

HAKV: A Hotness-Aware Zone Management Approach to Optimizing Performance of LSM-tree-based Key-Value Stores

HUI SUN, Computer Science and Technology, Anhui University, Hefei, China

QIANLI YUE, Anhui University, Hefei, China

GUANZHONG CHEN, Anhui University, Hefei, China

YI ZOU, Columbus State University, Columbus, United States

YINLIANG YUE, Zhongguancun Laboratory, Beijing, China

XIAO QIN, Computer Science, Auburn University, Auburn, United States

Log-Structured Merge tree-based key-value (KV) stores, like LevelDB and RocksDB, are extensively applied in large-scale data storage systems. This design excels in write-intensive environments by converting random writes into sequential append operations. Despite its advantages, KV stores struggle with real-world workloads where most updates in KV pairs are infrequent. The compaction process and hierarchical data organization result in high write and read amplification. To mitigate these issues, we propose *HAKV* – a hotness-aware zone management approach to optimizing performance of KV stores. *HAKV* first separates hot KV pairs from cold KV pairs, storing hot KV pairs in dedicated zones within persistent memory (PM), enabling centralized and lightweight compaction. Second, we propose a storage zone structure in PM to achieve space optimization for cold KV pairs. Third, to bolster cache hit ratio in PM, we provide a hierarchical data framework for hot KV pairs – and a recycling strategy for invalid hot KV pairs in a zone to enhance the space utilization of PM for hot KV pairs. Finally, we design a dynamic window-based adaptive adjustment mechanism for zone pool in PM to optimize the space utilization. Thus, *HAKV* significantly reduces write amplification while boosting overall read and write performance. The experimental results demonstrate that *HAKV* achieves write amplification reduction by up to 92.3%, 79.2%, 90.2%, 41.1%, 80.6%, and 62.4% compared with LevelDB, RocksDB, NoveLSM, LightKV, Wiskey, and UniKV, respectively, with average reduction rates of 89.6%, 74.4%, 84.9%, 32.3%, 63.7%, and 42.5%. Furthermore, *HAKV* boosts random write performance by up to 54.2×, 51.5×, 44.2×, 4.3×, 3.1×, and 4.3×, respectively—and the average improvement reaches 25.8×, 20.9×, 23.9×, 2.7×, 2.5×, and 3.4×.

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

Additional Key Words and Phrases: LSM-tree, key-value store, persistent memory, zone pool management, compaction optimization

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, China; e-mail: sunhuiworking@gmail.com; Qianli Yue, Anhui University, Hefei, Anhui, China; e-mail: qianl@ahu.edu.cn; Guanzhong Chen, Anhui University, Hefei, Anhui, China; e-mail: guanzhong@ahu.edu.cn; Yi Zou, Columbus State University, Columbus, Georgia, United States; e-mail: zhou_yi@columbusstate.edu; Yinliang Yue, Zhongguancun Laboratory, Beijing, Beijing, China; e-mail: yylhust@qq.com; Xiao Qin, Computer Science, Auburn University, Auburn, Alabama, United States; e-mail: xqin@auburn.edu.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2025 Copyright held by the owner/author(s).

ACM 1544-3973/2025/09-ART108

<https://doi.org/10.1145/3747848>

ACM Reference Format:

Hui Sun, Qianli Yue, Guanzhong Chen, Yi Zou, Yinliang Yue, and Xiao Qin. 2025. HAKV: A Hotness-Aware Zone Management Approach to Optimizing Performance of LSM-tree-based Key-Value Stores. *ACM Trans. Arch. Code Optim.* 22, 3, Article 108 (September 2025), 26 pages. <https://doi.org/10.1145/3747848>

1 Introduction

International Data Corporation (IDC) [14] forecasts that the global data volume will approach 250 **Zettabytes (ZB)** by 2027, with unstructured data accounting for 86.8% of the total. **Key-value (KV)** stores, as a vital solution for managing unstructured data, play a critical role in modern data centers, underpinning applications like web search, e-commerce platforms, and social networks. Log-structured merge tree (LSM-tree) is widely used in various KV stores such as LevelDB [20], RocksDB [10], Cassandra [18], HBase [12]. LSM-trees excel in write-intensive workloads by transforming random writes into sequential appends in a log format. In practice, however, repeatedly updated KV pairs are often a small minority: most pairs are written only once.

LSM-tree organizes KV pairs using a multi-level structure. Incoming KV pairs are initially inserted into an in-memory table called a MemTable. Once the MemTable reaches a threshold size, it is converted into an immutable MemTable being flushed to disk as a **sorted string table (SSTable)**. To achieve high read performance, the KV pairs at the same level (except for level L0) are ordered. SSTables in adjacent levels with overlapping key ranges are merged. This compaction process involves reading overlapping SSTables into memory, merging them, and writing the resulting SSTables to the next level. Compaction reclaims storage space by eliminating invalid KV pairs while preserving the sorted order within each level, ultimately optimizing read performance. This compaction process, unfortunately, leads to high write amplification, particularly for cold KV pairs. The LSM-tree structure also causes serious read amplification, hindering the overall performance.

Prior research has spearheaded leading-edge techniques to mitigate write amplification in KV stores [3, 13], broadly categorized into decentralized compaction [31, 39, 40] and KV separation [23, 33, 43]. Decentralized compaction methods adjust the granularity or timing of compaction to minimize adverse impact on read and write performance. For example, Wang et al. [35] reduce the compaction granularity from SSTable to block, aiming to lower compaction costs. PebblesDB [29] relaxes strict SSTables ordering within the same level by using key-range partitioning—and allowing overlapping key ranges within partitions, thereby avoiding multiple rewrites of KV pairs. However, these strategies merely postpone the onset of write amplification rather than addressing its root cause. KV separation strategies mitigate write amplification caused by value merging. These techniques separate keys and values, typically leveraging value log file indexes instead of directly embedding values within SSTables. While being effective in some scenarios, this approach requires **garbage collection (GC)** mechanisms to reclaim invalid value log files. Moreover, the scheme such as FenceKV [32] struggles to handle the majority of infrequently updated KV pairs (cold) KV pairs, leading to write amplification in the value log space.

To improve the performance of KV stores, prior works [5, 7, 19, 24, 42, 44, 46] have employed **persistent memory (PM)** as a cache for user data. For instance, NoveLSM [16] stores immutable tables in PM, whereas LightKV [11] places upper-level file partition segments in PM. However, these approaches often neglect PM page alignment and fail to differentiate between hot and cold KV pairs within PM. Consequently, despite improvements in read/write performance and reductions in write amplification, PM is not optimally utilized.

To address these challenges, we propose *HAKV*—a hotness-aware zone management approach in PM to optimize the performance of KV stores. *HAKV* leverages PM with awareness of hot and

cold access patterns. Our experimental results confirm that most KV pairs are only written once, with updates concentrated on a small subset. HAKA first segregates KV pairs into hot and cold data based on access patterns. This separation prevents the compaction of KV pairs that are unlikely to be updated, thereby reducing write amplification. Second, we build a two-level index in DRAM for storage zones in PM and the SSD. All zones are of uniform size, and each zone includes multiple KV pairs. Cold zones that store cold KV pairs ensure sequential-write KV pairs to be placed in the zone, whereas hot zones contain hot KV pairs. Notably, a KV pair initially stored in a cold zone becomes “hot” upon its first update. Frequently-accessed KV pairs are processed in PM, while infrequently-accessed KV pairs are migrated to SSDs, improving performance and preserving the locality of cold KV pairs. Third, HAKV applies a circular queue mechanism to efficiently manage PM space, enabling dynamic allocation coupled with optimized resource utilization. Finally, we propose a dynamic window-based adaptive GC strategy to maximize PM space utilization. The contributions of this work are summarized as follows.

- By analyzing the update patterns of user keys, we implement in HAKV a preprocessing mechanism to separate hot KV pairs from cold KV pairs. Hot KV pairs are selectively stored in storage zones within PM, facilitating centralized and lightweight compaction. This design significantly mitigates high write amplification caused by repeated updates.
- HAKV features the storage zone structure to achieve spatial optimization for cold KV pairs. To improve access locality for cold KV pairs, we group cold KV pairs with similar write timing into cold zones and temporarily store them in a cold zone pool within PM. HAKV preserves spatial locality and enhances the read/write efficiency of KV pairs.
- We propose a hierarchical management and recycling strategy to enhance the PM storage zone management. This approach selectively merges and reclaims hot KV pairs in PM while applying an in-place rewrite strategy for cold KV pairs stored on SSDs. The strategy is slated to reduce write amplification and improve the performance of hot KV pairs.
- We put forth a dynamic window-based adaptive adjustment mechanism optimizing PM space utilization. To efficiently manage PM space, this mechanism dynamically adjusts the reclamation and allocation strategies for the cold storage pool according to front-end user load. HAKV minimizes resource waste and maintains high read/write performance by adaptively allocating storage pool space in line with real-time workloads.
- We implement HAKV by extending LevelDB 1.20—and we conduct performance evaluations using DB_Bench and YCSB-C. The experimental results demonstrate that HAKV reduces write amplification by up to 92.3%, 79.2%, 90.2%, 41.1%, 80.6%, and 62.4% compared with LevelDB, RocksDB, NoveLSM, LightKV, Wisckey, and UniKV, respectively, with an average reduction of 89.6%, 74.4%, 84.9%, 32.3%, 63.7%, and 42.5%. HAKV boosts random write performance by up to 54.2×, 51.5×, 44.2×, 4.3×, 3.1×, 4.3×, respectively. The average improvement is 25.8×, 20.9×, 23.9×, 2.7×, 2.5×, and 3.4×, respectively.

The rest of this article is organized as follows. Section 2 discusses the background and motivation. Section 3 describes the design and implementation issues of HAKV. Section 4 presents the performance evaluation results of HAKV. Section 5 articulates the related work, and we conclude this article in Section 6.

2 Background and Motivation

2.1 Background

In this section, we provide an overview of the LSM-tree, LevelDB, and PM.

2.1.1 LSM-Tree and LevelDB. The LSM-tree [27], as a persistent data structure, is suited for write-intensive workloads because it can convert random writes into sequential ones, thereby achieving

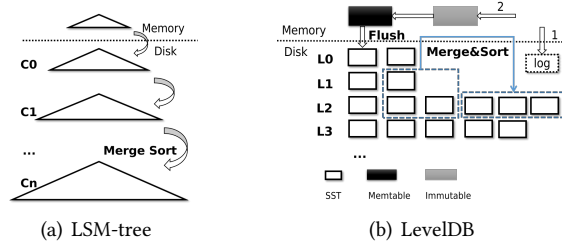


Fig. 1. The LSM-tree and LevelDB.

high write performance. The LSM-tree structure has been widely adopted in prominent storage engines, such as LevelDB [20], RocksDB [10], and Cassandra [18]. In Figure 1(a), the LSM-tree employs a hierarchical and orderly structure to manage KV pairs. The LSM-tree consists of multiple components $C_0 \sim C_n$. KV pairs are first written to C_0 in memory. When C_0 is full, C_0 is merged with C_1 in the disk. Then, the result is flushed into C_1 . Similarly, when C_1 is full, it will be merged with C_2 to C_2 . In LevelDB, a popular KV store utilizing the LSM-tree, KV pairs are first written into a write-ahead log file to ensure data persistence. Then, they are inserted into a sorted MemTable. Once the MemTable reaches a predefined size, it is converted into an immutable MemTable, which is then flushed to disk as a SSTable.

On disk, the LSM-tree organizes SSTables into multiple levels, with the capacity of adjacent levels increasing exponentially by an amplification factor (AF, default 10). To optimize read performance, SSTables at the same level cannot have overlapping key ranges (except level L0). As the number of SSTables grows, SSTables with overlapping key ranges in the upper level are read into memory to perform merge-sorting and then written to new SSTables in the next level. This process is called compaction, which leads to write amplification (i.e., WA). WA is defined as a ratio between the amount of data written into an underlying storage device and that requested by users. Large WA consumes I/O bandwidth and degrades performance. To mitigate WA, we propose separating hot and cold KV pairs based on the access patterns and placing updates accordingly.

Mostly, the hierarchical organization of the LSM-tree introduces three steps to search the target KV pairs. First, the search process begins within the MemTable, where the target key is initially inserted. Second, if the key is not found in the MemTable, the search proceeds to the shared memory data block cache. Finally, if the target key is not hit in the cache, it will be searched in the SSTables, starting at level L0 and sequentially progressing through lower levels (e.g., L6), through the key range. Notably, this multi-step process ensures efficient key lookups by leveraging the sorted nature of the LSM-tree. However, the read process across multiple levels also causes read amplification. The read amplification metric is defined as the ratio of the total number of data block reads to the number of reads that directly match the target key. To mitigate read amplification, we design the two-level index to directly locate the target KV pair in a zone.

2.1.2 Persistent Memory (PM). PM combines the byte-addressability of DRAM with the non-volatility of SSDs, making it an effective intermediate cache between DRAM and SSDs. PM enhances read and write performance while ensuring data consistency. PM provides higher read and write bandwidth compared with SSDs; however, it has some limitations, including higher write latency than DRAM ($7\times$ to $22\times$ greater) [36, 46] and the requirement for cache-line-aligned writes (e.g., 64-byte alignment) to avoid WA. Prior studies [11, 15, 24] rarely make full use of these characteristics of PM. To adapt to the cache-line-aligned write, we design PM-aligned storage zones and create storage zone pools in PM.

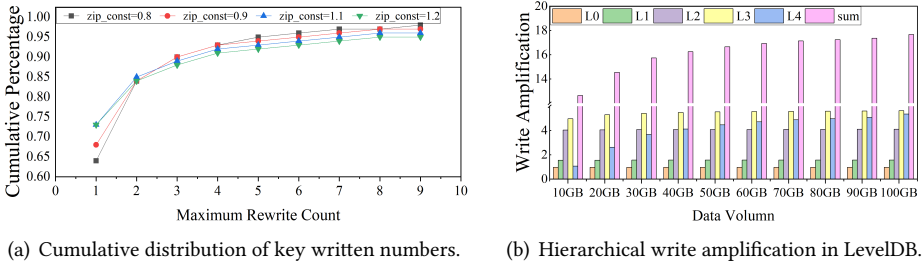


Fig. 2. Analysis of key written distribution and write amplification in LevelDB.

2.2 Motivation

In real-world workloads, most KV pairs are read-heavy and rarely updated. Once written to the LSM-tree, the majority are accessed primarily by read operations, while only a small fraction undergoes updates. In this study, we define KV pairs that remain unchanged after being written as “cold KV pairs”, while those that undergo updates are termed “hot KV pairs”. According to the prior from Facebook [2], approximately 73% of KV pairs in ZippyDB are cold, while only 27% are hot. Notably, among hot KV pairs, about 0.001% are updated more than ten times, which indicates that frequent-updated KV pairs constitute a small proportion. Furthermore, read operations exhibit strong spatial locality, with only a small subset of cold KV pairs being accessed frequently. These requests are typically clustered within specific periods, highlighting significant access skewness.

We conduct experiments on LevelDB under YCSB-C workloads to study the access characteristics of user keys. YCSB-C workloads are commonly used to simulate real-world requests. We configure 16-B key and 1-KB value in these experiments. We employ YCSB-C to generate 100 GB KV pairs randomly distributed following a Zipfian distribution and perform 100 million random-update operations to examine the update frequency distribution. Figure 2(a) presents the cumulative count of rewrites performed on distinct keys. Each key may undergo up to N rewrites, with N ranging from 1 to 9. The x-axis denotes the maximum number of rewrites, and the y-axis shows the cumulative percentage. When the Zipfian constant increases from 0.8 to 1.2, over two-thirds of the KV pairs are written only once, with this percentage rising as the Zipfian constant increases. When the Zipfian constant reaches 1.2, approximately 72.8% of the KV pairs are not updated after the initial write.

These results confirm that most KV pairs are not updated after being written to the LSM-tree. However, these KV pairs frequently participate in the LSM-tree’s compaction process, resulting in redundant writes and increased write amplification. To mitigate the write amplification caused by cold KV pairs, this study proposes a strategy to separate hot and cold KV pairs.

2.2.1 Write Amplification. To maintain read performance, newly written SSTables in level L0 must be merged with SSTables with overlapping key ranges in level L1. This merging process filters valid KV pairs, generates new SSTables, and writes them back to level L1. To maintain the order of KV pairs within each level, the newly generated SSTables must also undergo further compaction with adjacent lower levels. Consequently, from level L0 to L6, users’ KV pairs may undergo to multiple compaction operations, leading to high write amplification.

Notably, a substantial proportion of KV pairs in this process are cold (i.e., written only once and never updated); these cold KV pairs frequently participate in compaction due to minor overlaps in key ranges. Such unnecessary merging and reordering operations force cold KV pairs to repeatedly undergo compaction in a LSM-tree, thereby significantly increasing write amplification, consuming substantial I/O bandwidth, and adversely affecting overall performance.

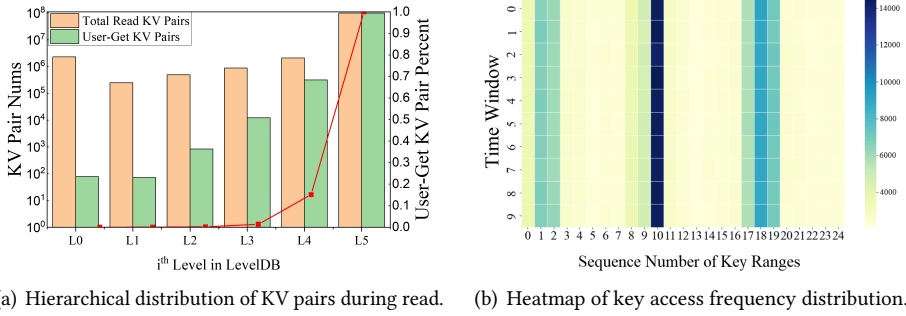


Fig. 3. Analysis of KV pair distribution and key access patterns. Colors represent different access frequency of KV pairs, where the lighter the color, the lower the access frequency. The darker, the higher the access frequency.

We conduct experiments on hierarchical write amplification in LevelDB with data volumes ranging from 10 GB to 100 GB, using DB_Bench workloads with random writes/reads. In Figure 2(b), the x-axis represents dataset sizes, whereas the y-axis denotes write amplification. The experimental results shown in Figure 2(b) reveal a clear upward trend in write amplification as the level increases. Specifically, the overall write amplification, defined as the ratio of data written to disk versus data written by the user, progressively rises with increasing data volumes. When the data size reaches 100 GB, the write amplification peaks at 17.67, indicating significant redundant writes in LevelDB. These findings demonstrate that the LSM-tree incurs substantial overhead from data rewrites as the data volume grows. Considering the proportion of KV pair updates, we design a hot-cold separation strategy to reduce write amplification caused by cold KV pairs repeatedly involved in compaction.

2.2.2 Read Amplification. The LSM-tree organizes KV pairs in a multi-level structure, with reads occurring sequentially from upper to lower levels. However, when the proportion of hot KV pairs is low, most read operations target cold KV pairs, which are typically not updated. These cold KV pairs are gradually migrated to the SSTables in lower levels, maintaining order within each level through successive compaction processes. Due to the randomized distribution of KV pairs in the LSM-tree, newly generated SSTables often overlap significantly in key ranges with those compacted into lower levels. This overlap causes read operations to span multiple levels, resulting in high read amplification. This issue is particularly severe for cold KV pairs in lower levels that are frequently accessed. The level-by-level lookup mechanism exacerbates read amplification, as each read operation requires accessing multiple SSTables across different levels, even though these cold KV pairs are no longer updated. This behavior increases read latency and consumes significant I/O bandwidth, ultimately degrading overall performance.

To analyze read amplification, we evaluate the distribution of read requests across multiple levels, as illustrated in Figure 3(a). We perform 100-GB random writes followed by 100-GB random reads. We record the total number of read I/Os per level (Total Read KV Pairs) as N_{read} , and the number of I/Os corresponding to user read hits (User-Get KV Pairs) as N_{hit} . Then, we have $R_{hit} = \frac{N_{hit}}{N_{read}}$, where R_{hit} denotes the percent of KV pairs that a user obtains per level (User-Get KV Pair Percent). The smaller the R_{hit} , the larger the read amplification. The results indicate that most read requests concentrate in the lower levels of the LSM-tree, which is the primary cause of severe read amplification. The multi-level management mechanism requires multiple block I/O operations to locate requested KV pairs, thereby increasing read amplification. To reduce read amplification, we design a two-level index to directly locate the target KV pair.

2.2.3 Strong Access Locality. To expedite read operations, LevelDB employs a file index block cache and a data block cache for SSTables. First, LevelDB identifies the SSTable that may contain the target key based on the key range and checks whether the index block is cached. If the index block is not cached, the index block will need to be read into memory via data block I/O. Then, LevelDB performs a binary search within the index block to locate the data block potentially containing the target key. Once the data block is identified, the system checks whether it is cached. If cached, the target key is read directly from the cache; otherwise, the data block is loaded into the cache via data block I/Os for subsequent accesses.

LevelDB strives to utilize a caching mechanism to reduce the number of I/O operations and enhance read efficiency. However, the frequent merging of SSTables during compaction causes previously cached data blocks to become invalid. This behavior significantly lowers the cache hit rate, and increases read latency, especially when the majority of KV pairs are cold. We perform a set of experiments on LevelDB under DB_Bench. The workload involves writing 100 million random KV pairs and 100 million random read operations. The value size is set to 1024B, with the key size set to 16B, and we use the default pseudo-random number generator to generate keys in DB_Bench. In Figure 3(b), the x-axis represents continuous key ranges, and the y-axis denotes time windows. The results reveal a distinct hot-and-cold localization in the access distribution of KV pairs. However, the data block cache hit rate during reads is nearly zero, confirming that the compaction process significantly disrupts spatial locality and thus reduces cache efficiency. To maintain the spatial locality of KV pairs, we design storage zones (zones) and a dynamic adaptive GC strategy for these zones in PM.

To mitigate the write amplification caused by compaction, existing solutions include two camps: decentralized compaction and KV separation. Decentralized compaction reduces the impact on read and write performance by adjusting the granularity or timing of compaction, thereby minimizing the size of individual compactions. Although this approach avoids the I/O overhead of large-scale consolidations, it does not fundamentally reduce the total compaction overhead; it merely defers and redistributes it. As a result, the issue of write amplification persists as data volume increases.

KV separation, on the other hand, stores keys and values separately while indexing the locations of values. During compaction, only the keys and indexes are merged, minimizing duplicate writes to the values and thereby reducing write amplification. However, this approach introduces additional I/O overhead during reads, as it requires accessing the value index first before retrieving the value from the value log file. Furthermore, invalid values in the value log file still require periodic reclamation. Existing strategies, e.g., scanning of value log files and partitioned reclamation in FenceKV, often overlook scenarios where a large number of cold KV pairs remain unchanged. This oversight leads to significant write amplification during the value reclamation process. We use DB_Bench to test 100 million random writes and reads on Wisckey. The write amplification reaches 3.39 under default parameters. This suggests that while KV separation schemes mitigate write amplification, they fail to fully address the issue in the value log file. Thus, developing an efficient method to reclaim the invalid space occupied by hot KV pairs remains challenging.

To address read and write latency, studies have proposed hybrid storage designs that leverage PM. PM not only reduces read-write latency but also ensures data consistency. For instance, approaches like NovelLSM and LightKV use PM as an upper caching layer for the LSM-tree, enhancing read and write performance through its caching capabilities. SLM-DB further improves performance by storing key indexes directly in PM, reducing index read latency, while HPDK uses PM as a complete replacement for SSDs to boost storage performance. However, despite its touted advantages, storing all KV pairs in PM is impractical for large databases.

While the aforementioned methods leverage PM to enhance read and write performance, most approaches merely replace the storage medium without fully considering the design of suitable

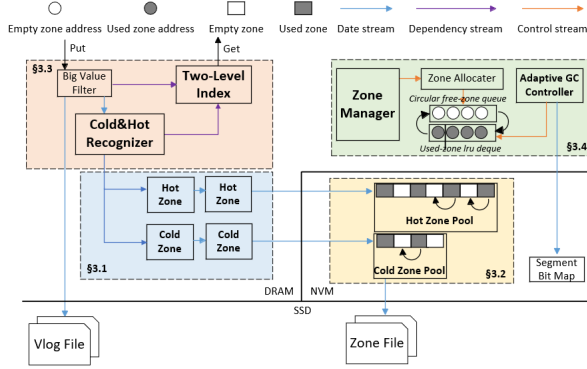


Fig. 4. The overview system of HAKV.

data structures for PM. Consequently, the underlying compaction process continues to cause write amplification. Moreover, the lack of separation between hot and cold KV pairs in PM leads to underutilization of PM storage, particularly when optimizing the storage of frequently-accessed data. Thus, efficiently utilizing PM to store frequently accessed data and reduce read latency remains a pressing challenge.

To address the aforementioned issues, we propose a hybrid storage system for PM on the basis of access pattern separation. The overarching goal is to separate a small subset of frequently updated hot KV pairs from a large number of cold KV pairs that are primarily read-only. These pairs are preprocessed and stored separately according to their access and update characteristics. This design prevents unnecessary compaction of cold KV pairs caused by the invalid version reclamation of hot KV pairs, thereby reducing write amplification. To facilitate fast reads of cold KV pairs and efficient space reclamation for hot KV pairs, we design separate zone pools in PM for hot and cold KV pairs. These pools manage allocation, release, merging, and flushing operations to optimize performance. Unlike traditional KV separation methods that rely on sequential scanning and reclamation of value log files, our approach selectively merges and reclaims data based on access characteristics through zone-based data management, thereby reducing write amplification.

3 Design of HAKV

HAKV is an efficient KV store designed to minimize write amplification and to enhance performance by leveraging PM to store KV pairs in designated storage zones. Figure 4 presents the overview of HAKV, which comprises the storage zone, PM zone pool, the two-level index for keys in PM and values in the SSD, and the dynamic GC strategy of storage zones in PM. HAKV can separate frequent-updated hot KV pairs from cold KV pairs and store them independently, thereby preventing the unnecessary impact of hot KV pairs on cold KV pairs during rewrites. HAKV employs a two-level key-location indexing structure in DRAM to quickly identify and access KV pairs.

In HAKV, to accelerate the write performance of cold KV pairs, we design an append-write zone structure that takes advantage of the spatial locality of cold KV pairs. The PM zone pool applies a circular free-zone queue to facilitate zone allocation and recovery in PM, ensuring effective PM space management. We adopt two-level index to distinguish the hot and cold data and to efficiently locate KV pairs in hot and cold zones. The first-level index is a zone index, and the second level is a location index. This design separates hot and cold data, thereby curtailing the write amplification caused by cold KV pairs. The dynamic GC reclaims zone in the pool in PM and zone files in SSD.

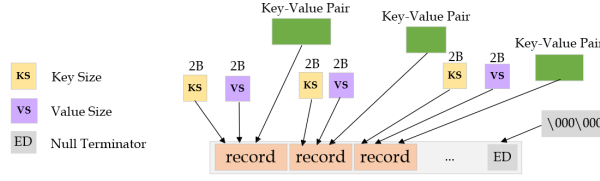


Fig. 5. Storage zone in HAKV. Each record in a zone contains KV pairs and metadata. A zone contains multiple consecutive KV pairs and a terminator consisting of two null characters.

The GC in PM adopts used-zone LRU deque and the circular free-zone queue. According to the load and the available space, HAKV executes GC operations based on the GC window to keep frequently-accessed zones in PM. The in-situ rewrite policy is deployed to write new zones to the invalid space in the zone file to lower the GC cost.

During the write process, HAKV first uses the two-level index to determine whether a KV pair is hot or cold, and then stores the data in a DRAM zone. When the zone is full, HAKV allocates a new zone from the PM zone pool and writes the buffer data to it. As the number of PM zones increases, HAKV releases used-zone space in PM. Meanwhile, we store frequently-accessed zones in PM as many as possible; therefore, we use a used-zone deque and a circular free-zone queue to manage the used zones and the free zones, respectively. During the read process, HAKV first searches for the target KV pair from the DRAM zone. If the KV pair is not found in the DRAM zone, the two-level index is used to check its hot/cold status and identify its zone location. HAKV then queries the PM zone or SSD based on the zone location, until the KV pair is either found or determined to be absent.

3.1 Storage Zone

Based on the I/O-access granularity of PM and the SSD, we design a KV-pairs storage structure named storage zone, optimized for PM and I/Os. In HAKV, all zones in PM and the SSD have the same size. A cold zone including cold KV pairs ensures sequential-write KV pairs to be stored in the zone. A hot zone stores hot KV pairs. When a KV pair in the cold zone is updated once, this KV pair is converted to a hot KV pair. Since most KV pairs are cold, this approach prioritizes quickly writing these cold pairs without sorting them in the memtable. The optimal write strategy involves appended writes in the order of incoming requests, reducing sorting overhead, accelerating writes, and preserving the access locality of cold KV pairs. To meet PM's cache line alignment requirements, we configured an appropriate size for a zone, which can maximize PM bandwidth utilization and minimize write amplification. Additionally, we developed a specialized record format in a zone to determine the validity of KV pairs and identify the end boundaries of KV pairs in a zone during reclamation. This record format optimizes GC efficiency while ensuring fast writes within the zone.

In Figure 5, the zone encapsulates write KV pairs into fixed-size blocks with a default size of 8 KB. Each zone stores multiple sequentially appended KV pairs. To ensure accurate identification of KV pair boundaries during merge and reclamation operations, a record includes key size (2 B), value size (2 B), key, and value. To facilitate the identification of the valid end position of the zone during reclamation, a 2-B “\0” terminator is appended to the end of all records. Notably, large-sized KV pairs that exceed the zone size are stored in a dedicated large vlog file.

HAKV separates cold and hot KV pairs by storing them in distinct zones, cold KV pairs in cold zones, and hot KV pairs in hot zones. The zones are designed to align PM write operations with zone boundaries, ensuring proper write alignment. Similarly, when data is swapped in and out of the PM zone, read/write operations are conducted in a zone unit, optimizing the access performance of the

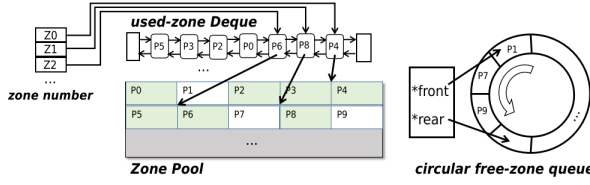


Fig. 6. PM storage zone pool space management.

zone file, which is divided into multiple data blocks of the same size as a zone and is responsible for storing cold zones. Furthermore, since cold KV pairs are stored in the same zone, even during merge and reclamation processes, these cold KV pairs remain aggregated rather than fragmented, preserving their access locality within the zone.

3.2 PM Zone Pool Management

As illustrated in Figure 6, HAKV accelerates read performance for frequently-accessed KV pairs by managing the PM space as a fixed-size zone. The PM space is further divided into a cold zone pool and a hot zone pool, which cache cold zones from cold zone files and frequent-updated hot zones, respectively. To efficiently allocate and reclaim zone space, the PM space is partitioned into multiple contiguous physical blocks, each corresponding to the size of a zone.

These allocation and reclamation operations are optimized by managing blocks through circular queues, consisting of a circular free-zone queue and a used-zone deque for PM zones. HAKV employs the circular free-zone queue to track the zone location in the PM zone pool. A used-zone deque is implemented to manage the used zone numbers and zone location pointers. When a zone is allocated, HAKV retrieves the block pointer—from the free-zone queue—to be moved to the used-zone deque. This FIFO-based allocation simplifies metadata management and eliminates the cost of search or reordering in the free-zone queue, thereby lowering the overhead of allocation and reclamation. During reclamation, HAKV releases the pointer at the end of the used-zone deque, followed by placing it into the tail of the circular free-zone queue. The rear position of the circular free-zone queue is updated accordingly. FIFO avoids random insertions or deletions in the circular free-zone queue. Overall, this design leverages a FIFO-based scheduling strategy, offering low overhead and enabling fast allocation and reclamation within the PM zone.

When the buffer of a zone in DRAM is insufficient, HAKV allocates a physical block in PM for writes. To locate the physical block address of a zone in PM, HAKV establishes a mapping between the zone number and the block address. HAKV employs a hash mapping mechanism for the used-zone deque, enabling rapid retrieval of a zone's location in PM based on its zone number. Similarly, HAKV identifies the corresponding physical block during reclamation, reclaims it, and returns it to the free-zone queue. This mapping mechanism significantly enhances the efficiency of read and reclamation operations in PM, effectively reducing access latency. By partitioning the PM space into uniformly sized physical blocks and managing free space with a circular queue, this design ensures byte alignment in PM space allocation and enhances the efficiency of zone allocation and reclamation. Additionally, it mitigates the challenges of frequent dynamic allocation and reclamation associated with smaller spaces. This unified allocation and reclamation mechanism enables HAKV to efficiently manage PM resources while maximizing the performance of PM.

3.3 Two-Level Index for Keys in PM and Values in the SSD

In HAKV, users' written KV pairs are stored in zones, where the KV pairs are unordered. To enable fast location and retrieval, HAKV employs an efficient two-level index structure that supports

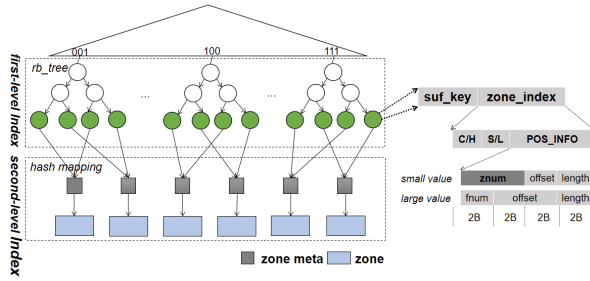


Fig. 7. Two-level index in HAKV. The first-level index employs red-black trees to store the zone information and offset indexes of keys. The second-level index utilizes a hash mapping to index the zone location in PM and SSD.

range queries and parallel insertions. The first-level index linking a key and its zone number $\langle \text{Key}, \text{ZoneID} \rangle$ is referred to as the zone index. The second-level index connecting the zone number and the zone location $\langle \text{Zone ID}, \text{ZonePos} \rangle$ is named as location index. The first-level index identifies the zone or file location of a key based on the key ranges managed by red-black trees instead of a skiplist [28] or B+ tree. First, the level structure of the skiplist is determined by its random number generator, which can lead to worst time complexity $O(n)$ for insertion and lookup operations. Second, B+ tree can move many nodes during an insertion. However, the red-black tree can achieve time complexity $O(\log(n))$ for insertion and query operations in the worst case due to its balance structure. This structure enables a system to guarantee stable performance and reduce peak latency. Prior work [21] reveals that the red-black tree outperforms the skiplist in query performance under large data volumes. In addition, the insertion and query time of the skiplist is higher than that of the red-black tree because the skiplist requires many level-structure adjustments during updates [1]. To support range scan and low latency, HAKV employs multiple red-black trees with different key ranges to prevent performance jitter during its insertion and deletion under write-intensive workloads. Figure 7 shows the zone information and offset indexes of keys in the nodes of red-black trees. The zone information includes the hot and cold zone identifier (1 bit), value size identifier (1 bit), zone number (30 bits), offset (2 bytes), and length (2 bytes). For large KV pairs, we store them using a format that includes identifiers (2 bits), file number (14 bits), offset (3 bytes), and length (3 bytes).

HAKV incorporates a second-level index powered by hash mapping to optimize the zone management in PM and SSD. Since multiple KV pairs reside in the same zone, the keys in the first-level index and the zone location pointer in the second-level index form a “many-to-one” relationship: multiple keys are mapped to the same zone location pointer, and a zone number is associated with the zone location pointer and the zone’s valid data size and state. When a zone is written back into the SSD during GC (see Section 3.4), the storage location of the KV pairs within the zone may be changed. Thus, if the key-zone pair index is employed, the index ought to be updated simultaneously for multiple KV pairs to be written back to the zone. Nonetheless, the two-level index scheme only requires modifying the pointer of the zone location in the second-level index, thereby significantly cutting back the index-update overhead through eliminating duplicated updates to multiple first-level indexes when KV pairs in a zone are merged with ones in other zones.

To separate frequent-updated hot KV pairs from read-only cold KV pairs, HAKV first identifies which KV pairs are likely to be repeatedly updated. When a KV pair is initially written, it is impossible to determine whether it is cold or hot. In addition, HAKV cannot predict whether the KV pairs to be updated. Therefore, HAKV stores the KV pair in the cold zone by default. If the KV

pair is subsequently updated, HAKV marks it as a hot KV pair and migrates it from the cold zone to the hot zone. Hot KV pairs are stored in the PM hot zone. To prevent the hot zone pool from growing indefinitely, HAKV monitors the space utilization of the hot zone pool. When the ratio of used-zone space exceeds a threshold (e.g., 80%), zones unaccessed for a long period are flushed to SSD. This procedure frees up PM space and avoids the unlimited increment of hot KV pairs in the hot zone pool.

Although some invalid hot KV pairs (i.e., outdated versions of updated keys) may remain in the cold zone, most of the KV pairs in this zone remain valid, as the proportion of hot KV pairs is small. This delayed separation strategy effectively reduces invalid space in the cold zone, and old versions of hot KV pairs are cleaned up during subsequent reclamation processes. It is challenging to quickly determine whether a KV pair is stored in the cold or hot zone during read and write operations. To address this issue, we designed a concise hot-cold distinction strategy. The first two bits of the zone index in the first-level index are used to fix the location of KV pairs. The first bit distinguishes between hot and cold zones: “0” and “1” represent the cold zone and hot zones, respectively. The second bit differentiates between common KV pairs and large-sized ones.

When a KV pair is initially written, HAKV sets the first bit of the node to 0, indicating that it is stored in the cold zone. If the KV pair is updated, the first bit is modified to 1, signifying that it has been migrated to the hot zone. In the first-level index, the hot/cold index state of the KV pair synchronously determines whether it should be placed in the hot zone. The background GC threads are slated to reclaim invalid KV pairs generated due to the index state change. Thus, the GC operations have no interfere with writes (see Section 3.4). For large-sized KV pairs, HAKV appends them to a large-sized value log file and sets the second bit of the node to 0. The specific file number, offset, and length are then used to locate the large-sized KV pair. This marking mechanism enables efficient differentiation between hot and cold KV pairs, ensuring accurate determination of their locations in zone files.

3.4 Zone Pool Reallocation

To effectively manage zones, we design a zone mapping table to keep the zone metadata in the second-level index, which embraces the zone location pointer, zone states, and valid data size. The zone location pointer identifies the current position of the zone. When the zone is in the PM zone pool, the pointer indicates the address of the zone. If the zone is flushed to the zone file on the SSD, the pointer will signify the file offset in the zone file. Four zone states are state 0, state 1, state 2 and state 3. The zone valid data size is the data volume of the valid KV pairs. These zone metadata assist HAKV to pinpoint the value of the target key and to perform GC operations of the zone in PM.

While reading a KV pair, HAKV first obtains the zone number and the key offset in the zone from the first-level index. Then, the mapping table is used to retrieve the zone location pointer according to the zone number. If the targeted value is in the zone pool, the retrieved value is returned to users. If the value is located in the SSD’s zone file, HAKV loads the zone into the PM and returns the value. When the KV pair becomes invalid due to an update, the valid data size of the zone is also updated. When the percentage of valid data in the zone is lower than the threshold, HAKV relocates the valid data to a new PM zone during GC; otherwise, the zone location pointer references to the data block in the zone file.

For the zone state, it is represented by 2 bits, which can represent the four states below. State 0: The zone is in DRAM (set to this state upon creation). State 1: The zone has been written to PM from DRAM but has not yet been flushed to the zone file on the SSD. State 2: The zone has been flushed to the zone file on the SSD, with no copy retained in PM. State 3: The zone is read from the zone file on the SSD, cached in PM, with a copy retained in the cold file. As shown in Figure 8, when the zone is first written to PM, its state transitions from state 0 to state 1. Once the background

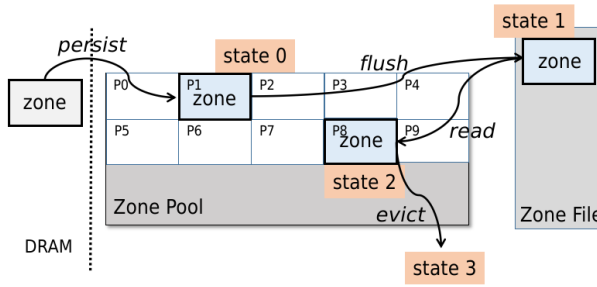


Fig. 8. Different states of the zone in HAKV.

thread flushes the zone to the zone file, the state changes from state 1 to state 2. When the zone is read and cached back into PM, its state converts to state 3. If the zone is selected for swapping out by the background thread while in state 3, its state becomes state 2. This enables HAKV to manage swap-in and swap-out operations and state transitions according to the current zone state, thereby reducing unnecessary I/Os from the SSD to PM.

As the number of allocated cold zones increases, the space utilization of the pool can be gradually improved. When the remaining space in the cold zone pool falls below a predefined threshold (default set to 20%), HAKV releases a portion of zone pool space to ensure the allocation of subsequent cold zones to be completed. To prioritize the most recently accessed zones, HAKV monitors access to the cold storage pool in real-time. Upon reading each zone, HAKV updates the used-zone deque based on the zone number, moving the most recently accessed zone to the front of the queue. With only write operations and no read operations, the newly written zone is inserted at the head of the used-zone deque. Because the write is the most recent operation, and zone's hotness is merely determined by the order in which it is written. A zone written earlier is colder than those written later. Consequently, if the workflow excludes read operations, HAKV prioritizes reclaiming the earliest written (coldest) zones once the PM space is constrained. This enables HAKV to prioritize the reclamation of less frequently-accessed zones during the background reclamation process, thereby achieving efficient space reclamation.

When PM space is insufficient, HAKV reclaims zones that have not been read for the longest duration. Based on the state of each zone, HAKV applies different reclamation strategies to minimize unnecessary file I/Os and to fully leverage PM's high read and write performance. HAKV sets a merge threshold to figure out whether a merging operation in the reclamation strategy should be performed in PM. The percentage of valid data in the zone is defined as the ratio of the zones' valid data size to the total zone size. HAKV sparks the merge operation to be performed in PM when the percentage of valid data is lower than $1/4$ —the threshold; otherwise, merge operations are averted. This threshold is a configurable parameter in HAKV: a high threshold value leads to more cold zones being involved in the merging process in PM. Therefore, merging will generate more new zones, leading to higher write amplification in both PM and SSD zones. Since the cold zone pool contains fewer invalid KV pairs, the overall impact is minimized, which is shown in the experiments to study the impact of this threshold on write amplification in Section 4.5.4.

The GC strategy is presented in Algorithm 1. First, the tasks to reclaim the zone are retrieved from the recycling tasks multi-queue (Line 4). Then, for each zone in the to-be-released queue (Line 5–6), if the zone state is state 1, the zone has been written to PM but has not yet been flushed to the zone file in the SSD. If the valid data percentage exceeds merge threshold, HAKV flushes the zone to the zone file and records its position in the second-level index (Lines 10–12). If the percentage is below the threshold, HAKV iterates through the valid records in the zone, and then writes them

ALGORITHM 1: PM Zone-Based Garbage Collection Strategy in HAKV

```

1: Input: GC task multi-queue:  $GTM$ 
2: Output: Reclaimed and optimized PM zones
3:
4:  $GZS \leftarrow GetZonesToMerge(GTM)$ 
5: if  $GZS \neq \emptyset$  then
6:   for  $Z_i \in GZS$  do
7:     if  $ZoneState(Z_i) == ZS.State1$  then
8:       if  $ValidSizePercent(Z_i) == 0$  then
9:          $RemoveZoneIndex(Z_i)$ 
10:      else if  $ValidSizePercent(Z_i) > T_{vsp}$  then
11:         $FlushIntoZonesFile(curZonesFile, Z_i)$ 
12:         $UpdateZoneState(ZS.state2, Z_i)$ 
13:      else
14:         $IterateAndMerge(Z_i)$ 
15:         $RemoveZoneIndex(Z_i)$ 
16:      end if
17:      else if  $ZoneState(Z_i) == ZS.State3$  then
18:        if  $ValidSizePercent(Z_i) == 0$  then
19:           $RemoveZoneIndex(Z_i)$ 
20:        SetInvalid( $curZonesFile, Z_i$ )
21:        else if  $ValidSizePercent(Z_i) > T_{vsp}$  then
22:           $UpdateZoneState(ZS.state2, Z_i)$ 
23:        else
24:           $IterateAndMerge(Z_i)$ 
25:           $RemoveZoneIndex(Z_i)$ 
26:           $SetInvalid(curZonesFile, Z_i)$ 
27:        end if
28:      end if
29:    end for
30:  end if
31:  $ReleaseZoneSpace(Z_i)$ 

```

to the current reclaimed zone (GC_ZONE), deletes the information of the old zone, and updates both the new zone and the associated key-index information (Lines 13–15). When the zone state is state 3 (Line 17), the zone exists in both the PM and the zone file. If the percentage of valid data exceeds the merge threshold, HAKV releases the PM space and changes the zone state to state 2 (Lines 21–22). Otherwise, if the valid data percentage is below the threshold, HAKV merges the valid KV pairs from the zone into the current reclaimed zone, updates the new zone information, deletes the old zone, and marks the corresponding old zone in the zone file as invalid (Lines 23–26), which can help to achieve in-place file rewrite.

The invalid KV pairs in zone files are primarily caused by initial version failures of a few hot KV pairs due to multiple update writes. These invalid data constitute a small portion of the total data. Since cold KV pairs are written only once, write amplification of zone files during reclamation becomes minimal, leading to high space utilization in the zone files.

To balance the coordination between the foreground load thread and background GC thread, HAKV implements a dynamic adaptive GC window strategy to adjust the GC rate of recovery threads. During foreground writes, it is crucial to maintain sufficient unused zones in PM for allocation while preserving frequent-read cold zones to maximize the read hit rate of PM zones.

When determining the proportion of zones to be reclaimed from the PM zone pool, HAKV must consider both the write and read loads of the foreground. This ensures that write operations are not blocked while minimizing the number of zones flushed to zone files. To balance read and write performance, HAKV sets a GC window threshold, which defines the percentage of zones in the used-zone deque to be reclaimed during each GC procedure, with a default initial setting of 20%. By dynamically adjusting this threshold, HAKV balances the rates of the foreground load thread and background GC thread, thereby optimizing its overall performance.

As shown in Figure 9, HAKV performs GC operations using a circular free-zone queue and a **least recently used (LRU)** used-zone deque. When a new zone is required, HAKV allocates it from the circular free-zone queue. Active zones are managed through the used-zone LRU queue to prioritize reclaiming based on zone usage frequency. The GC window dynamically selects zones from the LRU queue for GC, ensuring that zones with less recent activity are given higher priority for reclaiming. During the GC process, HAKV offloads the selected zones to the multitasking queue, enabling parallel processing and reducing latency. The multitasking queue ensures that reclaimed zones are efficiently processed and returned to the free-zone queue, thereby maintaining a continuous balance between zone allocation and reclamation.

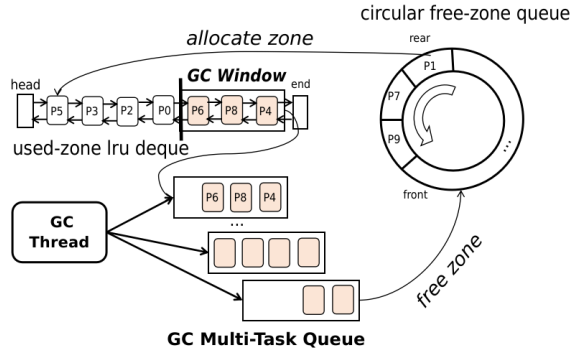


Fig. 9. Adaptive GC for PM zones in HAKV.

When the foreground write thread requests PM zone allocation, HAKV monitors the percentage of available zones in the pool in real-time. If the remaining space falls below the predefined threshold, when HAKV detects low available space in the PM zone pool and triggers a GC operation, It reclaims an appropriate number of cold zones using an exponential fast-start and linear slow-recovery model. This approach maximizes the space utilization of the PM zone pool while adapting to the changing workloads of the foreground thread. For instance, if zone allocation fails for the first time within the same duration, HAKV increases the GC window threshold by 0.5% and wakes up the background GC thread for recovery. If the allocation fails again after the GC completes, HAKV increases the GC window threshold by twice the previous increment (i.e., 1%), and so on, until it reaches the set upper limit of the GC ratio (default is 30%). If no allocation failure occurs within the set time window, HAKV gradually reduces the GC window threshold using a linear reduction strategy. It reduces the window size by 0.5% for every cycle until it reaches the smallest size of the GC window. This dynamic adjustment mechanism enables HAKV to maintain optimal utilization of the PM zone pool in response to workload changes in the foreground.

In HAKV, the number of invalid zones resulting from repeated writes is relatively small due to the low percentage of hot KV pairs in zone files. To avoid file read/write operations on zone files, we introduce a sparse bitmap mechanism for managing invalid blocks in zone files. The zone file is divided into multiple segments. For each segment, we record the bits of invalid blocks in the zone file and the number of these bits (i.e., one bit “0” representing an invalid block). Thus, we can perform the segment query, and then rewrite data into these invalid blocks in the segment. This mechanism efficiently marks invalid zones in zone files and enables in-situ recovery. By leveraging the sparse bitmap’s efficient spatial management and fast localization capabilities, HAKV effectively manages invalid zones in zone files, reducing the need for file read and write merge operations.

When the proportion of invalid records in a cold zone becomes too high, HAKV merges the valid records into a new zone, then the file block is occupied by the merged zone will be invalid. To enable the reuse of invalid space of zone files, a sparse bitmap is used to record the offset addresses of invalid zones in the zone file. Since the size of each zone is fixed, the new generated zones can be written to the locations of invalid zones marked in the sparse bitmap, thereby avoiding file read/write operations on zone files and reducing I/O overhead between PM and the SSD. Specifically, the sparse bitmap structure manages invalid blocks by partitioning the zone file and recording only the portions that contain invalid zones. It tracks the location of invalid blocks within each segment using a local bitmap. During zone merging operations, HAKV first queries the sparse bitmap to identify the locations of invalid blocks and writes new valid zones to these blocks, updating the

sparse bitmap to reflect the reuse of zone file space. This design ensures the utilization of invalid space while reducing the frequency of file-level GC operations.

HAKV significantly reduces the GC cost in zone files by managing invalid blocks using sparse bitmaps. Combined with PM's management of the zone pool, this approach optimizes the read performance of user KV pairs, improves the system performance, and enhances space utilization.

3.5 Discussion

Parallel Scan. Since HAKV stores KV pairs as unordered units within zones, it is crucial to optimize scan operations. Thus, we in HAKV designed a two-level index that supports parallel execution across the primary subrange indexes. HAKV identifies the key types in each subrange index and performs multi-threaded, parallel reads across large-sized vlog files, cold zones, and hot zones. This parallel processing design ensures efficient and scalable scan performance.

Large-sized values processing. For large-sized values, we implemented hot and cold separation based on access frequency, similar to the KV separation that uses file indexing for vlog files. Frequent-updated large-sized values are appended to hot vlog files, while infrequently updated values are stored in cold vlog files. This append-only write strategy minimizes write amplification and enhances write performance. For reclamation, we prioritize reclaiming hot vlog files with periodic reclamation of cold vlog files to ensure optimal space utilization.

4 Performance Evaluation

We first introduce the experimental environment and configuration. Second, we analyze the experimental performance under DB_Bench and **Yahoo Cloud Service Benchmark (YCSB-C)** [37].

4.1 Experimental Environment

In this section, we introduce workloads, settings, and the alternative KV stores in the experiments.

Workloads. We perform experiments using DB_Bench and YCSB-C workloads. Under DB_Bench workloads, we first study the performance of HAKV under workloads with 100 GB random writes followed by 100 GB random reads. We measure the performance of HAKV in terms of write amplification, read and write throughput, and latency distribution. The key size is 16B. We conduct comparisons across different value sizes, including 128B, 512B, 1KB, 4KB, and 16KB. We use YCSB-C to evaluate system performance across workloads—Load (100% writes), A (50% reads, 50% updates), B (95% reads, 5% updates), C (100% reads), D (95% reads, 5% inserts), E (95% scans, 5% inserts), and F (50% reads, 50% read-modify-writes). Each workload executes corresponding transactions after loading 100 million KV pairs, except for workload E, which uses the default maximum scan length of 100 and involves 10 million transactions. In YCSB-C, the key size is 16B, and the value size is 1KB. We also apply three Twitter's traces[38] – cluster12, cluster37, and cluster43 – to evaluate HAKV performance under real production workloads. Cluster12 is a write-intensive workload consisting of 80% set and 20% get operations, with an average 44-byte key and 1058-byte value. Cluster37 is a read-intensive workload, featuring 35% set and 65% get operations, with an average 72 bytes key and 15,001 bytes value. Cluster43 is a mixed read/write workload, with 50% set and 50% get operations. The average key and value sizes are 44 bytes and 2,128 bytes, respectively.

Alternative KV Stores. To evaluate the performance of HAKV, we employ several state-of-the-art KV stores: LevelDB [20], RocksDB [10], NoveLSM [16], LightKV [11], Wisckey [25], and UniKV [43]. NoveLSM leverages PM to store the Memtable and write-ahead log of the traditional LSM-tree structure, while LightKV uses PM to cache segment content, enabling faster access to hot data. Wisckey adopts a KV separation approach for value indexing, optimizing storage and retrieval efficiency. UniKV mitigates compaction overhead by designing hash indexes for level L0 and employs a single-level ordered file structure to manage KV pairs at level L1 and below.

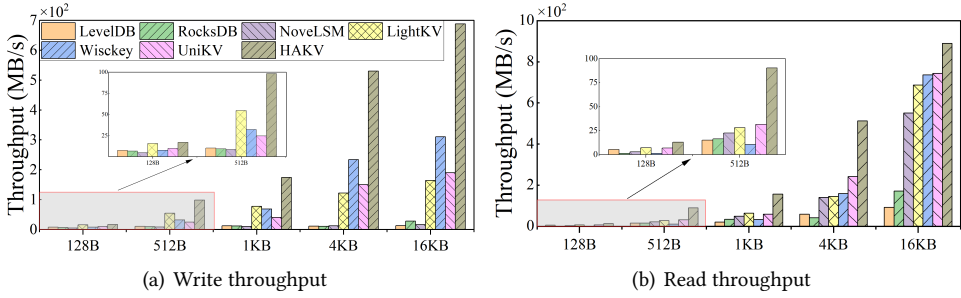


Fig. 10. Impact of different value sizes on throughput of HAKV.

Experiment Settings. We implemented HAKV based on LevelDB (version 1.20), a widely used KV store developed by Google. In this experiment, we use Intel Optane DC PM as the middle-tier device and map the PM to the process space through the `pmem_map_file` interface in PMDK. All experiments were conducted on a Dell PowerEdge MX840c server equipped with an Intel Xeon Platinum 8153 CPU (2.0 GHz, 16 cores), 16×16 GB RDIMMs, 24×128 GB Intel Optane DC PM modules, and Samsung PM1643A SAS SSDs, running Ubuntu 22.04 LTS. The default zone size is 8KB, and the dynamic window adjustment range is between 5% and 30% of PM's used zones. Each KV store is configured in the `kNoCompression` uncompressed mode, and all other configurations are set to the default values.

4.2 Performance of HAKV under DB_Bench

Random-Write Performance. Figure 10(a) demonstrates that HAKV significantly outperforms LevelDB (baseline), and other KV stores in terms of random-write throughput. Specifically, its write throughput is up to $54.2\times$ higher than that of LevelDB, $4.3\times$ higher than that of UniKV, and up to $3.6\times$ higher than that of LightKV. This performance advantage is primarily attributed to the design of the zone and the direct persistence of data to the PM zone pool via memory-mapped copies. Although NoveLSM and LightKV also use PM as a cache, their write throughput is lower due to the large number of sorting and merging operations performed during cold data processing.

Random-Read Performance. As shown in Figure 10(b), HAKV achieves high random-read performance. Compared with LevelDB, HAKV's random read throughput is improved up to $9.6\times$. Compared with UniKV, it is improved up to $2.9\times$. The random-read performance of HAKV is improved up to $3.5\times$ than that of LightKV. The performance advantage of HAKV is especially noticeable when handling small-sized values because of the two critical optimizations in HAKV. First, HAKV employs a dynamically adaptive zone reclamation mechanism to efficiently cache frequently-accessed zones in PM. Second, HAKV is designed with two-level index to enable direct and fast KV pair localization, thereby avoiding write amplification caused by multi-level reads.

Read Latency. Figure 11 shows the read tail latency across varying value sizes (ranging from 128B to 16KB). The experimental results demonstrate that HAKV significantly improves latency, particularly in P50, P90, and P99 tail latency, outperforming other KV stores across most value sizes. Compared with LevelDB, HAKV achieves substantial optimizations: a 66% (minimum) to 95% (maximum) reduction in P50 tail latency, an 82% (minimum) to 95% (maximum) reduction in P90 tail latency, and a 77% (minimum) to 96% (maximum) reduction in P99 tail latency. Notably, the reduction in P99 tail latency is most pronounced in high-tail latency scenarios, and the P99.99 tail latency maintains a significant advantage with small values. These improvements are primarily attributed to HAKV's hot and cold separation mechanism design and the dynamic window reclamation mechanism for the PM zone pool. By caching frequently-accessed KV pairs in PM, the number of

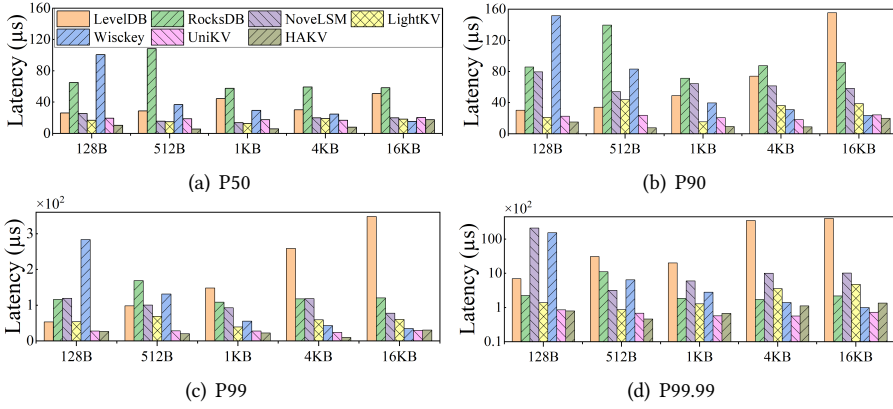


Fig. 11. Read latency under DB_Bench workloads with various value sizes.

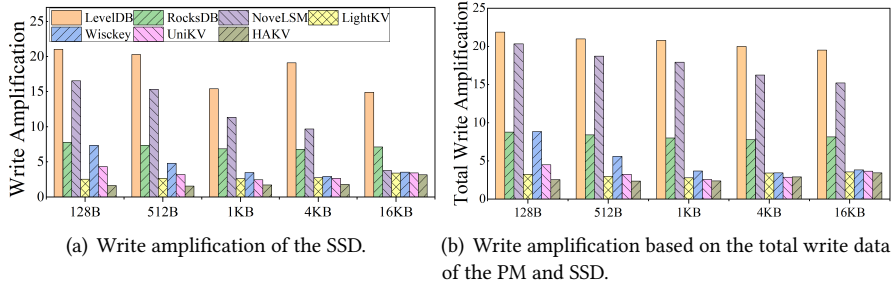


Fig. 12. Write amplification of HAKV under DB_Bench workloads.

reads in SSD are reduced, which lowers the high latency of access to the cold KV pairs. The high access performance of PM provides a substantial improvement in the overall read latency of HAKV.

Write Amplification (WA). Figure 12(a) illustrates WA of HAKV. HAKV consistently maintains low WA across all value sizes, demonstrating a clear advantage. Compared with LevelDB and RocksDB, HAKV significantly reduces WA, highlighting its superior efficiency and performance in most value sizes. Specifically, HAKV's WA has a maximum of 3.15 and an average of 1.97 for value sizes ranging from 128 B to 16 KB, while LevelDB's WA reaches a maximum of 21.04 with an average of 18.15. HAKV's WA is 89.6%, 42.5%, and 32.3% lower than that of LevelDB, UniKV, and LightKV, respectively. This demonstrates that HAKV effectively reduces WA among different value sizes, particularly excelling under workloads with small-sized values (e.g., 128 B and 512 B).

The total write amplification, defined as the ratio of total data written to both PM and SSD relative to the user write data, see Figure 12(b). The overall trend closely follows that of SSD WA. In contrast, the NoveLSM, despite employing a skiplist data structure in PM, exhibits higher WA, as most of the data still needs to be merged from SSD, particularly for smaller KV pairs. Wiskey struggles with minimizing WA due to the absence of hot/cold separation, leading to higher amplification than HAKV. UniKV achieves lower WA through the hash indexing in level L0 and partitioned storage in lower levels, particularly with large KV pairs. UniKV's overall WA remains slightly higher than that of HAKV. LightKV reduces write amplification by utilizing a prefixed hash tree structure; however, its write amplification remains higher than that of HAKV due to the lack of hot and cold data separation. In contrast, HAKV minimizes invalid writes through its hot and cold separation strategy and the dynamic adaptive reclaiming mechanism in PM, ensuring consistently low write

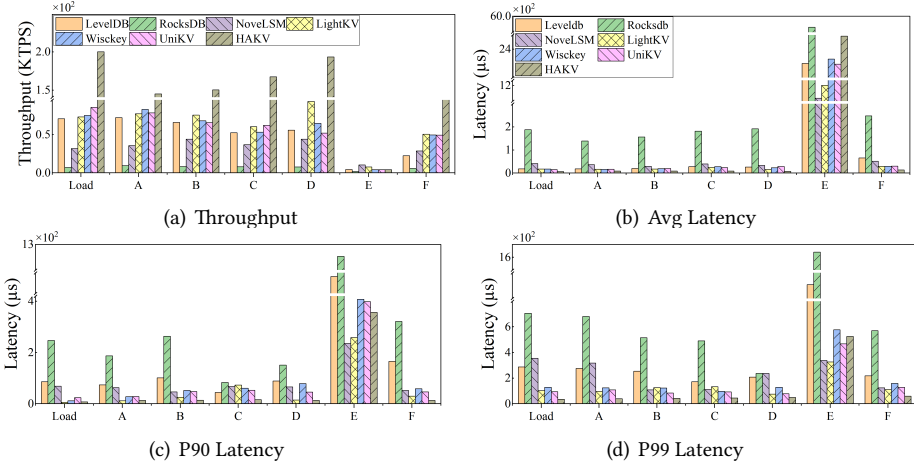


Fig. 13. Throughput and latency of HAKV under YCSB-C.

amplification across different value sizes. This makes HAKV particularly well-suited for storage scenarios requiring low write amplification.

HAKV adopts a hot-and-cold data separation mechanism, where cold KV pairs are initially cached in the cold zone pool in PM. When the space of the cold zone pool becomes insufficient, the infrequently accessed KV pairs will be flushed to the cold-zone files on the SSD. Hot KV pairs are first stored in the hot zone pool in PM. When the hot zone pool space is inadequate, the hot KV pairs will be written into the hot-zone files on the SSD. The hot-and-cold data separation mechanism in HAKV prevents cold KV pairs from being frequently involved in compaction. Hence, redundant writes of cold KV pairs are reduced, thereby lowering unnecessary I/Os on SSD and extending SSD lifetime.

4.3 Performance of HAKV under YCSB-C

As shown in Figure 13(a), HAKV demonstrates a significant advantage in load throughput compared with LevelDB, LightKV, and UniKV, and achieves $2.83\times$, $2.73\times$, and $2.34\times$ improvement, respectively. Under read-intensive workloads (B, C, and D), HAKV achieves throughput that is $2\times$ (minimum) to $3\times$ (maximum) higher than the other KV stores. This is because most of the hot read data is cached in PM, resulting in a higher read cache hit rate, as mentioned in section 4.5.2. For workload A (50% mixed writes), HAKV's performance improvement over the other KV stores is slightly lower compared with the read-intensive workloads but still shows a performance increase of $1.75\times$ (minimum) to $1.87\times$ (maximum).

Workload F consists of 50% read-modify-write operations, where all KV stores present performance degradation. However, HAKV maintains approximately twice the throughput of the other KV stores, thanks to its hot and cold data separation strategy and PM dynamic adaptive GC mechanism. Under workload E, HAKV's performance is slightly lower than LevelDB but comparable to Wisckey and UniKV. Under workload E, scan operations are suited for value-ordered storage structures. NovelLSM achieves the best scan performance in this scenario. Although HAKV employs multi-threaded parallel reads to optimize scan performance, it slightly underperforms NovelLSM under this workload due to its unordered storage management.

As shown in Figure 13, HAKV outperforms the other KV stores in terms of tail latency, especially under most workloads except for workload E. Compared with LevelDB, RocksDB, NovelLSM, LightKV, Wisckey, and UniKV, HAKV exhibits significant reductions in average tail latency, with

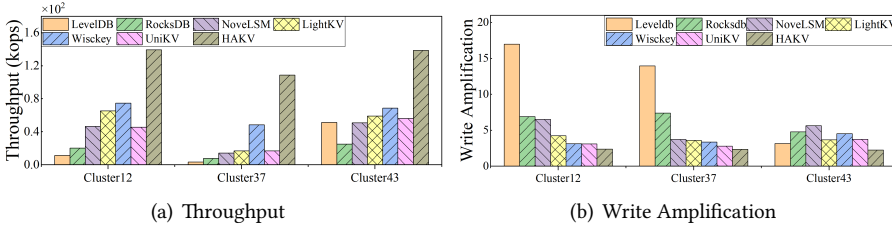


Fig. 14. Throughput and write amplification of HAKV under Twitter traces.

average percentage reductions of 65%, 95%, 76%, 55%, 58%, and 58%, respectively. Regarding P90 tail latency, the average percentage reductions are 83%, 92%, 78%, 25%, 66%, and 68%, respectively. For P99 tail latency, the average percentage reduction is 80%, 90%, 72%, 56%, 64%, and 53%, respectively, compared with the same alternative KV stores. The average percentage reduction represents the improvement in tail latency across all workloads. These performance improvements are attributed to HAKV's efficient separation and management of hot and cold KV pairs and the optimized use of PM space and the dynamic GC mechanism.

HAKV achieves high-performance enhancements in mixed read/write and read-intensive workloads. This is attributed to its effective hot and cold KV pairs separation and optimized cache management for PM storage. Experimental results demonstrate HAKV's superiority in large-scale transaction processing, delivering high throughput under diverse workloads.

4.4 Performance of HAKV under Production Workloads

To evaluate the performance of HAKV, we incorporate production workloads, Twitter traces[38]–Cluster12, Cluster37, and Cluster43. The experiment involves running 50 GB of KV pairs. As shown in Figure 14(a), HAKV achieves an average throughput improvement of 11.21 $\times$, 5.90 $\times$, 1.99 $\times$, 1.13 $\times$, 0.86 $\times$, and 2.06 $\times$ compared with LevelDB, RocksDB, NoveLSM, LightKV, Wisckey, and UniKV, respectively, under write-intensive workload Cluster12. Under read-intensive Cluster37, HAKV improves throughput by 31.01 $\times$, 13.10 $\times$, 6.66 $\times$, 5.39 $\times$, 1.23 $\times$, 5.37 $\times$, respectively. When it comes to the Cluster43 case, HAKV enhances throughput by 1.7 $\times$, 4.5 $\times$, 1.73 $\times$, 1.34 $\times$, 1.02 $\times$, 1.47 $\times$, respectively. The largest improvement is observed under read-intensive workloads because of HAKV's efficient management of the PM zone pool and the dynamic adaptive reclamation policy in the zone pool.

In Figure 14(b), under Cluster12, HAKV reduces write amplification by 86%, 66%, 63%, 44%, 24%, and 23%, compared with LevelDB, RocksDB, NoveLSM, LightKV, Wisckey, and UniKV, respectively. Under Cluster37, HAKV reduces write amplification by 83%, 68%, 37%, 35%, 30%, and 16%, respectively. Under Cluster43, HAKV achieves the reductions of 29%, 53%, 60%, 39%, 50%, and 40%, respectively, compared with the same alternative schemes. These improvements are attributed to HAKV's separation strategy for hot and cold KV pairs and its dynamic reclamation policy of zones in PM.

As shown in Figure 15, HAKV demonstrates significant improvements across multiple tail latency metrics under Cluster12. Specifically, the average tail latency, P99 latency, and P99.9 latency are shortened by up to 92%, 98%, and 99%, respectively, while P90 latency is reduced by up to 88% compared with the alternative KV stores. In Cluster37, HAKV cuts back average tail latency, P99 latency, and P99.9 latency by 97%, 98%, and 99%, respectively, and lowers P90 latency by up to 98%. P99 and P99.9 latencies of Cluster43 are reduced by up to 83% and 94%, with significant reductions also observed in average and P90 latencies. HAKV outperforms the other alternative key-value store systems under real-world production workloads: this advantage stems from its separation of hot and cold data, efficient zone management, and the dynamic GC policy that maintains hot zones in PM through an adaptive GC window.

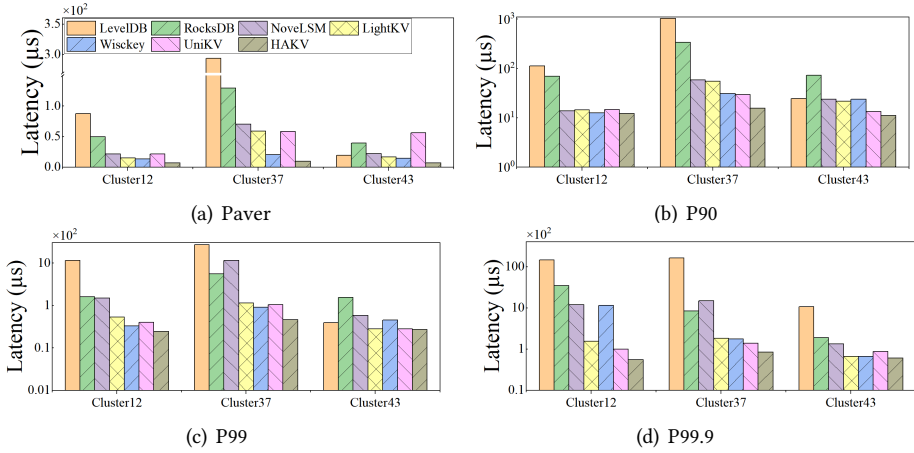


Fig. 15. Latency of HAKV under Twitter traces.

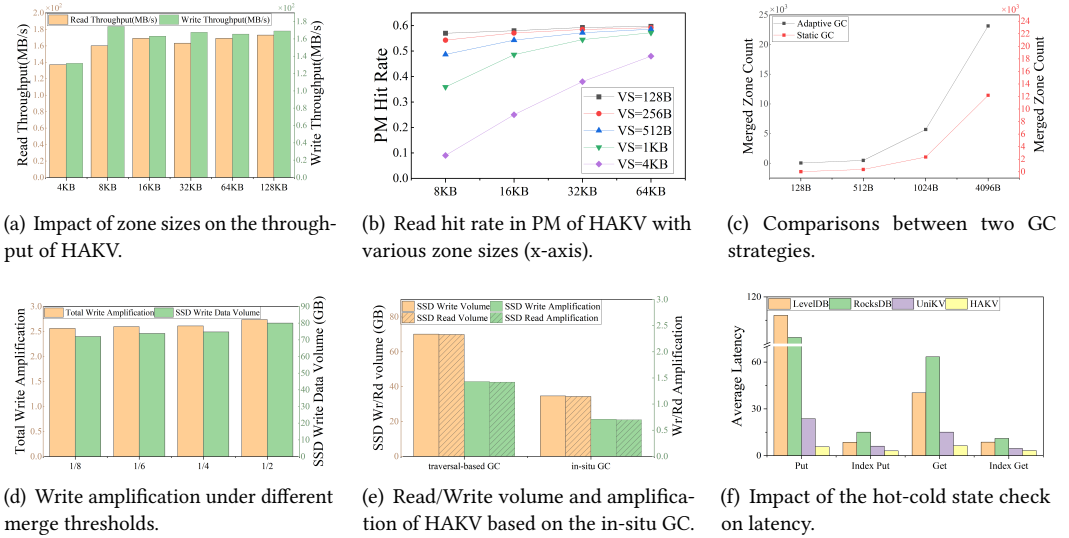


Fig. 16. Impact of the parameter settings on the performance of HAKV. Symbol VS denotes the value size.

4.5 Impact of HAKV Parameter Settings

4.5.1 Impact of Zone Size. We study the impact of zone sizes on HAKV's throughput. As shown in Figure 16(a), increasing the zone size from 4 KB to 128 KB improves the throughput, with the most significant gains occurring between the 32 KB and 64 KB range. This is attributed to appropriately sized zones, in which HAKV handles more KV pairs per I/O operation, reducing the frequency of requests and improving the throughput. However, when zone sizes exceed 64 KB, throughput gains become smooth, diminishing gains with further increases in zone size. This plateau indicates that while moderate zone size increments enhance the throughput, large-sized zones provide minimal additional benefits. This limitation can be attributed to caching constraints and the saturation of I/O parallelism, which prevents HAKV from fully utilizing larger zones beyond a threshold.

4.5.2 Read Hit Rate in PM. As shown in Figure 16(b), the read hit rate in PM exhibits an overall upward trend with increasing zone sizes. This trend remains consistent across all value size settings, with larger value sizes (e.g., 4KB) showing more pronounced improvements. In contrast, for smaller value sizes (e.g., 128B and 256B), the PM hit rate saturates at smaller zone sizes, and the improvement diminishes as the zone size increases. These results highlight significant variations in PM hit rate across different combinations of value size and zone size. This is because the value size and zone size impact PM cache utilization. Data blocks are compact and frequent-access for smaller value sizes, enabling a high read hit rate in PM even with smaller zone sizes. As the zone size increases, HAKV can cache more zones in PM, leading to an overall improvement in the access hit rate. In contrast, for larger value sizes (e.g., 1 KB and 4 KB), each zone consumes more space, necessitating a larger-sized zone to fully leverage the PM caching, thereby enhancing the read hit rate in PM.

Experimental results indicate that in PM-based KV stores, the zone size should be selected based on the value sizes to achieve a higher read hit rate in PM. For smaller value sizes, a smaller-sized zone suffices to maintain a high read hit rate in PM, whereas for larger value sizes, increasing the zone size can enhance the read hit rate in PM and improve overall system performance. Consequently, in practical applications, the zone size is dynamically adjusted according to the data distribution and access patterns to ensure utilization of PM and optimize read/write performance.

4.5.3 Comparison between Adaptive and Static GC Strategies. To assess the improvement of dynamic GC on HAKV's performance, we perform an experiment between dynamic GC and static GC under DB_Bench. In this study, the static GC employs a fixed-sized window (defaulting to 20% of the active PM queue length), and an 8 KB zone. The dynamic GC employs an adjustable window size from 5% to 30% of the active PM queue length. The experiment includes 100-GB random writes, followed by 100-GB reads, where we collect the number of merged small-sized zones (zones with less than 1/4 of valid data that require GC).

Figure 16(c) shows the number of zones requiring GC increases with larger value sizes. As value size grows, a small number of values can reduce the size of a zone, thereby triggering a GC operation. Furthermore, the number of small zones is higher in the dynamic GC process compared with a static GC. Then, we find that the dynamic GC, by adjusting the window size, reduces redundant GC operations, enhancing system performance. Experimental results demonstrate that HAKV with the dynamic GC strategy improves the system performance in scenarios with large-sized values.

4.5.4 Write Amplification (WA) of HAKV with the Merge Thresholds. To evaluate the impact of the merge thresholds on WA, we configure HAKV with various thresholds during zone reclamation in PM. When the valid data percentage of a zone falls below the threshold, HAKV migrates the remaining valid KV pairs to a new zone. After setting the threshold values to 1/8, 1/6, 1/4, and 1/2, we perform 100 GB of random writes followed by 100 GB of random reads using DB_Bench with a 16-B key and 1 KB value. Total WA of HAKV is the ratio of the total data written to both the SSD and PM to the volume of user write data. Figure 16(d) unveils that the total WA of HAKV and SSD write data volume increase as the threshold rises because higher thresholds relocate more valid KV pairs in PM, flushing more zones to the SSD and leading to higher SSD write data volume. When the threshold increases from 1/8 to 1/2, HAKV total WA climbs from 2.56 to 2.73 – indicating that the merge threshold marginally affects WA.

4.5.5 The Read/Write Advantage of In-Situ GC. In this section, we evaluate the write amplification of HAKV using two GC strategies for zone files: (1) in-situ GC based on a sparse bitmap mechanism and (2) traversal-based GC using a normal bitmap (one bit corresponding to a valid or invalid data block). We configure DB_Bench to perform 50 GB random writes followed by 50 GB random reads. For the traversal-based GC, a greedy strategy is used to select zone files with the lowest

proportion of valid data for reclamation. In Figure 16(e), with the sparse bitmap-based in-situ GC, HAKV achieves fewer file read/write operations and lower read/write amplification compared with the bitmap traversal GC method. Compared with the traversal-based GC, HAKV with the in-situ GC reduces the write amplification by 50.5% and the read amplification by 50.8%. This is because the segmented sparse bitmap avoids relocating large amounts of valid KV pairs, which is must be conducted in HAKV with the traversal-based GC. As a result, the in-situ GC uses the segmented sparse bitmap to improve GC efficiency with lower file read/write amplification.

4.5.6 Impact of the Hot-Cold State Check on Latency. We conduct experiments to gauge the impact of the index state check on latency in HAKV. Compared with LevelDB, RocksDB, UniKV—using a traditional MemTable structure—HAKV has a two-level index to distinguish the hot or cold state of KV pairs and configure a DRAM zone to buffer KV pairs. We measured average read/write latency and memory access latency, compared HAKV with LevelDB, RocksDB, UniKV, respectively. For LevelDB, RocksDB, and UniKV, the average memory access latency includes write data to the MemTable or read data from the MemTable. For HAKV, average memory access latency includes the hot/cold check overhead and write to (or read from) the DRAM zone. In Figure 16(f), experimental results show that HAKV achieves significantly lower write and read latency than LevelDB, RocksDB and UniKV. Specifically, LevelDB’s memory access latency is $2.81\times$ higher for writes and $2.67\times$ higher for reads than HAKV’s DRAM zone access latency. RocksDB’s memory access latency is $4.94\times$ higher for writes and $3.4\times$ higher for reads than HAKV’s DRAM zone access latency. UniKV’s memory access latency is $1.98\times$ higher for writes and $1.41\times$ higher for reads than HAKV’s DRAM zone access latency. These results demonstrate that the overhead introduced by hot/cold checks in HAKV is negligible for access latency.

5 Related Work

Optimizing KV Stores with Non-Volatile Memory. In traditional LSM-tree structures, frequent disk writes result in significant write amplification, which degrades overall performance. RangeKV [41] addresses this issue by introducing RangeTab in NVM to manage data in level L0. SLM-DB [15] reduces WA by organizing SSTables using a B⁺-tree index and a single-level structure. Similarly, LightKV [11] improves read and write efficiency by storing KV pairs as segments in PM with indexing, although file merging and compression on SSD remain as performance bottlenecks. Chen et al. [34] optimize query performance and reduces write amplification by redesigning secondary indexes in LSM storage systems with PM. Miodb [9] proposes byte-addressable, persistent skip lists to replace the on-disk data structures of LSM-trees, effectively mitigating write amplification. HPDK [24] enhances performance by employing PM as an SSD alternative and maintaining B⁺-tree indexes in DRAM; however, frequent merging operations increase storage costs.

In contrast, HAKV minimizes merge operations by pre-separating frequently updated KV pairs across storage zones and restricting PM recycling to zones with minimal updates. This approach reduces write amplification and storage costs while improving write and read performance by concentrating recycling efforts on frequently updated data, optimizing resource allocation, and enhancing operational efficiency.

Optimization of LSM-tree structure. PebblesDB [29] reduces write amplification by employing a sharded LSM-tree structure, which avoids rewrites at the same level. SlimDB [30] improves write performance through a semi-ordered storage model minimizing low-level data merging and sorting. Wiskey [25] introduces KV separation to mitigate write amplification caused by large values. HashKV [4] enhances access efficiency by using hash partitioning and a lightweight merging strategy, while UniKV [43] incorporates a hash index at level L0, significantly reducing both write amplification and read latency. Dostoevsky employs an adaptive hot/cold data merge strategy

to reduce unnecessary merging, thereby improving write performance. Additionally, KVell [19] and HiLSM [22] optimize I/O concurrency control and implement multi-level cache management to enhance overall LSM-tree performance. HAKV minimizes write amplification and optimizes overall performance by pre-separating keys based on update frequency and adopting a lightweight merging approach.

Optimizations for NVM-based index. HiKV [36] stores hot data in NVM using a two-level index to accelerate the overall performance. DPTree [45] introduces a new persistent index—the **differential persistent tree (DPTree)**—to minimize expensive PM writes. Nam et al. [26] present cacheline-conscious extendible hashing, which reduces the overhead of dynamic memory block management while ensuring constant hash table lookup time. Chen et al. [6] propose a lock-free concurrent hierarchical hash structure to improve concurrent access efficiency and reduce lock contention. PACTree [17] uses a high-performance persistent range index based on the PAC criterion to enhance the efficiency of range queries. HybridKV [8] combines a Radix-Tree and B⁺-tree in its index, leveraging NVM to address the read-write amplification problem. HAKV employs a two-level index based on zones to enable fast KV pair location and improve performance.

6 Conclusion

Traditional LSM-trees suffer from substantial write and read amplification due to their hierarchical ordered structure. Key indexing mechanisms, such as KV separation, also encounter challenges related to value log file recovery overhead. HAKV alleviates these limitations by separating frequently-updated (hot) KV pairs from read-only (cold) KV pairs based on update patterns in workloads. Then, HAKV employs a lightweight merge process to minimize write amplification. Furthermore, we devised the zone management scheme in PM to enhance the performance of HAKV, in which in-place zone reclamation is enabled. We implemented the highly efficient HAKV anchored on LevelDB, and the empirical study demonstrates that HAKV offers significant write amplification reduction, substantial performance improvement, and shortened tail latency. HAKV's zone management approach is readily adaptable to SSDs and other zoned storage devices.

References

- [1] Luis Barba and Pat Morin. 2014. Top-down skiplists. arXiv:1407.7917. Retrieved from <https://arxiv.org/abs/1407.7917>
- [2] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, modeling, and benchmarking {RocksDB}{Key-Value} workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223.
- [3] 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.
- [4] Helen HW Chan, Chieh-Jan Mike Liang, Yongkun Li, Wenjia He, Patrick PC Lee, Lianjie Zhu, Yaozu Dong, Yinlong Xu, Yu Xu, Jin Jiang, et al. 2018. {HashKV}: Enabling efficient updates in {KV} storage via hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 1007–1019.
- [5] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proc. Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1077–1091.
- [6] Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo. 2020. Lock-free concurrent level hashing for persistent memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 799–812.
- [7] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. 2020. {SplinterDB}: Closing the bandwidth gap for {NVM}{Key-Value} stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 49–63.
- [8] Chen Ding, Jiguang Wan, and Rui Yan. 2021. Hybridkv: An efficient key-value store with hybridtree index structure based on non-volatile memory. In *Journal of Physics: Conference Series*, Vol. 2025. IOP Publishing, 012093.
- [9] Zhuohui Duan, Jiabo Yao, Haikun Liu, Xiaofei Liao, Hai Jin, and Yu Zhang. 2023. Revisiting log-structured merging for kv stores in hybrid memory systems. In *Proc. 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 674–687.

- [10] Facebook. 2019. Rocksdb, a persistent key-value store for fast storage environments. Retrieved January 15, 2025 from <http://rocksdb.org/>
- [11] Shukai Han, Dejun Jiang, and Jin Xiong. 2020. LightKV: A cross media key value store with persistent memory to cut long tail latency. In *Proc. 36th International Conference on Massive Storage Systems and Technology (MSST'20)*.
- [12] Tyler Harter, Dhruva Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. Analysis of {HDFS} under {HBase}: A facebook messages case study. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*. 199–212.
- [13] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Blyan. 2018. Closing the performance gap between volatile and persistent {Key-value} stores using {cross-referencing} logs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 967–979.
- [14] IDC. 2023. Idc innovators: Privacy-preserving computation. Retrieved January 15, 2025 from <https://www.idc.com/getdoc.jsp?containerId=prCHC51469323>
- [15] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-ri Choi. 2019. {SLM-DB}:-{Single-level}{key-value} store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 191–205.
- [16] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning {LSMs} for nonvolatile memory with {NovelSM}. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 993–1005.
- [17] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. PACTree: A high performance persistent range index using PAC guidelines. In *Proc. ACM SIGOPS 28th Symposium on Operating Systems Principles*. 424–439.
- [18] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.
- [19] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. Kvell: The design and implementation of a fast persistent key-value store. In *Proc. 27th ACM Symposium on Operating Systems Principles*. 447–461.
- [20] Google. 2017. LevelDB, LevelDB is a fast key-value storage. Retrieved January 15, 2025 from <https://github.com/google/leveldb>
- [21] Jiazheng Zhang and Shouwei Li. 2023. Blockchain-Based Prefix-Hash and Improved Red-Black Tree Index Mechanism for Data Retrieving in Industry 4.0. Retrieved January 15, 2025 from <http://dx.doi.org/10.2139/ssrn.4655411>
- [22] Wenjie Li, Dejun Jiang, Jin Xiong, and Yungang Bao. 2020. HiLSM: An LSM-based key-value store for hybrid NVM-SSD storage systems. In *Proc. 17th ACM International Conference on Computing Frontiers*. 208–216.
- [23] Yongkun Li, Zhen Liu, Patrick PC Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. 2021. Differentiated {key-value} storage management for balanced {I/O} performance. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 673–687.
- [24] Bihui Liu, Zhenyu Ye, Qiao Hu, Yupeng Hu, Yuchong Hu, Yang Xu, and Keqin Li. 2024. HPDK: A hybrid PM-DRAM key-value store for high I/O throughput. *IEEE Transactions on Computers* 73, 6 (2024), 1575–1587.
- [25] Lanyue Lu, Thanumalayan Sankaranarayanan Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Wiskey: Separating keys from values in ssd-conscious storage. *ACM Transactions On Storage (TOS)* 13, 1 (2017), 1–28.
- [26] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H. Noh, and Beomseok Nam. 2019. {Write-Optimized} dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 31–44.
- [27] Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.
- [28] William Pugh. 1990. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (1990), 668–676.
- [29] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proc. 26th Symposium on Operating Systems Principles*. 497–514.
- [30] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A space-efficient key-value storage engine for semi-sorted data. *Proc. VLDB Endowment* 10, 13 (2017), 2037–2048.
- [31] 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.
- [32] 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.
- [33] Jing Wang, Youyou Lu, Qing Wang, Minhui Xie, Keji Huang, and Jiwu Shu. 2022. Pacman: An efficient compaction approach for {log-structured}{Key-Value} store on persistent memory. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 773–788.
- [34] Jing Wang, Youyou Lu, Qing Wang, Yuhao Zhang, and Jiwu Shu. 2023. Revisiting secondary indexing in {LSM-based} storage systems with persistent memory. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 817–832.

- [35] Xiaoliang Wang, Peiquan Jin, Bei Hua, Hai Long, and Wei Huang. 2022. Reducing write amplification of lsm-tree with block-grained compaction. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 3119–3131.
- [36] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. {HiKV}: A hybrid index {key-value} store for {DRAM-NVM} memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 349–362.
- [37] Yahoo!. 2019. YCSB-C, a C++ version of YCSB. Retrieved January 15, 2025 from <https://github.com/brianfrankcooper/YCSB>
- [38] Juncheng Yang, Yao Yue, and KV Rashmi. 2021. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. *ACM Transactions on Storage (TOS)* 17, 3 (2021), 1–35.
- [39] Ting Yao, Jiguang Wan, Ping Huang, Yiwen Zhang, Changsheng Xie, and Xubin He. 2023. GearDB: A GC-free key-value store on HM-SMR drives with gear compaction. In *Proc. ACM Turing Award Celebration Conference-China 2023*. 51–52.
- [40] 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 matrix container in NVM. In *2020 USENIX Annual Technical Conference*. 17–31.
- [41] Ling Zhan, Kai Lu, Zhilong Cheng, and Jiguang Wan. 2020. RangeKV: An efficient key-value store based on hybrid DRAM-NVM-SSD storage structure. *IEEE Access* 8, 0 (2020), 154518–154529.
- [42] Baoquan Zhang and David HC Du. 2021. Nvlsm: A persistent memory key-value store using log-structured merge tree with accumulative compaction. *ACM Transactions on Storage (TOS)* 17, 3 (2021), 1–26.
- [43] Qiang Zhang, Yongkun Li, Patrick PC Lee, Yinlong Xu, Qiu Cui, and Liu Tang. 2020. UniKV: Toward high-performance and scalable KV storage in mixed workloads via unified indexing. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 313–324.
- [44] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. 2021. ChameleonDB: A key-value store for optane persistent memory. In *Proc. Sixteenth European Conference on Computer Systems*. 194–209.
- [45] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. 2019. DPTree: Differential indexing for persistent memory. *Proc. VLDB Endowment* 13, 4 (2019), 421–434.
- [46] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. {Write-optimized} and {high-performance} hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 461–476.

Received 18 February 2025; revised 22 June 2025; accepted 30 June 2025