secondary index management from primary index management to accelerate query performance. LTG-LSM [31] maintains a hotness prediction model at each level. However, these approaches do not address the issue of needing to traverse all files in Level 0 during read operations due to the relaxed order in Level 0. MTDB strengthens the order in Level 0, reducing the lookup cost and expanding the number of SSTables Level 0 can accommodate.

Index. KVell [11] achieves fast read and write speeds by managing unordered KV data on disk using a complete B-tree index in DRAM. However, it comes with high memory costs and slow recovery. REMIX [32] focuses on improving range query efficiency by proposing a space-efficient KV index structure for a global sorted view of KV data-related files. SLM-DB [10] uses NVM storage for B+tree indexing of KV data on an SSD, avoiding KVell's downsides but requiring special hardware support and incurring additional crash consistency maintenance overhead. UniKV [33] combines hash indexing with LSM-tree, using hash indexing for new data on disk and LSM-tree for old data. LSM-trie [34] uses LSM-tree-based prefix hash indexing to manage keys, introducing a partitioned hierarchical approach to reduce write amplification. ForestDB [35] proposes a new hybrid index structure called HB+trie to efficiently manage variable-length string keys and reduce the number of disk accesses during index operations, achieving high throughput for both read and write operations. However, these solutions do not achieve in-place updates in memory under update workloads. MTDB utilizes B+tree for both memory component key-value separation and unified indexing structure, reducing global index memory overhead and accommodating update workloads with in-place updates.

Hot and Cold Separation TRIAD [36] suggests separating hot and cold keys in memory components to address write amplification, but its smaller Memtable size limits the effectiveness of hotness detection. Siberia [37] records access timestamps and predicts hot records, saving hot data in memory and moving cold data to disk. MTDB, using B+tree, achieves hot and cold key separation in memory components. However, Siberia and TRIAD do not consider the scenario where cold keys in disk become hot after some time, lacking effective hotness strategies for already written disk key-value pairs. MTDB partitions based on key subspaces and transfers hotness, creating indexes for hot key subspaces, providing access priority for hot key-value pairs that were already written to disk.

Partitioning LWC-tree [38] uses vertical partitioning to narrow the key range of dense SSTable groups, reducing write amplification. Pebblesdb [39], inspired by skip lists, integrates the sentinel concept into LSM-tree management, using a segmented log structure-merge tree to build a key-value store. PebblesDB relaxes the constraint of non-overlapping key ranges outside Level 0 and introduces protection to prevent rewriting data at the same level, reducing Compaction costs. MTDB implements partitioning design only in Level 0, maintaining non-overlapping key ranges in other levels of LSM-tree. This contrasts with PebblesDB's coarse-grained data structure, which results in serious space amplification. MTDB adapts its memory B+tree structure to fit the Level 0 partition design, mitigating heavy Level 0-Level 1 Compaction costs, achieving Level 0 partitioning without affecting write performance. To balance overall performance, MTDB maintains the non-overlapping key range setting in the remaining levels, avoiding additional read operation costs if partitioning were applied.

Cache AC-Key [40] sets three different granularities of cache: key-value cache, key-pointer cache, and block cache. It can dynamically adjust the sizes of these three cache components based on the cost and benefit recorded in the cache to adapt to workload changes, improving the performance of point read and range lookup operations. However, it cannot avoid the issue of Block Cache invalidation under mixed read-write workloads. LSbM [41] adds an inter-layer merge buffer on the disk, adaptively maintaining a collection of frequently accessed data to minimize cache invalidation caused by merge operations, ensuring cache hit rates during queries. Similarly, Leaper [42] addresses the problem of block cache invalidation and the resulting drop in cache hit rates caused by inter-layer merge operations by predicting hot data and prefetching it into the cache when it affects background operations, thereby reducing cache invalidation and improving read performance. However, the Block Cache does not reduce the lookup path for read requests. Lookup requests in the LSM-tree still need to search layer by layer from the higher levels to the lower levels, and steps like obtaining file handles and retrieving index blocks cannot be avoided. MTDB uses a B+tree index to shorten the lookup path, allowing direct access to the data block addresses of hot key-value pairs. Under skewed workloads, the access frequency of MTDB's hot tree data blocks is high, and since no Compaction operations occur in the hot tree, the probability of cache invalidation in the cache is reduced.

Filter ElasticBF [13] automatically adjusts the Bloom filter false positive rate based on the hotness and access frequency of keys, considering storage constraints. Monkey [43] optimizes the overall system performance of LSM-tree-based systems by worst-case analysis of lookup operations. Using filters can reduce the number of data block accesses but cannot reduce the overhead of opening SSTables, reading, and searching index blocks. Additionally, it requires reading extra filter blocks and using additional memory space to cache the filter. MTDB uses a B+tree to index hot key-value pairs, allowing direct retrieval of the target key-value pair's data block address, thereby avoiding the overhead of reading and searching index blocks.

7 Conclusion

We introduce MTDB, a novel database system that effectively combines B+tree, single-level hot tree and LSM-tree. MTDB utilizes hotness detection in B+tree to distinguish between hot and cold key-value pairs. It indexes a Single-Level Hot tree in B+tree and efficiently manages new and early-written key-value pairs in LSM-tree based on hotness characteristics. This approach improves the read efficiency of lower levels in LSM-tree under workloads with evident range hotness features. Experimental results demonstrate that, compared to prior research, MTDB reduces memory consumption by 58.85–86% and index reconstruction overhead, leading to enhanced read and write performance.

Author Contributions XL, YP, WF, HZ contributed to conceptualization, methodology, and investigation. XL done formal analysis, software, validation, and writing—original draft. YP and ML done supervision and resources. YP, WF, HZ, ML helped in writing—review and editing.

Funding This work was supported by Nature Science Foundation of Fujian Province under Grant No. 2021J01319, Fundamental Research Founds for the Central Universities of Huaqiao University under Grant No. ZQN-910, National Natural Science Foundation of China under Grant No. 61872086, Natural

Science Foundation of Fujian Province of China under Grant No. 2022J06020, and the Young Top Talent of Young Eagle Program of Fujian Province, China under Grant No. F21E0011202B01.

Data availability No datasets were generated or analysed during the current study

Declarations

Conflict of interest The authors declare no conflict of interest.

References

1. O'Neil Patrick, Cheng Edward, Gawlick Dieter, O'Neil Elizabeth (1996) The log-structured merge-tree (lsm-tree). Acta Inf 33:351–385
2. Chang Fay, Dean Jeffrey, Ghemawat Sanjay, Hsieh Wilson C, Wallach Deborah A, Burrows Mike, Chandra Tushar, Fikes Andrew, Gruber Robert E (2008) Bigtable: A distributed storage system for structured data. ACM Trans. Comput. Syst. (TOCS) 26(2):1–26
3. LevelDB (2021) https://github.com/google/leveldb
4. RocksDB (2022) http://rocksdb.org/
5. Cassandra (2020) http://cassandra.apache.org/
6. HBase (2020) http://hbase.apache.org/
7. Lu Lanyue, Pillai Thanumalayan Sankaranarayana, Gopalakrishnan Hariharan, Arpaci-Dusseau Andrea C, Arpaci-Dusseau Remzi H (2017) Wisckey: separating keys from values in ssd-conscious storage. ACM Trans Storage 13(1):1–28
8. Agrawal Nitin, Prabhakaran Vijayan, Wobber Ted, Davis John D, Manasse Mark, Panigrahy Rina (2008) Design tradeoffs for ssd performance. In: 2008 USENIX Annual Technical Conference
9. Kai Lu, Zhao Nannan, Wan Jiguang, Fei Changhong, Zhao Wei, Deng Tongliang (2021) Tridentkv: a read-optimized lsm-tree based kv store via adaptive indexing and space-efficient partitioning. IEEE Trans Parallel Distrib Syst 33(8):1953–1966
10. Kaiyrakhmet Olzhas, Lee Songyi, Nam Beomseok, Noh Sam H, Choi Young-ri (2019) Slm-db:single-levelkey-value store with persistent memory. In: 17th USENIX Conference on File and Storage Technologies, pp 191–205
11. Lepers Baptiste, Balmau Oana, Gupta Karan, Zwaenepoel Willy (2019) Kvell: the design and implementation of a fast persistent key-value store. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles, pp 447–461
12. Ma Chenlin, Yang Hao, Shangyu Wu, Wang Yi, Mao Rui (2022) Tidal-tree-mem: Toward read-intensive key-value stores with tidal structure based on lsm-tree. IEEE Trans Comput Aided Des Integr Circuits Syst 42(2):423–436
13. Li Yongkun, Tian Chengjin, Guo Fan, Li Cheng, Xu Yinlong (2019) Elasticbf: Elastic bloom filter with hotness awareness for boosting read performance in large key-value stores. In: 2019 USENIX Annual Technical Conference, pp 739–752
14. Atikoglu Berk, Xu Yuehai, Frachtenberg Eitan, Jiang Song, Paleczny Mike (2012) Workload analysis of a large-scale key-value store. In: Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, pp 53–64
15. Cao Zhichao, Dong Siying, Vemuri Sagar, Du David HC (2020) Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In: 18th USENIX Conference on File and Storage Technologies, pp 209–223
16. Yang Juncheng, Yue Yao, Rashmi KV (2021) A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. ACM Trans Storage (TOS) 17(3):1–35
17. Cooper Brian F, Silberstein Adam, Tam Erwin, Ramakrishnan Raghu, Sears Russell (2010) Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM symposium on Cloud computing, pp 143–154
18. Wikipedia. Bloom filter. https://en.wikipedia.org/wiki/Bloom_filter
19. Chen Jiqiang, Chen Liang, Wang Sheng, Zhu Guoyun, Sun Yuanyuan, Liu Huan, Li Feifei (2020) Hotring: A hotspot-aware in-memory key-value store. In: 18th USENIX Conference on File and Storage Technologies (FAST 20), pp 239–252
20. Appsflyer (2021) https://appsflyer.com

21. Flurry analytics (2021) https://flurry.com
22. Google firebase (2020) https://firebase.google.com
23. Pugh William (1990) Skip lists: a probabilistic alternative to balanced trees. Commun ACM 33(6):668–676
24. Pugh William (1998) A skip list cookbook. Technical report
25. Wei Qian, Chen Zehao, Chen Xiaowei, Zhang Yuhao, Cai Xiaojun, Jia Zhiping, Shen Zhaoyan, Wang Y, Shao Zili, Li Bingzhe (2023) A semantic-integrated lsm-tree based key-value storage engine for blockchain systems. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems
26. WiredTiger storage engine (2023) https://docs.mongodb.com/manual/core/wiredtiger/
27. Dai Yifan, Xu Yien, Ganesan Aishwarya, Alagappan Ramnatthan, Kroth Brian, Arpaci-Dusseau Andrea, Arpaci-Dusseau Remzi (2020) From wisckey to bourbon: A learned index for log-structured merge trees. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pp 155–171
28. Li Fei, Lu Youyou, Yang Zhe, Shu Jiwu (2020) Sinekv: Decoupled secondary indexing for lsm-based key-value stores. In: 2020 IEEE 40th International Conference on Distributed Computing Systems, pp 1112–1122
29. Luo Yuhan, Lin Mingwei, Pan Yubiao, Zeshui Xu (2022) Dual locality-based flash translation layer for nand flash-based consumer electronics. IEEE Trans Consum Electron 68(3):281–290
30. Zhang Jianpeng, Lin Mingwei, Pan Yubiao, Zeshui Xu (2023) Crftl: Cache reallocation-based page-level flash translation layer for smartphones. IEEE Trans Consum Electron 69(3):671–679
31. Yu JiaPing, Chen HuaHui, Qian JiangBo, Dong YiHong (2020) Ltg-lsm: The optimal structure in lsm-tree combined with reading hotness. In: 2020 IEEE 26th International Conference on Parallel and Distributed Systems, pp 1–8
32. Zhong Wenshao, Chen Chen, Wu Xingbo, Jiang Song (2021) Remix: Efficient range query for lsm-trees. In: 19th USENIX Conference on File and Storage Technologies, pp 51–64
33. Zhang Qiang, Li Yongkun, Lee Patrick PC, Xu Yinlong, Cui Qiu, Tang Liu (2020) Unikv: Toward high-performance and scalable kv storage in mixed workloads via unified indexing. In: 2020 IEEE 36th International Conference on Data Engineering, pp 313–324
34. Wu Xingbo, Xu Yuehai, Shao Zili, Jiang Song (2015) Lsm-trie: An lsm-tree-basedultra-largekey-value store for small data items. In: 2015 USENIX Annual Technical Conference, pp 71–82
35. Ahn Jung-Sang, Seo Chiyoung, Mayuram Ravi, Yaseen Rahim, Kim Jin-Soo, Maeng Seungryoul (2015) Forestdb: A fast key-value storage system for variable-length string keys. IEEE Trans Comput 65(3):902–915
36. Balmau Oana, Didona Diego, Guerraoui Rachid, Zwaenepoel Willy, Yuan Huapeng, Arora Aashray, Gupta Karan, Konka Pavan (2017) Triad: Creating synergies between memory, disk and log in log structured key-value stores. In: 2017 USENIX Annual Technical Conference, pp 363–375
37. Levandoski Justin J, Larson Per-Åke, Stoica Radu (2013) Identifying hot and cold data in main-memory databases. In: 2013 IEEE 29th International Conference on Data Engineering, pp 26–37
38. Yao Ting, Wan Jiguang, Huang Ping, He Xubin, Fei Wu, Xie Changsheng (2017) Building efficient key-value stores via a lightweight compaction tree. ACM Trans Storage 13(4):1–28
39. Raju Pandian, Kadekodi Rohan, Chidambaram Vijay, Abraham Ittai (2017) Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In: Proceedings of the 26th Symposium on Operating Systems Principles, pp 497–514
40. Wu Fenggang, Yang Ming-Hong, Zhang Baoquan, Du David HC (2020) Ac-key: Adaptive caching for lsm-based key-value stores. In: 2020 USENIX Annual Technical Conference (USENIX ATC 20), pp 603–615
41. Teng Dejun, Guo Lei, Lee Rubao, Chen Feng, Ma Siyuan, Zhang Yanfeng, Zhang Xiaodong (2017) Lsbm-tree: Re-enabling buffer caching in data management for mixed reads and writes. In: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), pp 68–79. IEEE
42. Yang Lei, Hong Wu, Zhang Tieying, Cheng Xuntao, Li Feifei, Zou Lei, Wang Yujie, Chen Rongyao, Wang Jianying, Huang Gui (2020) Leaper: a learned prefetcher for cache invalidation in lsm-tree based storage engines. Proc VLDB Endowment 13(12):1976–1989
43. Dayan Niv, Athanassoulis Manos, Idreos Stratos (2017) Monkey: Optimal navigable key-value store. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp 79–94

Authors and Affiliations

Xinwei Lin[1,2] · Yubiao Pan[1,2] · Wenjuan Feng[1] · Huizhen Zhang[1] · Mingwei Lin[3]

✉ Yubiao Pan
panyubiao@hqu.edu.cn

✉ Mingwei Lin
linmwcs@163.com

Xinwei Lin
linxinwei@stu.hqu.edu.cn

Wenjuan Feng
fengwenjuan@hqu.edu.cn

Huizhen Zhang
zhanghz@hqu.edu.cn

[1] The School of Computer Science and Technology, Huaqiao University, Xiamen 361021, Fujian, China

[2] Data Security Department, Xiamen Key Laboratory of Data Security and Blockchain Technology, Xiamen 361021, Fujian, China

[3] The College of Mathematics and Informatics, Fujian Normal University, Fuzhou 350000, Fujian, China