TrieKV: A High-Performance Key-Value Store Design With Memory as Its First-Class Citizen

Hui Sun, Deyan Kong, Song Jiang, *Senior Member, IEEE*, Yinliang Yue, and Xiao Qin, *Senior Member, IEEE*

***Abstract*—Key-value (KV) stores based on log-structured merge tree (LSM-tree) have been extensively studied and deployed in major information technology infrastructures. Because this type of systems is catered for KV store accessing disks, a limited disk bandwidth increases the difficulty of serving online data requests. One solution involves using a large DRAM such that frequent KV pairs are buffered and accessed from the main memory – and this solution exposes a major design drawback of the KV store: its lack of support for integrated data management in memory and on disks. For example, data in the most popular LSM-tree implementation – RocksDB – may reside in a small write buffer (MemTable) that organizes KV pairs for disk writes, a buffer cache for disk blocks, a write-ahead log on the disk for data persistence, and in various LSM levels on the disk. Without the integrated management of indexes, data, and their persistence in a hierarchical memory/disk architecture, memory is under-utilized along with missed performance optimization opportunities. We propose a KV store, TrieKV, which holistically incorporates DRAM, persistent memory (PMem), and disk with certain desired features: (1) fast in-memory access, (2) accurate identification of hot/cold data at an adaptable granularity, (3) customized memory space allocation for minimized fragmentation, (4) hotness-aware data placement across the storage hierarchy, (5) in-place data persistence in the PMem, and (6) hotness-aware LSM-tree compaction. TrieKV employs a single, integrated trie-structured index for all KV pairs in memory, where access hotness can be consistently discovered. Accordingly, the KV placement is dynamically determined according to the hotness and persistence needs of the storage hierarchy spanning the DRAM, PMem, and solid-state drive. In the experiment, we demonstrate that the 99th latency of RocksDB and NoveLSM is 38x and 6x higher than that of TrieKV, respectively. In addition, TrieKV outperforms RocksDB and NoveLSM by a factor of 5.6 and 1.7in terms of throughput, respectively.**

***Index Terms*—Hybrid memory, key-value (KV) store, log-structured merge tree (LSM-tree), persistent memory.**

Received 9 December 2023; revised 26 July 2024; accepted 29 September 2024. Date of publication 2 October 2024; date of current version 25 October 2024. The work of Xiao Qin's was supported by the U.S. National Science Foundation under Grant IIS-1618669 and Grant OAC-1642133, in part by National Aeronautics and Space Administration under Grant 80NSSC20M0044, in part by National Highway Traffic Safety Administration under Grant 451861-19158, and in part by Wright Media, LLC under Grant 240250 and Grant 240311. This work was supported in part by the National Natural Science Foundation of China under Grant 62472002 and Grant 62072001. Recommended for acceptance by J. Lofstead. *(Corresponding author: Yinliang Yue.)*

Hui Sun and Deyan Kong were with the School of Science and Technology, Anhui University, Hefei, Anhui 230601, China (e-mail: sunhui@ahu.edu.cn; e21201086@stu.ahu.edu.cn).

Song Jiang is with the Department of Computer Science and Engineering, University of Texas, Arlington, TX 76019 USA (e-mail: song.jiang@uta.edu).

Yinliang Yue is with the Third Department, Zhongguancun Laboratory, Beijing 100094, China (e-mail: yylhust@qq.com).

Xiao Qin is with the Department of Computer Science and Software Engineering, Auburn University, Auburn, AL 36849-5347 USA (e-mail: xqin@auburn.edu).

Digital Object Identifier 10.1109/TPDS.2024.3473013

I. Introduction

KEY-VALUE (KV) stores are becoming increasingly important in the era of Big Data, leading to their widespread deployment. Most KV stores leverage a log-structured merge tree (LSM-tree) [1] to achieve a high write performance accompanied by well-sorted KV pairs. The examples of LSM-tree-based KV stores include LevelDB [2], RocksDB [3], BigTable [4], and Cassandra [5]. To sort the KV pairs incrementally, LSM-trees store KV pairs at multiple levels according to their insertion timing – and expensive compaction operations are constantly conducted across these levels. Although numerous approaches were proposed to reduce the compaction cost and improve read/write performance [6], [7], [8], [9], [10], the performance of KV stores is ultimately constrained by the slow disk I/O performance, impeding data access for online requests.

An efficient solution is to augment the memory capacity of the system, which increases the number of KV pairs being accessed from memory. Because KV pairs are stored in files, file blocks containing the KV pairs are cached in the memory buffer managed by a file system or store itself [11]. Accordingly, a large memory would accommodate a large buffer cache; thus, more KV pairs can be kept in the cache to push the hit ratio high. In this manner, memory is treated as an add-on to the KV store system, which is traditionally designed as a storage system on block devices like hard disks and solid-state drives (SSDs): there is no serious performance issue because the memory available to the store has a relatively small capacity – and most access requests are served by the disk. Nonetheless, in-memory performance becomes increasingly crucial when the store handles requests from online services. Although the limited disk speed limits the speed of accessing data from the disk, optimizing the memory utilization is critical to matching the performance needs of KV applications.

Without prioritizing memory in the design of KV stores, existing KV stores miss important performance optimization opportunities – and precious memory resources are underutilized for the following three reasons. First, when KV pairs are stored at different granularities and are distributed in different management domains, including the read buffer cache, write buffer (e.g., MemTables in LevelDB and RocksDB), and LSM-tree on the disk, their hotness cannot be evaluated consistently to identify the globally hot KV pairs and retain them in the cache. For example, in RocksDB, the buffer cache runs its least recently used replacement policy to identify hot and cold data in its allotted memory space. These are individually indexed to the write buffer. They are loaded into and managed in the read buffer cache

1045-9219 © 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Auburn University. Downloaded on March 14,2025 at 20:35:39 UTC from IEEE Xplore. Restrictions apply.

of the disk block. When KV pairs are read randomly, the blocks in the read buffer cache contain both hot and cold KV pairs. This results in low memory utilization because of the caching of fixed-sized blocks. Second, due to the small and variable size of KV pairs, frequent memory allocation operations for storing individual KV pairs are prone to memory fragmentation: this problem leads to a reduced allocation speed and memory inefficiency. Third, a fundamental conflict is observed between performance and persistence when only DRAM memory and disk are considered. High-speed DRAM is volatile, meaning newly admitted KV pairs to the memory cannot be quickly persisted in place. The common practice of using a write-ahead log (WAL) to maintain KV pairs on a disk incurs additional, expensive disk operations and redundant data.

The key to addressing the above issues is a holistic design that organizes all in-memory KV pairs into a single high-performance index for individual KV pairs. With this standard index, data hotness can be consistently monitored and gauged at an adaptable size that is as small as one KV pair. When persistent memory (PMem) is leveraged, the hotness and persistence requirements of the KV pairs guide their placement in the DRAM, PMem, or disks in a streamlined arrangement. The growing importance of memory in modern KV services and inefficient use of memory in the existing KV system designs are well understood, and efforts have been made to address these concerns. For example, some studies proposed memory-allocation strategies that are adaptive to the workload characteristics between different memory components [12], [13], [14].

In addition, studies – introducing PMem into KV storage systems – are focused on data management within new devices and selecting KV pairs to be stored [15], [16], [17]. For example, NoveLSM [15] utilizes both mutable and persistent MemTable in the PMem to increase the buffer size in DRAM. The KV pairs of immutable MemTable inside DRAM are first flushed into the large-sized MemTable in the PMem one by one. In the PMem, the write size are smaller than 256 B, which causes high write amplification (WA). In LightKV [18], a global index radix hash tree (RH-Tree) that consists of an upper-level radix tree in DRAM and hash table-based leaf nodes in the PMem is proposed to index KV pairs in the PMem. The key-range partition of RH-Tree is used to group KV pairs with the same key prefixes into SSTables in the same partition. However, the unbalanced key distribution in data-skew workloads can cause some partitions to be overloaded. Then, the query performance is degraded when many SSTables with overlapping key ranges exist in these partitions. These techniques are inadequate as they are unable to maintain all in-memory KV pairs within one common management structure.

We propose TrieKV as an LSM-based PMem-enabled KV store that addresses the aforementioned limitation. We first developed a trie-tree index for all in-memory KV pairs across the DRAM and PMem to unify their management in diverse medium memory spaces and for various usage purposes. The proposed index structure curtails the memory access latency while improving the read throughput. TrieKV updates the indexes in DRAM, which doesnt cause WA. As the size of KV pairs buffered in DRAM is larger than the write unit of PMem, the KV pairs are written into the PMem, avoiding small writes in the PMem. A major challenge is to identify hot/cold-access regions in the integrated index. Tracking individual KV pairs is expensive and often unnecessary. Moreover, consistently hot regions exhibit variations in size. Accordingly, TrieKV uses a two-level hotness identification approach (for coarse- and fine-level hotness) to accurately determine hot and cold regions for placement in different devices. We design a hotness-region-aware compaction strategy that selects read-intensive hot-key ranges for prioritized compaction in high-performance point and range queries.

The contributions of this study are tabulated as follows:

- We propose the trie-tree index to organize KV pairs in DRAM and Intel Optane PMem. Because of the diverse performance characteristics of the DRAM and PMem, we design two dynamic memory allocation methods for the index and KV pairs. By searching the common trie index for the in-memory reads and writes of the KV pairs, TrieKV achieves a high in-memory performance for KV pairs.
- We put forth a variable-granularity hot-region identification approach for coarse and fine granularity to accurately classify hot regions whose sizes are consistent with the actual hot-key distribution.
- We optimize the compaction performance by leveraging the detected distribution of the key hotness regions to minimize I/O costs and improve overall read performance in systems.
- We implement TrieKV by modifying the code base of HyperLevelDB [19] and conducted extensive experiments to evaluate TrieKV by comparing its performance with state-of-the-art KV stores such as NoveLSM. The results show that TrieKV bolsters the performance of HyperLevelDB, RocksDB, NoveLSM, and LightKV by a factor of 2.8, 2.9, 1.8, and 0.6, respectively. Compared with NoveLSM, TrieKV reduces write amplification (WA) by a factor of 3.2 and decreases tail latency by a factor of 6.1.

The remainder of this paper is organized as follows. Section II presents the background and motivation of this study. In Section III, we describe the design and implementation of TrieKV. We describe the experimental platform environment in Section IV. In Section V, the devised system is evaluated. Section VI elaborates on the related studies. We provide the discussion in Section VII. Finally, we conclude the paper in Section VIII.

II. Background and Motivation

This section provides relevant background pertaining to LSM-tree-based KV stores and the application of PMem. We also shed lights on the implications of a large memory.

A. LSM-Tree

LSM-tree-based key-value store systems or KV stores postpone write requests and batch them in memory to take advantage of the high sequential write bandwidth in disks. RocksDB – a widely deployed implementation of KV store [3] – comprises of three primary components: a WAL that resides temporarily on a disk to protect data in DRAM from being lost during system

Authorized licensed use limited to: Auburn University. Downloaded on March 14,2025 at 20:35:39 UTC from IEEE Xplore. Restrictions apply.

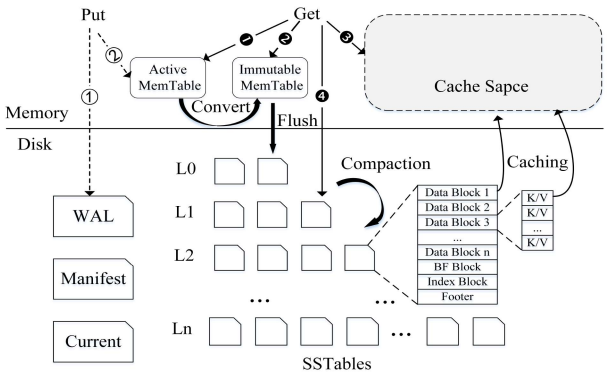


Fig. 1. Overview of RocksDB. ① Write-ahead logging. ② Active MemTable index updating. ❶ Search in the active Memtable. ❷ Search in Immutable Memtable. ❸ Search in cache. ❹ Search in level L_0, L_1, L_2.



(a) Leveling Compaction Policy.

(b) Tiering Compaction Policy.

Fig. 2. LSM-tree compaction policies.

failures, a MemTable that is located in memory and acts as a write buffer that can be either sorted or unsorted, and sorted string tables (SSTables) that are kept on the disk.

More often than not, MemTable employs a skip lists index structure to organize KV pairs. Fig. 1 shows the structure and operations of RocksDB. Initially, a new KV pair is ① committed to the WAL and ② buffered in a MemTable. When the MemTable is full, it is converted into an immutable MemTable. The immutable MemTable is then flushed to level L_0 on the SSD to create SSTables. The LSM-tree on-disk structure consists of multiple levels of exponentially increasing sizes. A new SSTable is placed at a lower level like L_0. Multiple SSTables may exist at level L_0 with overlapping key ranges; however, the key-range overlapping is not allowed at other levels. KV pairs are continuously migrated from level L_0 to higher levels (L_1, L_2,... L_n) though compaction operations.

The LSM-tree structure adopts two major compaction policies [20] with different implications on WA and read performance: leveling and tiering. In the leveling compaction policy as demonstrated in Fig. 2(a), the components in two adjacent levels are merged – and new components are placed into the lower level. In the tiering compaction policy illustrated in Fig. 2(b), each level consists of T components with overlapping key ranges. When level L_i becomes full, a merge-sort operation will be conducted among the T components – and the resulting KV pairs will be written as a new component of level L_{i+1} . In general, the leveling compaction policy optimizes the query performance because the search in the LSM-tree considers fewer components; however, this policy suffers a high WA owing to its aggressive sort-merging policy. The tiering compaction policy is write-optimized because it reduces the compaction frequency. As the tiering compaction policy cannot effectively limit the number of overlapping tables, tiering leads to a much higher search cost than the leveling compaction policy.

With a large-sized memory, enormous new KV pairs can be sorted in the memory, potentially cutting down the compaction operations on the on-disk KV pairs. A large memory capacity enhances the high-speed insertion of KV pairs. Even so, a new complication emerges. Instead of constantly generating a fixed-size SSTable from a full MemTable, we have more in-memory KV pairs than the SSTable can accommodate. We ought to select a range of KV pairs from all new in-memory KV pairs or a partition in the key space when writing them to the disk.

KV-store workloads mostly exhibit data skewing [21], [22], where a portion of hot KV pairs have a higher probability of being accessed than cold keys. Prior studies [6], [23] attempted in-memory compaction, which merges hot records in memory in advance to maintain frequently updated KV pairs before repeatedly flushing them to disks: this strategy results in frequent compaction operations. Currently, hot KV pairs are identified at a fixed granularity – individual KV pairs, blocks, or MemTables – with the goal of pinpointing a sequence of KV pairs (a partition) of consistent hotness for placement in the storage hierarchy. Although accessing individual KV pairs or from the disk is inefficient, a large partition should be used. However, fixed-size partitions, such as in MemTable and blocks, are often inconsistent with the distribution of hot keys. A partition should be of variable size, aligning with the detected hot/cold KV distribution, and determining the partition demand for new solutions.

Applications that rely on KV stores often exhibit high spatial locality. A typical design pattern concatenates two or more attributes to form a composite primary key. For example, Flurry [24] utilizes various attributes (app ID, user ID, time, timestamp, etc.) to generate a key. Google's BigTable [4] generates keys by combining row name, column name, and timestamp. These keys are used for data storage within KV stores. Workloads typically manifest a skewed distribution. Consequently, when data are accessed using a composite key, spatial locality is normally demonstrated. The prefix characteristics of the workloads motivate us to use prefix trees and hot-region identification for the prefix trees.

The LSM-tree is not ideal for workloads with high spatial locality due to two reasons. First, owing to the leveled structure of the LSM-tree, hot-key ranges are fragmented across

Authorized licensed use limited to: Auburn University. Downloaded on March 14,2025 at 20:35:39 UTC from IEEE Xplore. Restrictions apply.

TABLE I
THE PERFORMANCE OF PMEM AND OPTANE SSD

	Sequential read	Sequential write	Random read	Random write
Intel Optane PMem [30]	40 GB/s	12 GB/s	30 GB/s	9 GB/s
Samsung PMem 1643A 960G [31]	2100 MB/s	1000 MB/s	380K IOPS (1484 MB/s)	40K IOPS (156 MB/s)
Intel Optane SSD P5800X [32]	7200 MB/s	6100 MB/s	1500K IOPS (4K Blocks) (5859 MB/s)	1350K IOPS (4K Blocks) (5273 MB/s)
Netlist NV4 NVDIMM-N 16GB [33]	1857 MB/s	1400 MB/s	449K IOPS (4K Blocks) (1754 MB/s)	382K IOPS (4K Blocks) (1492 MB/s)

multiple files. The read operation must traverse each level, causing severe read amplification. Second, the compaction procedure indiscriminates against key popularity. Because new files are always merged with old files, hot-key ranges continue to be repeatedly re-located by compaction. We put forth a KV-store design catering for spatial locality and a skewed workload without losing the benefits achieved by the LSM-tree approach.

B. Persistent Memory

PMem and non-volatile memory (NVM) provide byte-addressing capability. The Intel Optane DCPMM [25], a popular PMem product, can be used as a memory extension offering in-place data persistence. PMem – performing similarly to DRAM – is less expensive [26]: this feature allows users to expand memory capacity at an affordable price. A large-sized memory bolsters the performance of storage systems by accommodating large amounts of data. Although the Intel Optane DCPMM has several advantages, a performance gap remains between DRAM and PMem: the random write throughput of the Intel Optane DCPMM is lower than that of DRAM [25], [26], and the read latency of Optane PMem is approximately four times higher than that of DRAM. The existing studies [27], [28] demonstrate that Optane PMem has a write unit of 256 bytes; however, the size of the individual index nodes is small. Data points smaller than the write unit require three steps to be performed – read, modify, and write – increasing the WA of PMem and consuming the system's bandwidth.

Compared with SSDs, PMem offers a lower read/write latency and higher bandwidth – and PMem's performance is approximately two orders of magnitude higher than that of SSDs [25], [26]. As listed in Table I, we present the performance of persistent memory and SSDs used in our experimental platform. Other PMem products, such as Micron's QuantX and Samsung's Z-NAND [29] products, can provide high performance on par with that of Optane. Because KV stores necessitate data persistence – and the disk is slow and block-accessed, PMem emerges as a block-accessed alternative offering quick persistence and fine-grained data access.

III. TRIEKV DESIGN

We designed the TrieKV store to accommodate the diverse characteristics of DRAM, PMem, and SSDs. The key considerations in the method are listed as follows: (1) the placement of the index and KV pairs across the storage hierarchy; (2) the organization of KV pairs in heterogeneous devices; and (3) the decision of which and how KV pairs migrate between devices. The TrieKV fully exploits the high performance of DRAM, the persistence of PMem, and the large capacity of SSDs.

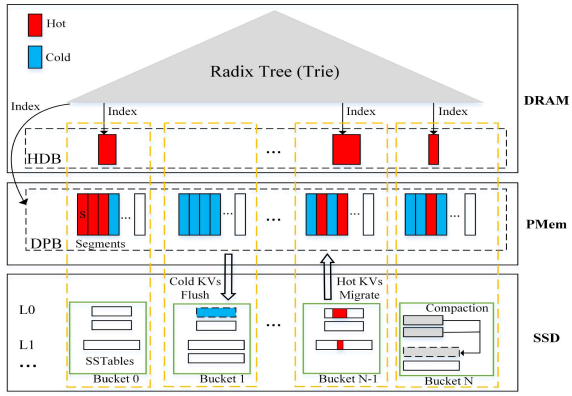


Fig. 3. Overview system of TrieKV.

A. System Architecture

Fig. 3 sketches the TrieKV architecture, where the key space is partitioned into a certain number of buckets. Each bucket admits new KV items directly from a buffer in PMem that corresponds to that bucket. The buffer receives new KV items with keys in the same range as those in the corresponding bucket. TrieKV manages two buffers for KV pairs: one is a hot data buffer (HDB) in the DRAM that caches frequently accessed KV pairs, and the second is a data persistence buffer (DPB) in the PMem immediately ensuring that newly written KV pairs persist. The DPB consists of multiple segments, and KV pairs with different buckets are written into different segments in an append-only manner. TrieKV places its index entirely in the DRAM to efficiently service every request. The index in the DRAM is a trie structure, in which leaf nodes contain pointers to the KV pairs in these buffers. When the DPB is full, KV pairs are migrated to the SSD. Notably, TrieKV only flushes cold keys to the disk and maintains hot keys in the PMem. Like the LSM-tree's flush process, we refer to the generated file as an SSTable. We utilize the LSM-tree structure within each bucket. Integrated with a write-efficient tiering compaction policy and a bucket layout with the key distribution adapted to the hot region, TrieKV balances low WA and fast searches.

Fig. 4 shows the workflow of TrieKV, where several components are incorporated to spot hot and cold regions in addition to the management of the on-disk data. *Hotness monitor.* This component accurately determines actual hot key distribution using a variable granularity hot-region identification approach on a trie tree, based on the spatial locality of the workload. *Bucket manager.* Bucket compaction is scheduled using the detected hot-key region distribution. TrieKV splits a disk bucket based on hot-key regions, thereby adapting the bucket to workload shifts in terms of key distribution. *Data migrator*. This component bears two responsibilities. First, to fetch hot KV pairs from SSD

Authorized licensed use limited to: Auburn University. Downloaded on March 14,2025 at 20:35:39 UTC from IEEE Xplore. Restrictions apply.

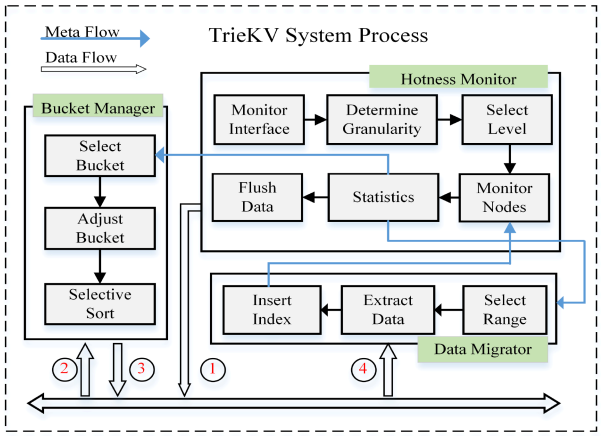


Fig. 4. Workflow of TrieKV.

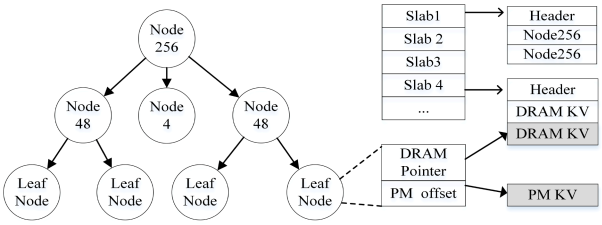


Fig. 5. An example of memory integrated management in ART.

to PMem, the data are deserialized and inserted into the trie-tree. Second, it is in charge of loading hot data from the PMem to the DRAM – and modifying the pointers to point to these data.

B. Memory-Integrated Management

Updates to persistent indexes typically involve multiple small random write operations that incur write amplification. Additionally, ensuring consistency of the index nodes in the PMem adds overhead. To balance the read/write efficiency and the need for data persistence, we place indexes in the DRAM, and the data is selectively placed in the PMem and DRAM according to the hotness.

Fig. 5 shows the structure of the memory-integrated management. The reconstruction of volatile index nodes by scanning data in PMem is sufficiently fast; therefore, volatile index nodes can be tolerated. Trie trees have good cache locality and support fast prefix queries. In this study, we employ an efficient trie-tree index variant – adaptive radix tree (ART) [34] – as the index. The memory pool is a slab-based memory allocation pool, which allocates memory space for the ART. The ART dynamically elects multiple compact inner nodes (e.g., Node4, Node48, Node256in Fig. 5) to reduce space occupation. The frequent replacement of internal indexes results in memory fragmentation. We apply a slab-based memory allocator [35] to mitigate memory fragmentation by dividing the memory into a series of 'slabs', each of which is a collection of fixed-size objects. The slab-based memory allocators can efficiently handle the allocation and reclamation of fixed-size data structures. In the slab-based memory allocator, when an object is freed, it is retained in the slab rather than returned to the operating system. When it needs to allocate a new object, it can be quickly reallocated from these existing slabs, which speeds up the procedure of allocation and reclamation. Since objects are often allocated from the same slab, they have close physical addresses in memory, which increases the cache hit rate and improves system performance. In the leaf node, we store DRAM pointers to KV pairs in the DRAM or PMem offsets of the KV pairs in the PMem. PMem stores multiple versions of values for a key, and the PMem offset in the leaf node stores only the offset of the latest value.

The performance of the PMem is two orders of magnitude higher than that of SSDs, see Section II-B. Therefore, our system writes KV pairs to the PMem to achieve rapid data persistence. Furthermore, the read latency of the DRAM is lower than that of the PMem [36], and space in DRAM is reserved for frequently accessed KV pairs and the DRAM pointer is set in the leaf node accordingly.

C. Data Persistent Buffer

The access granularity differs between the cache and PMem [37], and a good example is the access granularity discrepancy between the 64-B cache line and 256-B Optane PMem. Because small random writes tend to cause WA, we apply the append-write method to mitigate high WA – and improve the TrieKV performance. If we store KV pairs in the PMem in only an append-write manner, then the location of the KV pair in the storage log will be determined by write sequence. In realistic workloads [21], [38], the access pattern of requests is random write-intensive, and a random distribution may follow KV pairs within a key range. However, the storage location of KV pairs is determined by a write order, and KV pairs in the key range may be randomly distributed. This trend makes the reclamation and reuse of space overhead relatively high. Therefore, we consider in TrieKV the spatial locality of KV pairs when data are stored in PMem: TrieKV can group KV pairs based on the key range and reclaim the PMem space in batches, thereby alleviating the reclamation overhead.

We divide the persistent memory into fixed-size segments and allocated a segment to each bucket's PMem buffer. The KV pairs written to a key range are appended to an allocated PMem segment. If the segment space is insufficient, TrieKV applies another PMem segment to be linked to the previous one – and more PMem segments may be allocated for frequent-write key ranges. Nonetheless, for frequent-updated key ranges, all versions of a value corresponding to the same key are written to the same segment group. The old versions of this value consume the PMem space, but the frequently updated key ranges could be reclaimed to reduce the PMem usage. The cold KV pairs in the segment are flushed to the disk, and we extract hot KV pairs to be re-appended to a new segment followed by modifying pointers in the index.

D. Variable Granularity Hot-Region Recognition

As discussed in Section II-A, the hotness identification method that employs a small granularity (e.g., KV pairs) can

Authorized licensed use limited to: Auburn University. Downloaded on March 14,2025 at 20:35:39 UTC from IEEE Xplore. Restrictions apply.

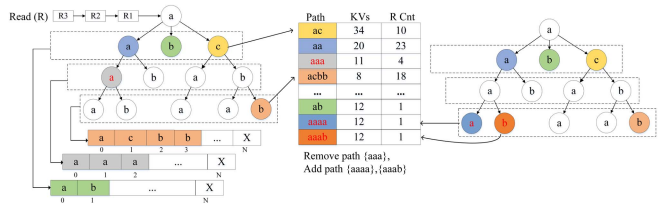


Fig. 6. Variable granularity hot-region recognition.

identify hot keys being frequently accessed. This scheme, however, cannot determine hot regions that embrace these hot keys. The hotness identification method that incorporates a large granularity (e.g., a block or MemTable) is often inconsistent with hot-region size. Unlike the KV pair, block, and MemTable granularity, variable-sized key ranges in a trie tree structure – serving as a data index – are utilized in our proposed hot-region identification scheme.

Fig. 6 sketches that three types of prefixes exist in the trie tree. The key length is fixed to N – and the longer the prefix of a fixed-size key, the smaller the key range it represents. The range of keys prefixed with *acbb* is defined as the small range of keys; the range of keys prefixed with *ab* as the large range of keys; and the range of keys prefixed with *aaa* as the medium range of keys. A simple approach involves identifying hot regions at all levels based on the trie tree, and selecting the required level according to workload characteristics. Nevertheless, it is evident that this strategy is unnecessary (Section II-A); applications of KV stores often exhibit high spatial locality. We advocate for capturing the spatial locality of workload – and accurately identifying hot-regions areas by combining coarse and fine recognition granularities, thereby reducing recognition overhead. In the trie tree, the coarse granularity at a high level (near a root node) is counted to determine key-range hotness. Similarly, the key-range hotness is gauged using a fine granularity at the low level (near leaf nodes). The recognition granularity is dynamically adjusted according to the given level.

In a variable-granularity hot-region identification scheme, it is non-trivial to determine whether the level is high or low. At the lowest level (e.g., leaf-node level), the hotness of a key range is translated into that of the KV pairs. This method achieves accurate hotness information at the cost of high identification overhead. Measuring hotness at a high level may result in inaccurate identification of data hotness. Notably, in this study, we employed YCSB in the experiment, which features the workload scan with spatial locality. The scan length is about 1-100 KV pairs; therefore, in TrieKV, we select two or three levels close to the leaf nodes, which in the adaptive radix tree can cover up to 10-100 KV pairs. It means that the hot-region that can be recognized matches the actual hot-region.

For the coarse-grained hot-region identification, since the keys in the real-world workloads are in the form of combined keys, these combined keys' public prefix determines the coarse-grained hot-region identification level. There are two examples in a production environment corroborating our choice of the first two or three layers for coarse-grained hotspot-region identification. In Google's BigTable [4], the database table is decomposed into KV pairs for storage in a distributed KV store. A key is composed of the row name, column name, and timestamp. In the social network of Facebook [39], a graph edge is indexed by a key consisting of two object ids and an association type. These random numbers are generated as keys in DB_Bench and YCSB. In the case of coarse-grained hot-region identification, we select the first few digits of the valid numbers as the public prefix in the combined keys for coarse-grained hot-region identification. The level in KV stores selected by the coarse-grained hot-region identification is determined by the generation method of the combined keys.

We implement two approaches to collecting hotness information. First, we store tracking hot information in index nodes and hotness information in each index node. Undoubtedly, this identification method adds space overhead to the index nodes, and a large amount of memory is wasted on data index path nodes that are never accessed. Second, we employ a centralized approach to monitor the hotness information of data. We lightly sample requests to curtail the overhead of hotness identification.

We can store the hotness information in DRAM or PMem. Notably, when storing the hotness information of data in the PMem, modifications to even a few bytes of this information can lead to high WA in the PMem. Thus, we store the hotness information in the DRAM instead of PMem. We utilize a list to glean and manage hotness measures. Index nodes at a level of the adaptive radix tree selected to monitor hotness have a pointer, which refers to an entry within the list containing the hotness information. The structure of this hotness information includes the number of data entries within the entire range as well as the number of accesses made to that range. The hotness information can be updated in DRAM, which does not cause WA.

Fig. 6 makes use of a concrete example to illustrate the hot-region recognition procedure. Let us consider searching for a key prefixed with *aaab*. When this key is accessed, the search starts from the root node of the trie. If the search reaches the prefix *aa*, it is discovered that the node has a pointer to the hot-region recognition list, and we modify the information regarding the list entry that it points to, such as the count in the key range. The search process continues for the trie, and when we reach the prefix *aaa*, we find another node pointing to a list entry and the previous step is repeatedly undertaken. Notably, at most two nodes that point to the hot-region identification list are encountered in the entire search path, which is the coarse and fine granularity of the aforementioned hot region identification. Indeed, the coarse-grained hot-region identification is initialized with the prefix *aa*. When a key range becomes hot, we add a fine-grained hot region identification to pinpoint the smallest hot region, (e.g., *aaa*). Suppose we initialize the fine-grained hot region identification at a high level (e.g., *aaa*): if this hot region is persistently hot, we adjust this fine-grained hot-region identification level downwards (e.g., *aaaa* and *aaab*) to determine the smallest hot-region.

All ranges are classified as hot or cold, and all hotness information ought to be sorted to keep track of data hotness. More precisely, we weigh the average number of accesses in a key range and the average number of accesses in a key range over

Authorized licensed use limited to: Auburn University. Downloaded on March 14,2025 at 20:35:39 UTC from IEEE Xplore. Restrictions apply.

the last period. Then, we have

$$H_{\text{i}} = \alpha \times \frac{C_{\text{hit}}}{C_{\text{cur}}} + (1-\alpha) \times H_{\text{i-1}}\,, \tag{1}$$

where H_i denotes the hotness in a key range over the current time, and C_{hit} represents the number of accesses to this key range. C_{cur} is the number of KV pairs in the key range. The average access hotness is obtained by dividing the number of accesses to the key range by the number of KV pairs in this range. Variable α is a decay factor that denotes the weight given to new hotness and how quickly to decay the old hotness. Let H_{i-1} be the hotness of the key range during the previous period. Then, we sort the key ranges based on their hotness, aiming to categorize data ranges into two families of hot and cold. We define a hot key range to include keys among the top 10% of the key ranges. In this case, the hottest part of the PMem data is copied into the DRAM based on the DRAM bucket to accelerate the data lookup, and data that is cooling down is discarded from the DRAM. When the PMem is under-budgeted, cold KV pairs are dumped onto the disk.

E. On-SSD Data Management

1) Dynamic Bucket Management: The key space in TrieKV is partitioned into multiple buckets, each of which is dedicated to a key range, and the SSTables are organized into these buckets. We apply the tiering compaction policy within the buckets to achieve a lower WA. TrieKV does not pre-allocate numerous buckets during the storage initialization phase; the spatial locality of the workload ensures that numerous buckets can be pre-allocated if it is required and the information is known in advance. In case that the storage contains a large amount of data, an excessive number of files will overlap in the buckets: this phenomenon degrades read performance. TrieKV sorts the KV pairs within the buckets to improve the read performance. The range covered by a bucket is typically larger than that in the hot region. To further mitigate high WA while bolstering read efficiency, we split the buckets – and treat the hot regions as separate buckets.

Selecting dividing points is critical to congregating hot regions in a bucket. For example, the data partitioning method in LightKV [18] is determined based on the range division of the leaf nodes in the RH-Tree. The generation of leaf nodes in the RH-Tree depends on the written order of KV pairs. The generated leaf node does not change, and the partitioning method relies on data's arrival order. The partitioned key space may be inconsistent with hot regions in workloads; compacting partition to improve read performance may result in duplicate writes of cold data, causing a serious write amplification issue combined with wasted I/O bandwidth. Therefore, we judiciously handle hot region distribution in the key space to determine a dividing point. We predetermine the number of new buckets that would be split from an old bucket. If hot regions are in a bucket, the bucket will be divided into new buckets in accordance with the splitting points of these hot regions; otherwise, the bucket is divided into new buckets evenly. Fig. 7 unveils that buckets *[52, 93)* must be split where a hot region *[62, 67)* exists, and the hot region



Fig. 7. Bucket management inside the SSD.

is managed in a separate bucket – and the remaining buckets are divided evenly based on data size. This strategy may result in small read-sensitive buckets that can be removed by merging small buckets with their neighbors when the bucket is cooling down.

The process of TrieKV splitting buckets is articulated as follows. An old bucket is pre-split into multiple new buckets. During splitting, the old bucket continues to service incoming requests. When these new buckets are created, the KV items in memory are written to one of them according to their keys. Subsequently, the data in the old bucket are fully sorted and divided based on division keys, and multiple SSTables are originated and placed in new buckets. We pre-allocated sequence numbers for the SSTables during the splitting process to ensure that their order and flushed SSTables are rigorously maintained.

2) Bucket- and Hotness-Based Compaction: We achieve optimized write performance by relaxing the fully sorted requirement at each level; however, this tactic leads to more overlapping SSTables, which in turn spark more read requests on disks. For this reason, compaction must be conducted to push KV pairs downward and reduce the number of levels, but intensive compaction consumes considerable I/O bandwidth and decreases the number of concurrent read requests. If we prioritize the compaction of read-intensive KV pairs and reduce the number of levels hosting these hot read requests, the adverse impact of compaction operations on read requests can be mitigated. To elaborate further, TrieKV, ought to orchestrate the priority scheduling of read-sensitive buckets. Moreover, write-intensive buckets require periodic compaction because determining whether access patterns are constant is a daunting obstacle. A reasonable read performance is demanded when requests are instantaneously transferred to write-intensive buckets. To achieve high write/read performance, we address the following three research problems:

- *Problem 1:* How do we identify access patterns in a workload?
- *Problem 2:* How do we determine an appropriate compaction granularity?
- *Problem 3:* How do we regulate the compaction priorities for read- and write-intensive buckets?

For Problem 1, TrieKV can identify the access patterns in a workload via the hotness identification. To tackle Problem 2, TrieKV uses dynamic data buckets to minimize cold data involved in compaction while conserving I/O bandwidth. When it comes to Problem 3, TrieKV integrates the hotness information and workload access pattern in a bucket to configure the compaction priority. The hot data only accounts for a small portion of the total data. For example, there is less than 3% of the keys being accessed during a 24-hour User Database (UDB) workload [22].

Authorized licensed use limited to: Auburn University. Downloaded on March 14,2025 at 20:35:39 UTC from IEEE Xplore. Restrictions apply.

In the workloads of ZippyDB, there are around 1% of the KV pairs being responsible for 50% of the total Get requests [13]. Then, prioritizing the hot read data to be involved in compaction does not compete with flush threads for I/O bandwidth for a long time.

We present the processes of bucket- and hotness-based compaction as follows. First, hot regions are obtained from the hotness identification module. Second, the compaction priority of the disk bucket is adjusted as

$$p = \frac{\sum_{i=0}^{N} S_{H_i}}{S_{total}} + \frac{S_{total}}{S_{threshold}}, \tag{2}$$

where p is the priority of a bucket and S_{H_i} is the amount of data covered by a hot range in the bucket. S_{total} denotes the total amount of data in the bucket, and $S_{threshold}$ is the rated data size in the bucket. Hence, we combine the data hotness and volume to select the on-disk buckets that must be involved in the compaction at the most. By employing a higher number of hot regions, TrieKV enables the aggressive compaction of read-intensive buckets for high-performance read services. Moreover, the amount of data in the buckets is incorporated to facilitate a slightly more write-intensive compaction to save I/O bandwidth.

In realistic workloads [13], [40], [41], the hot read region is constantly changing, and this region is likely shifted to a write-sensitive bucket – severely degrading the read performance. Therefore, we slightly compact write-intensive buckets, and the maximum number of levels in the bucket is controlled to avoid read performance fluctuations resulting from having too many levels.

F. Crash Recovery in TrieKV

The data persistence buffer (DPB) in the PMem plays a similar role to WAL that ensures the data update consistency through replaying the data persisted in the PMem. As TrieKV may crash during write operations under certain workloads, data crashes are handled in two ways: (1) KV pairs in DRAM and (2) the radix-tree index in DRAM.

KV pairs in DRAM: The PMem has a backup store for hot data. In case of a system crashes, hot data are moved from the PMem to the DRAM. Note that when KV pairs are not persistent in the PMem, the data are lost without being able to be recovered.

Radix-tree index recovery in DRAM: In a normal shutdown, TrieKV converts the indexes in the DRAM to persistent radix-tree indexes in the PMem. Upon restart, the persistent indexes in the PMem are written back to the DRAM. In case of an abnormal power failure, TrieKV employs replaying the write-ahead logs to recover the MemTable in memory, like the existing KV stores (e.g., LevelDB, RocksDB, etc.). However, TrieKV reissues a write operation to the storage system by scanning the KV pairs in the PMem segments and the backup data to rebuild the index. The multi-threaded parallel scan of persistent segments can be used to rebuild indexes. Because segments may not be fully written, the replaying procedure stops when searching the end markers in the segment.

IV. Experimental Setup

Platform: We deploy a Dell PowerEdge MX840c server with four Intel Xeon Platinum 8153 CPUs, 786GB of DRAM, and 24 128-GB Intel Optane DC PMem [42]. We configure a Samsung PMem 1643A SAS with 960GB of data storage. We install Ubuntu 22.04 LTS with Linux kernel 5.19.0 and an ext4 file system. We configure the Optane DC PMems in App Direct mode. We organized 24 PMems into four regions, and there is only one region in the experiment. In the experiment, we use an 8GB PMem, like prior work using the DRAM-PMem-SSD architecture, such as NoveLSM [15], MatrixKV [43], and MioDB [44]. When the data in the PMem is more than 8GB, a flush procedure is triggered to dump the data to the SSD. Therefore, when loading 100GB data in the experiment, there is 92GB data stored in the SSD. We created an FSDAX-based namespace. The namespace is a block device, and we set up an ext4 file system on the PMem and mounted it in DAX mode. In addition, we create the PMem, where an 8-GB file is constructed. We utilize mmap to build a page table for data accesses, thereby mapping the page table to the DPB in the PMem in TrieKV.

Alternative KV stores: We compare TrieKV to HyperLevelDB and RocksDB. The MemTable size is 64MB. The adjacent level amplification factor and SSTable size were set to default. To study the impact of MemTables of various sizes, we set up an 8-GB MemTable for RocksDB. HyperLevelDB and RocksDB are disabled from the PMem, in which a WAL is adopted for data recovery. In addition, we deploy NoveLSM and LightKV in the App Direct mode. NoveLSM relies on PMem to cut back the (de)serialization cost. We configure NoveLSM with a 64MB DRAM MemTable and an 8GB PMem MemTable for the test. We set up the other default configuration parameters in this test. LightKV is a cross-media KV store anchored on a DRAM-PMem-SSD approach, which reduces read latency and WA by establishing an RH-Tree and partition-based compaction. The recent studies [45], [46] demonstrated that parameter tuning affects the KV store performance. In addition, the findings suggest differences in performance based on the default parameters. Because HyperLevelDB, RocksDB, and NoveLSM have various default parameters, we decide to – in the experiments – use a 64MB MemTable – and configure level0-slow and level0-stop as 8 and 12, respectively. For simplicity, we turn off compression in the comparison KV stores. Unless otherwise specified, the default key and value size are 16 B and 1 KB, respectively. Table II lists the characteristics of DB_Bench and YCSB.

V. An Empirical Study

A. Write Amplification (WA) Under DB_Bench

We conduct an experiment to investigate the WA and random write throughput of the comparison KV stores under db_bench with a value of 1 KB, diverse write data volumes, and a uniform distribution. Fig. 8 reveals that under the workload of 100 GB, the WAs of HyperLevelDB, RocksDB, RocksDB-8GB, NoveLSM, LightKV are higher than those of TrieKV by a factor of approximately 6.9x, 6.7x, 4.2x, 3.8x, and 1.4x respectively. As the write data volume increases, the WA of TrieKV stabilizes

Authorized licensed use limited to: Auburn University. Downloaded on March 14,2025 at 20:35:39 UTC from IEEE Xplore. Restrictions apply.

TABLE II
CHARACTERISTICS OF WORKLOADS

Workload 1 (DB_Bench)			
Type	Feature	Distribution	Value Size(1KB)
Load	100% Writes	Uniform	0.25x,1x, 4x,16x,64x
Read	100% Reads	Uniform	
Workload 2 (YCSB)			
Type	Feature		Distribution
Load	100% Writes		Uniform
A	50% Reads, 50% Updates		Zipfian
B	95% Reads, 5% Updates		Zipfian
C	100% Reads		Zipfian
D	95% Reads, 5% Inserts		Latest
E	5% Inserts, 95% Range Queries		Zipfian
F	50% Reads, 50% Read-Modify-Writes		Zipfian

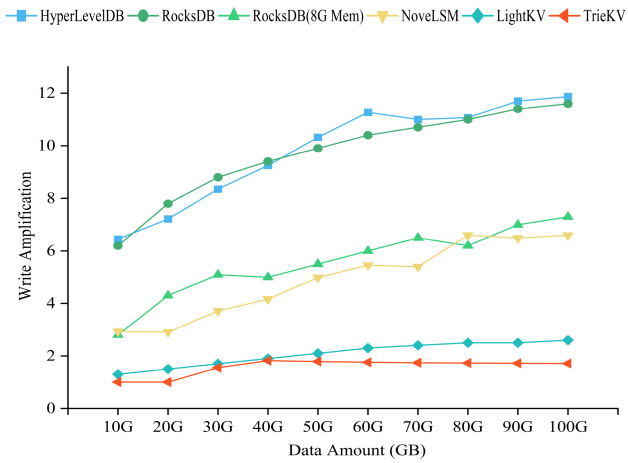


Fig. 8. WA under the workloads with various data volume.

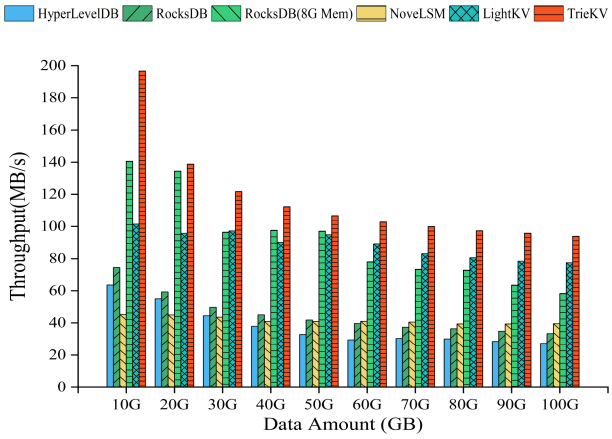
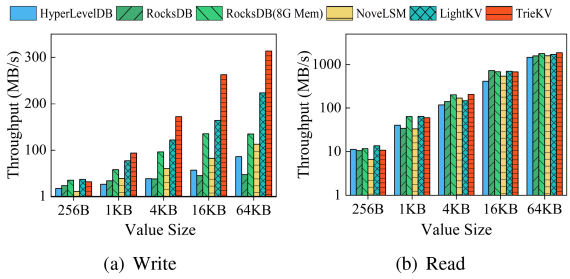


Fig. 9. Throughput under workloads with different data volume.

whereas the WAs of the other KV stores go up. Fig. 9 shows the throughput of the tested KV stores with various data volumes. During compaction, TrieKV aggregates KV pairs with the same prefix into the same bucket, and only the data in the same bucket is involved in the compaction. HyperLevelDB and RocksDB regulate the data volume involved in the compaction according to the ranges of the SSTables, resulting in a high WA.



(a) Write (b) Read

Fig. 10. Write/Read performance under DB_Bench with various-sized values and 100-GB data volume.

NoveLSM employs a large persistent write buffer to curb the flush frequency, which mitigates high WA. Nevertheless, overlapping key ranges also occur in NoveLSM during compaction, giving rise to a large WA. LightKV and TrieKV are similar in terms of organizing KV pairs with the same key prefix into the same partition through the partition of the RH-Tree, and controlling the amount of compressed data to reduce the WA; however, TrieKV incorporates the hotness factor more than LightKV: first, cold data – in the flush phase – are actively migrated from the write buffer to the SSD to boost the memory utilization rate and retain the data with frequent updates in the PMem to realize the update; second, compaction prioritizes the frequently accessed buckets to optimize TrieKV's throughput.

B. Performance Evaluation Under DB_Bench

Write Performance: Fig. 10(a) shows the write performance of TrieKV where 100GB of uniformly distributed data are randomly inserted. The write throughput of all the KV stores linearly rises as the value size reaches 64KB from 256 B. TrieKV improves the random write throughput by a factor of 2.8x and 2.9x compared with HyperLevelDB and RocksDB, respectively. Compared with NoveLSM, TrieKV revamps the throughput by up to 2.1x thanks to the following reasons. In NoveLSM, a large-sized MemTable in the PMem handles write requests – and the large write buffer stores a portion of update requests, which benefit the WA measures. However, the large memory component occupies space for a long period; thus, the space may be insufficient for buffering write requests. Moreover, NoveLSM overlooks the problem of the large memory component involved in flushing – a major culprit behind a high compaction overhead. TrieKV's throughput improvement, compared against LightKV, ranges from 20% to 60% because of optimized memory utilization: TrieKV's hotness-aware flush retains update-intensive data in memory – and flushes cold data to the disk.

Read performance: Let us study the performance of TrieKV by randomly reading data from a dataset with 100GB of random load data. At first, the read performance was unstable because of compaction in the KV stores. After compaction was completed, the read performance stabilized. Fig. 10(b) shows the random performances of the comparison KV stores. The NVM stores only 8% of the data, and the SSD read performance outperformed the other read performances. The compaction procedure triggered by reads enables data to be stored in the SSD at a level.

Authorized licensed use limited to: Auburn University. Downloaded on March 14,2025 at 20:35:39 UTC from IEEE Xplore. Restrictions apply.

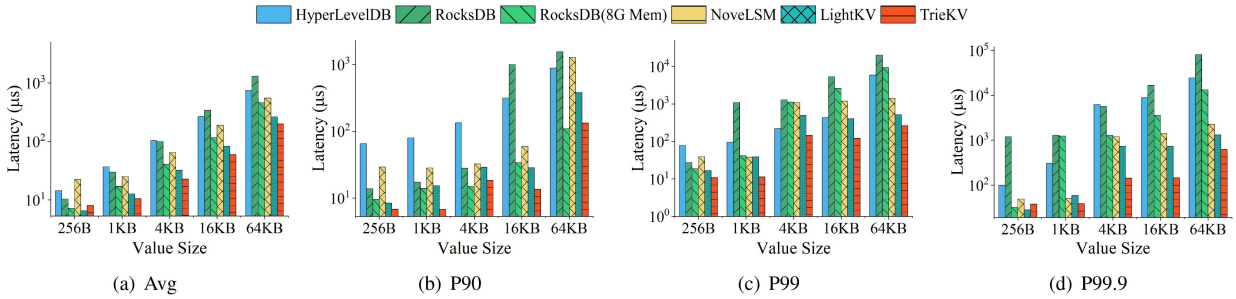


Fig. 11. Write latency under DB_Bench with various-sized values.

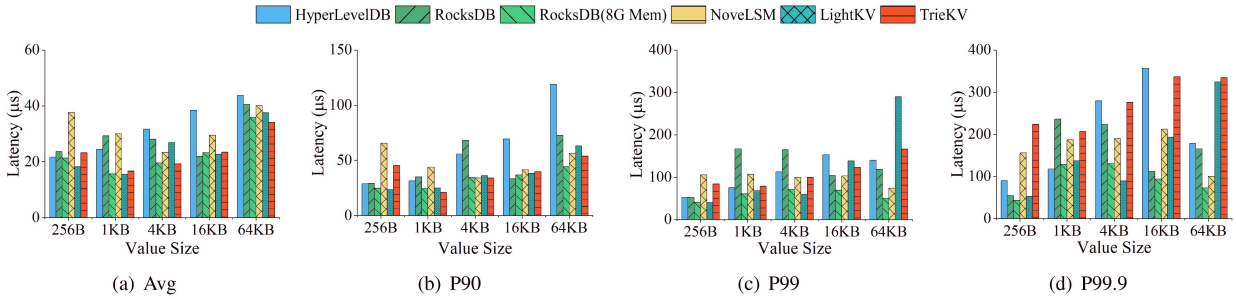


Fig. 12. Read latency under DB_Bench with various-sized values.

In TrieKV, bucket- and hotness-based compaction rapidly stores data. TrieKV has a clear read-throughput edge over the other KV stores.

Latency: We drive the experiments using DB_Bench to measure the read/write tail latency of the comparison KV stores under the workloads that follow a uniform distribution. We load 100GB of data randomly and read them randomly. Figs. 11 and 12 unveil the average, 90th, 99th, and 99.9th percentile latency of the comparison KV stores under the workloads with diverse value sizes. For example, the 99.9th percentile latency of HyperLevelDB, RocksDB, and NoveLSM is higher than that of TrieKV by 41x, 39x, and 7x, respectively. TrieKV exhibits the lowest average read latency compared with the other KV stores; in contrast, the uniform-distribution reads lack hotness, so the bucket- and hotness-based compaction mechanisms are rarely invoked. When the data are compacted to present the ordering, the trie-tree index brings down the search overhead in memory, which contributes to the shortened average read latency.

DRAM ratio sensitivity analysis: In this paper, we configure 8GB of data persistence buffer in the PMem [15], [43], [44]. In TrieKV, the DRAM space is used to store indexes of KV pairs that are placed in the PMem. When there are size limits for PMem in TrieKV, the DRAM usage size depends on the number of KV pairs in the PMem. Notably, the worst-case space consumption is 35 B per key in the adaptive radix tree implemented in TrieKV.

To study the impact of DRAM and PMem usage on the system performance, we configure DB_Bench with five value sizes, that are 256 B, 1 KB, 4KB, 16 KB, and 64 KB. We used 8-GB PMem in TrieKV and LightKV under DB_Bench with 100 GB initial data loading in the experiment. Then, we record the DRAM usage of the two KV stores after they became stable.[1] The experimental results are listed in Table III. The DRAM usage of TrieKV is slightly higher than that of LightKV because we add some pointers for hot-region recognition in the internal nodes of the adaptive radix tree in TrieKV. However, this space cost can be negligible on a commercial server with hundreds of GB DRAM [47]. Under workloads with large-sized KV pairs, the DRAM space of LightKV and TrieKV is significantly lower than that of HyperLevelDB, RocksDB, and NoveLSM.

TABLE III
DRAM USAGE OF KV STORES UNDER WORKLOADS WITH DIFFERENT VALUE SIZES

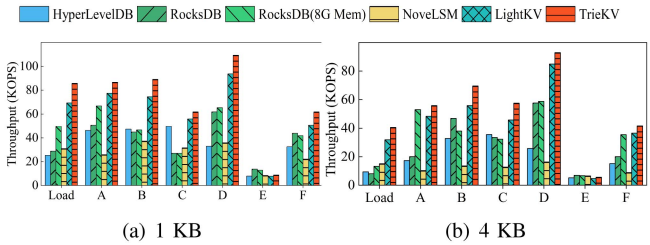
Value size	DRAM Usage				
	HyperLevelDB	RocksDB	NoveLSM	LightKV	TrieKV
256 B	138 MB	136 MB	135 MB	910 MB	1100 MB
1 KB	128 MB	132 MB	128 MB	227 MB	296 MB
4 KB	129 MB	127 MB	128 MB	57 MB	75 MB
16 KB	126 MB	128 MB	131 MB	14 MB	19 MB
64 KB	128 MB	128 MB	126 MB	3 MB	4 MB

C. Performance of TrieKV Under YCSB

Under YCSB, we first feed 100 GB data – and replay workloads A-F. The default value size in YCSB is 1 KB, and NoveLSM is configured with a 4-KB value to achieve the best performance. Fig. 13 plots the measured performance of the evaluated KV stores under workloads of 1 and 4 KB.

[1]The stable system refers to the one where the performance and DRAM usage of the KV stores does not present significant fluctuation under the workloads; thus, the measured DRAM usage can represent the space occupation of the KV stores in most cases. When the KV store is just running, the DRAM usage increases slowly. Meanwhile, when the system triggers a flush operation or the immutable memtable is deleted, the DRAM usage decreases. Thus, it is difficult to reflect the DRAM usage of the KV store before the KV store becomes stable.

Authorized licensed use limited to: Auburn University. Downloaded on March 14,2025 at 20:35:39 UTC from IEEE Xplore. Restrictions apply.



(a) 1 KB (b) 4 KB

Fig. 13. Throughput under YCSB with 1-KB and 4-KB value sizes.

In the load stage, the throughput of TrieKV is 2.7× higher than that of NoveLSM under the write-dominated workloads. NoveLSM – managing skip lists as its index – has a time complexity of *O(logn)* for locating a data point. The cost of inserting/updating KV pairs is high. Conversely, TrieKV adopts a trie-tree as its index, so TrieKV's lookup time complexity is lower than that of its peers. As expected, TrieKV achieves a higher throughput than NoveLSM. In summary, TrieKV outperforms NoveLSM in terms of read performance because read requests following a Zipfian distribution can hit in the memory.

Under read-intensive workloads (B, C, and D), TrieKV exhibits a significant performance improvement over the other KV stores: TrieKV employs a trie-tree index to optimize read performance. We observe strong access locality in the workload with a Zipfian distribution. When read and write requests arrive, variable granularity hot-region recognition stores read- and update-intensive data in memory to store the data updates in the PMem. The bucket- and hotness-based compaction schemes enable TrieKV to identify read-intensive buckets, where the prioritized compaction scheme is integrated. The bucket serving the most read requests quickly becomes ordered, achieving a high read performance.

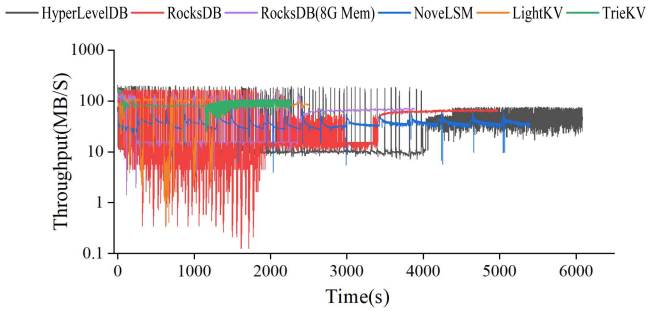
Under scan workload E, these queries must access data at each level in the KV stores. TrieKV has more levels than HyperLevelDB and RocksDB. The bucket- and hotness-based compaction alleviates the adverse impact of multiple levels on the scan performance. Therefore, all KV stores show similar performance.

Tail latency and write stall: The KV stores are sensitive to tail latency under write-intensive and latency-critical workloads in production environments. Mixed read-write workloads have recently gained popularity. Now, we examine the read-write tail latency in this experiment. We apply the same measurement method mentioned in a previous study [48] to investigate the tail latency. We evaluate the tail latency of TrieKV under YCSB-A with 1 KB and 4 KB KV pairs. Table IV tabulates the average 90%, 99%, and 99.9% percentile latency of TrieKV, LightKV, and NoveLSM. When the value is 4 KB, the 99.9 percentile latency of LevelDB, RocksDB, and NoveLSM are 36, 4.5, and 7.3 times higher than that of TrieKV, respectively. With a value of 1 KB, TrieKV has a higher average write latency than NoveLSM.

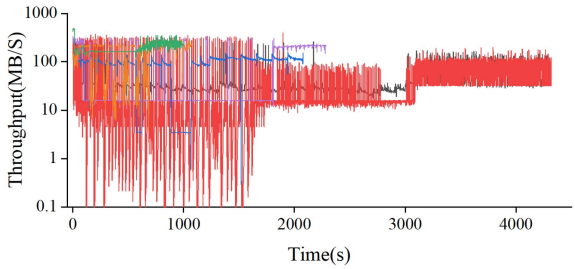
Fig. 14 presents the real-time throughput of each KV store under YCSB-A. During the load phase, NoveLSM refreshes the MemTables into SSTables, thereby increasing the read/write latency due to periodic write stalls.

TABLE IV
TAIL LATENCY OF TESTED KV STORES UNDER YCSB-A

Value Size	KV Store	Latency			
		avg.	90%	99%	99.9%
1 KB	HyperlevelDB	21	56.9	108.2	465.9
	RocksDB	15.6	28.7	219	242.4
	RocksDB(8G)	14.8	23.1	210.5	229.9
	NoveLSM	28	42.3	84	153.1
	LightKV	13.5	25.1	260.1	328.3
	TrieKV	**11.4**	**19.5**	**67.6**	**147.3**
4 KB	HyperlevelDB	61	116.2	277.1	8182.6
	RocksDB	49	53.4	1108.6	1219
	RocksDB(8G)	18.7	28.5	221.6	251
	NoveLSM	43.2	50	98.9	1830.3
	LightKV	23.7	39.6	81.2	593.2
	TrieKV	**17.7**	**21.3**	**84.9**	**220.2**



(a) 1 KB



(b) 4 KB

Fig. 14. Real-time throughput under YCSB-A with 1 KB and 4KB value sizes.

LightKV and TrieKV perform partition compaction to adjudicate the amount of compacted data, which avoids long write stalls. When it comes to loading workload A, read requests – looking up data level by level – cause a high read latency that leads to deteriorated real-time throughput. The compaction triggered by the read requests lowers the residence level. The disk is accessed when there is only one level. Subsequently, the throughput is enhanced through the shortened read latency. Compared with the other KV stores, TrieKV compacts frequently read data – and quickly pushes the data to reduce the hierarchy level, accelerating the hot-region data lookup. The findings reveal that TrieKV achieves a higher throughput than the other methods.

D. Extended Experiment

1) Performance of KV Stores Under Strict Constraints on the Size Occupied on Disk: We studied the performance of KV stores in the case of different percentages of disk occupancy, as

Authorized licensed use limited to: Auburn University. Downloaded on March 14,2025 at 20:35:39 UTC from IEEE Xplore. Restrictions apply.

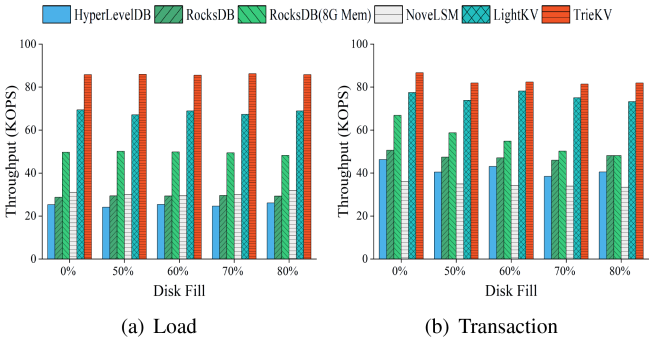


Fig. 15. The throughput of the KV stores in load and transaction execution stages under different constraints on the size occupied on the disk.

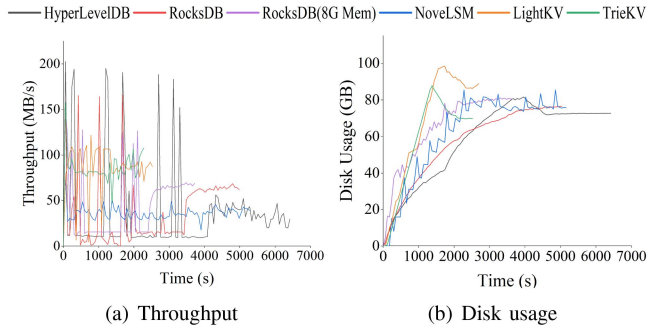


Fig. 16. Throughput and disk usage of KV stores under YCSB-A.

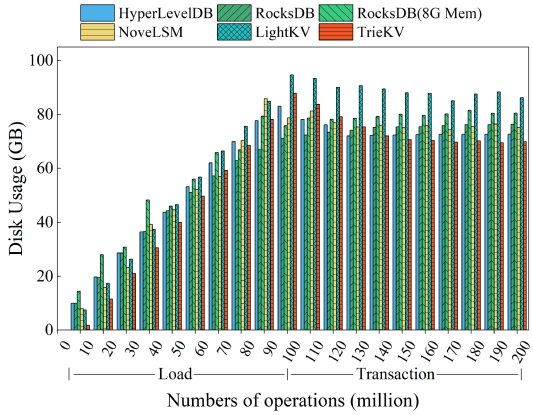


Fig. 17. Impact of numbers of operations (in millions) on the disk usage in the load and transaction stages.

shown in Fig. 15. The capacity of the tested SSD is 960GB. There are invalid and valid data of the LSM-tree; thus, the actual data size is larger than the valid data size. To guarantee enough space for the workload, we measure the performance of the KV store under strict constraints on the size occupied on disk such as 0% full, 50% full, 60% full, 70% full, and 80% full. Under YCSB-A, the performance of all KV stores slightly decreases as the amount of available space on the disk reduces; however, the change is minor. We present the reasons as follows. When we prefill data in the SSD, the SSD controller re-distributes the prefilling data as evenly as possible. In this case, the incoming random data cannot be concentrically written in a portion of space in flash memory, thereby avoiding wear-leveling issue. Using this redistribution mechanism, the data prefilling procedure rarely impairs the SSD performance. As the large amount of prefilling data can degrade the SSD performance. In addition, inside an SSD, the over provisioning (OP) is used to improve wear-leveling and garbage collection, which can improve the impact of pre-written data on SSD performance.

In addition, we conducted experiments to illustrate the space usage on the disk of TrieKV and LightKV using YCSB-A workload. Notably, YCSB-A is configured with a uniform distribution at load phase and 50% read-50% update and Zipfian distribution at transaction phase. We measure the throughput of KV stores and space occupation on the disk. As shown in Fig. 16(a), TrieKV achieves higher throughput at the load phase because its tiring compaction strategy is featured with high memory utilization. TrieKV provides higher space utilization on the disk that HyperlevelDB and RocksDB but has a similar disk occupation to LightKV. At the transaction phase, TrieKV performs compaction on the updated-intensive buckets, quickly reducing the disk space occupied by invalid data. As illustrated in Fig. 16(b), we find that there are exiting fluctuations in the disk usage for KV stores. In addition, before the throughput decrease, the disk usage decreases. This is because the high amount of data involved in the compaction, which leads to an intensive read/write I/Os. Then, CPU usage and bandwidth between the CPU and SSDs are highly utilized, which blocks foreground write operations and degrades the real-time throughput. Upon completion of the compaction, the disk space occupied by invalid data is reclaimed, lowering disk usage. The throughput remains high as disk usage increases because the amount of data involved in the compaction is relatively small in this period. Consequently, the compaction will not take up the disk bandwidth for a long time, which mainly servers the flush process.

2) Amount of Data Written to the SSD in KV Stores: We performed experiments to study the space occupation on the disk, see Fig. 17.

We used YCSB-A to load 100 million KV pairs with 1 KB value into the KV stores, and the data volume is around 100 GB. Then, 100 million requests (50% read and 50% update) are issued to KV stores, in which there are around 50 GB updated data. We record the disk usage of KV stores per 10 million requests. When the amount of load data is less than 8 GB, HyperLevelDB and RocksDB dump SSTables into the disk; however, RocksDB (8 G Mem), NoveLSM, LightKV, and TrieKV cannot store data in the disk because the data in the PMem is no more than 8 GB. When the data written into the PMem exceeds 8 GB, the KV store writes data in the disk. When loading 10 GB data, RocksDB (8 G Mem), NoveLSM, and LightKV dump the data once. The two KV stores trigger the dumping procedure when the memory component size reaches a threshold (e.g., memtable, segment). The whole component is dumped into the disk at once. TrieKV dumps the cold data to disk and keeps the hot data in the PMem according to the data hotness.

After loading 100 million KV pairs into the KV store, there is only about 80 GB data in the disk. We find that the random-generation keys have some duplicates. Then, KV stores need

Authorized licensed use limited to: Auburn University. Downloaded on March 14,2025 at 20:35:39 UTC from IEEE Xplore. Restrictions apply.

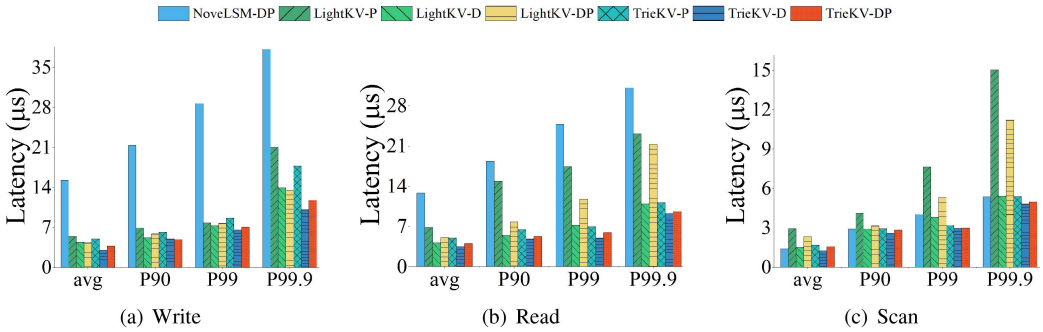


Fig. 18. Performance of random read, write, and scan when index and data are placed in DRAM, PMem, and DRAM-PMem.

to reclaim this portion of the data in compaction. It means that some of the KV pairs have been updated in the write procedure. The tiering compaction-based KV stores (LightKV, TrieKV) have larger disk usage than HyperLevelDB, RocksD, RocksDB (8 G Mem), and NoveLSM. At the transaction execution phase, TrieKV prioritizes compaction or prefetching for hot read data in the disk; afterward, it conducts compaction on the write/update-intensive bucket. At the early stage of the transaction phase, the disk usage is higher than that of other KV stores. When the compaction on the hot read data is complete, TrieKV performs compaction on the bucket with more invalid data, which decreases the disk usage. We find that the disk usage of TrieKV is the least compared with alternative KV stores.

3) Memory-Integrated Management: To study the performance of memory-integrated management in TrieKV, we modify the NVM-based skip list index persistent MemTable in NoveLSM by placing the index in the DRAM and the data in PMem. We refer to this method as NoveLSM-DP, which is compared with TrieKV. We also evaluate TrieKV alongside LightKV (LightKV-DP). In addition, we implement two distinct storage approaches for comparison: one where both indexes and data are allocated in the DRAM, referred to as TrieKV-D, and another where both are situated in the PMem, designated as TrieKV-P. We name the memory-integration management in TrieKV as TrieKV-DP. There is no restriction on the PMem size, which is sufficient to accommodate all KV pairs. Next, we randomly introduce ten million records into the KV store and conduct random write/read and scan operations.

Fig. 18 shows the latency of random read, write, and scan of TrieKV and LightKV with different indexes and data placement strategies. The symbol LightKV-P refers to LightKV that maintains the RH-Tree in the PMem and keeps its strong consistency through a synchronizing update approach. LightKV-D manages the RH-Tree and the persistent write buffer (PWB) in the DRAM without persisting. LightKV-DP locates the RH-tree in DRAM and PMem, and stores the PWB in the PMem. TrieKV-D stores both the index and DPB in DRAM. TrieKV-P places both the index and DPB in the PMem. TrieKV-DP denotes that the index is placed in DRAM and the DPB is stored in the PMem. The results unveil that the write latency of NoveLSM-DP and LightKV-DP is 310% and 27% higher than that of TrieKV-DP. The two KV stores' read latency is 220% and 25% higher than that of TrieKV-DP. In TrieKV-D, the average latency of random read, write, and scan is 8%, 27%, and 43% lower than their respective counterparts of LightKV-D. The skip-list structure organizes data ordered through data comparison. The trie-tree structure makes use of prefix matching to order data, which in turn improves read and write performance.

Under the scan workload, the skip list structure requires only one pointer jump to locate the subsequent key in a query. Conversely, the trie-tree structure of TrieKV requires multiple pointer jumps to find a target key. We then use a memory pool to cluster indexes, aiming to enhance the cache hit rate of the indexes compared with skip lists that dynamically and randomly store the indexes. As a result, the scan performance across these tested KV stores share a comparable level of efficiency. The hash structure is employed in the RH-Tree leaf nodes in LightKV. In a scan operation, all data in the leaf nodes must be traversed.

Both TrieKV and LightKV place KV pairs' indexes in DRAM and KV pairs in the PMem to balance performance and persistence. In other word, when placing the indexes in the PMem, instant recovery can be achieved at the cost of lower performance. However, placing the indexes in DRAM provides higher performance, but the system recovery cost is high when rebuilding the indexes. Unlike TrieKV-DP, TrieKV-P achieves real-time recovery at the expense of a low performance. TrieKV-D offers the highest performance but involves high data recovery cost. On the contrary, the separate placement approach for indexes and data can balance system performance and recovery costs. TrieKV-DP well balances the system performance and recovery cost.

4) Memory Pooling: To validate the memory-pooling mechanism of the trie-tree structure, we keep MemTable in the DRAM and PMem. Then, we assess the allocation and recycling efficiency of the memory pooling by inserting the data into the MemTable. Subsequently, we employ a value of 1KB and an average index node size of approximately 35 B.

Fig. 19 unravels the random read, write, and scan performances of the KV stores. TrieKV-D has an edge over TrieKV-D-nopool in terms of allocation recycling efficiency. Unlike TrieKV-Log, TrieKV-DPB allocates data with the same prefix to the same region. This allocation technique increases the cache hit rate – and lowers the latency under scan workloads. Flush operations involve a low serialization overhead, and bucket management reduces the space recycling overhead. In addition, we also compared the slab-based memory allocator

Authorized licensed use limited to: Auburn University. Downloaded on March 14,2025 at 20:35:39 UTC from IEEE Xplore. Restrictions apply.

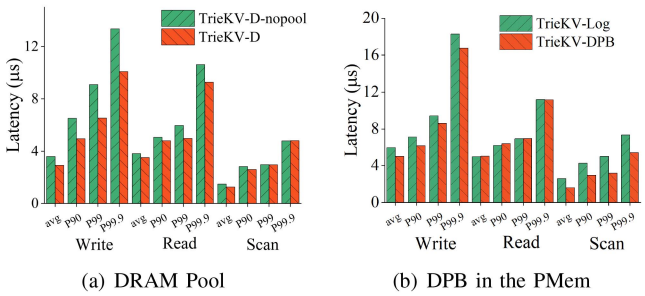


Fig. 19. The latency of TrieKV using memory pooling in the DRAM and DPB in the PMem.

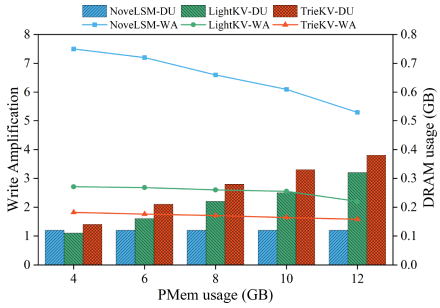


Fig. 21. The DRAM usage and write amplification of KV stores using different-sized PMem.

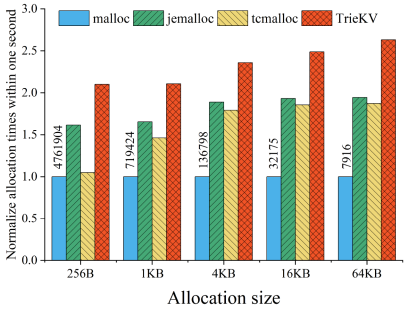


Fig. 20. Normalized Numbers of allocations that can be completed in a second with different allocators. The numbers are normalized against those of the malloc allocator.

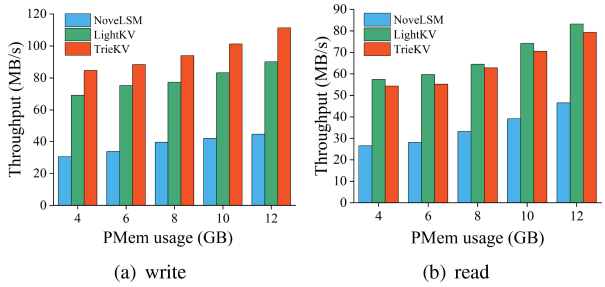


Fig. 22. Throughput of random read and write of KV stores with different PMem usages.

used in TrieKV with jemalloc and tcmalloc that link different libraries and use different memory allocation methods. We study the allocation efficiency of each allocator by measuring the allocation times for different-sized objects (ranging from 256 B to 64KB) per second. As shown in Fig. 20, the slab-based memory allocator used in TrieKV has the largest number of allocation operations in the case of all memory allocation sizes. Our proposed allocator mostly requests large-sized chunks from memory and then manages them in TrieKV. During the allocation and reclamation, fewer locks are used, which avoids the costs of switching from the user to the kernel mode, thereby reducing allocation overhead. The large number of allocation operations performed by the memory allocator in the same period means the high allocation efficient in TrieKV.

5) The Impact of PMem Usage on the Performance: We also experimentally investigated the impact of PMem usage on the system's performance. LightKV has a configurable parameter MAX LEAF NODES, representing the number of partitions. A larger MAX LEAF NODES leads to more partitions. Accordingly, there are more partitions in the PMem. We adjust the PMem usage in LightKV by setting its MAX LEAF NODES.

Fig. 21 presents the write amplification of KV stores under different PMem usages. We find that the write amplification of KV stores decreases with the increment of PMem usage. This is because the larger-size PMem stores more KV pairs, and the amount of data in the disk reduces, thereby lowering the frequency of triggering compaction and write amplification. The DRAM usage of the KV stores is shown in Fig. 21. LightKV and TrieKV have higher DRAM usage because there are more indexes for KV pairs. DRAM usage is positively correlated with the number of indexed KV pairs. However, this memory cost can be negligible on commercial servers that are equipped with hundreds of GB memory [13].

Fig. 22 illustrates the throughput of random read and write in the case of different-sized PMem. For random writes, NoveLSM and TrieKV do not flush data to the SSD when the data volume in PMem is less than 8GB. NoveLSM flushes the data from PMem to the SSD once PMem is fully filled. However, the data in PMem that has been written to the SSD still occupies the PMem. Therefore, the data in DRAM cannot be dumped into PMem. In TrieKV, when the PMem space is fully filled, the cold data is dumped to the SSD according to the data hotness, freeing up the PMem space to serve new writs. LightKV dumps data when each data segment is full, and LightKV does not distinguish between hot and cold data. Then, TrieKV provides a high performance. There are more data being buffered in the PMem as its space usage increases. Thus, the KV stores can quickly fetch data from the PMem, which increases the random read throughput.

6) Performance of Compaction Schemes: Now, we assess the performance of bucket- and hotness-based compaction. We compare – under YCSB-A – TrieKV, bucket-based TrieKV-no-hot, and partition-based LightKV.

Fig. 23 unfolds the real-time throughput of each KV store under YCSB-A. The key distribution is uniform in the load stage. As opposed to LightKV and TrieKV-no-hot, TrieKV embraces an aggressive flush operation to ensure sufficient memory space to accommodate write requests with a high throughput. The requests in the run phase follow a Zipfian distribution. In contrast to the fixed-partition in LightKV and TrieKV-no-hot, TrieKV

Authorized licensed use limited to: Auburn University. Downloaded on March 14,2025 at 20:35:39 UTC from IEEE Xplore. Restrictions apply.

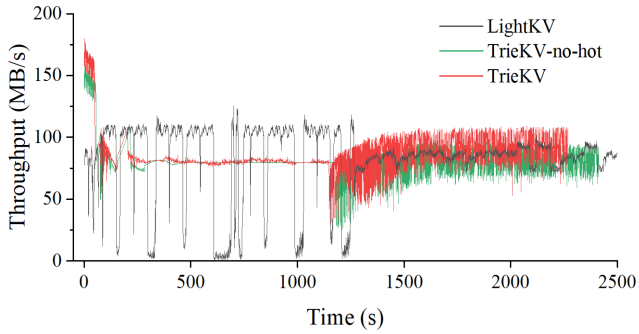


Fig. 23. Real-time throughput under YCSB-A.

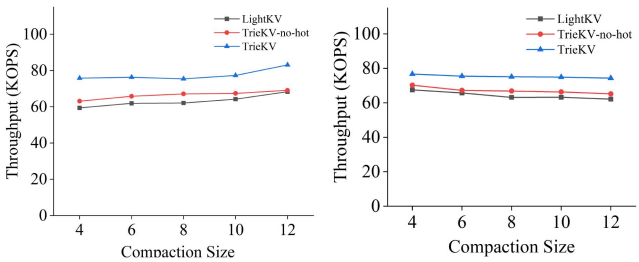


Fig. 24. Load and transaction execution throughput with different numbers of SSTables involved in a compaction (compaction size).

dynamically adjusts the memory bucket to achieve a high write performance. Unlike LightKV and TrieKV-no-hot with bucket-based compaction, TrieKV identifies a read-intensive bucket that is prioritized to realize superior read performance with high-priority compaction.

7) the Impact of Compaction Size on Throughput: Fig. 24 presents the throughput in the case of a compaction size, which is denoted by the different numbers of SSTables involved in a compaction. Since a smaller number of partitions is more likely to trigger compaction, we set 64 partitions in each KV store, which is the same as that of LightKV in terms of sensitivity experiments on compaction size. In this case, we can obtain the number of SSTables involved in compaction. In this experiment, we employed YCSB-A. The partition-based compaction strategy in LightKV is a variant of tiering compaction. When increasing the number of SSTables involved in compaction, the compaction frequency can decrease, reducing write amplification. However, a low frequency of compaction occurrences results in more SSTables being stored in a partition, which reduces search efficiency. The throughput of random read decreases as the number of SSTables involved in a compaction increases. This trend also exists in other KV stores; however, TrieKV prioritizes the hot read partitions for compaction, achieving the lowest read performance degradation under read-skew workloads.

VI. Related Work

We present a comprehensive review of KV stores from the perspectives of PMem, hotness identification methods, and techniques that enhance in-memory flush performance.

KV stores for PMem: Indexes are placed in DRAM or PMem tailored to specific applications. While the volatile index housed in DRAM delivers enhanced performance, it necessitates a long recovery time. The persistent index in PMem offers instant availability after a restart, but its performance is not on par with DRAM's speed. Table V lists PMem- and DRAM-PMem-based studies. NoveLSM [15] introduces a persistent MemTable that reduces the cost of data serialization. In NoveLSM, KV pairs are inserted to MemTable in the PMem from MemTable in the DRAM. The write size in the PMem are smaller than 256 B, causing high WA. When the large-sized MemTable in the PMem is flushed to the SSD, the performance of the KV store suffers from serialization/deserialization costs. TrieKV uses the adaptive radix-tree structure in DRAM to index the data in the PMem. The indexes are updated in the DRAM, which does not cause WA; the KV pair is buffered until its size larger than that of the PMem. Afterward, the KV pair is written into the PMem, which avoids small writes in the PMem. TrieKV dumps the cold data in the PMem to the SSD and keeps the hot data in the PMem, reducing the overhead of querying the hot data with serialization/deserialization. MioDB [44] manages multilevel persistent skip lists in PMem, but using skip lists for data indexing in PMem degrades the query performance. SLM-DB [16] applies a B+-Tree to index data on the disk but reduces the scan performance. wB+Tree [49] stores the internal and leaf nodes in PMem, thereby reducing the persistence time by the virtue of bitmaps and slot arrays. Regardless, this scheme is rooted in expensive logging or CoW to split a node. In Wort [50], three radix tree variants are employed for the PMem to implement correct writing and fault recovery. MatrixKV [43] provides a matrix container to receive MemTables in NVM – and performs fine-grained data compression, with the goal of curtailing the cost of data compression from that of NVM to that of an SSD.

FPTree [51] exploits fingerprints to accelerate point lookup –and utilizes hardware transactional memory for concurrency control. HiKV [17] leverages hash tables in the PMem and B+ trees in the DRAM to speed up single-key and range queries. Viper [52] allocates threads to different PMem regions to minimize the ratio of threads to memory size. Data are kept in the memory-aligned segments. LightKV [18] deploys an RH-tree as the data index; However, the leaf nodes must be reordered in a sequence query. When a segment in PMem is filled, it is dumped into the SSD. The approach to indiscriminately migrate data can move hot data to low-speed storage devices and cold data to high-speed storage devices, which can degrade the read performance. In TrieKV, we consider the data hotness and only dump cold KV pairs to the SSD, which avoids hot data being dumped to a low-speed storage device. In addition, TrieKV uses the hotness-based compaction strategy to prioritize the compaction of hot data and reduce the level of hot data, reducing unnecessary disk accesses caused by data querying and improving the system read performance. Yan et al. [27] integrated universal data structures into the existing OLTP storage engines powered by the LSM-tree. Pacman [37] proposed a PMem-aware compaction method that simplified garbage collection based on PMem features. The proposed TrieKV method uses a cross-medium trie-tree index. Unlike LightKV, TrieKV strategically evaluates the hotness of

Authorized licensed use limited to: Auburn University. Downloaded on March 14,2025 at 20:35:39 UTC from IEEE Xplore. Restrictions apply.

TABLE V
RELATED WORKS BASED ON PMEM AND DRAM-PMEM (✔: GOOD, ❍: PLAIN, –: WORSE)

	Publication	Index Structure	Index Persistence	Insertion Performance	Scan Performance	Space Efficiency	Persistence Overhead
PMem	NoveLSM	Skip Lists	✔	✔✔✔	❍	✔	✔
	MioDB	Skip Lists	✔	✔✔✔	❍	✔	✔
	SLM-DB	B+Tree	✔	✔	❍	✔	–
	wB+Tree	B+Tree	✔	✔	✔	✔	–
	Wort	Trie	✔	✔✔	✔	❍	✔
	MatrixKV	RowTable	✔	✔	❍	✔	✔
DRAM-PMem	FP-Tree	B+Tree	–	✔	✔	✔	–
	HiKV	B+Tree+Hash	❍	✔	✔	✔	–
	Viper	Hash	–	✔✔✔		✔	✔
	LightKV	Trie + Hash	–	✔✔	❍	–	✔
	Yan et al.	Trie	–	✔✔	✔	–	✔
	TrieKV	Trie	–	✔✔	✔	–	✔

data and the read/write capabilities of DRAM and PMem to optimize data placement. Our approach effectively capitalizes on the strengths of a heterogeneous memory architecture.

Hotness Identification methods: L2SM [7] maintains a hotness-detecting bitmap (HotMap) composed of multilevel bloom filters in memory, maps keys to the hotness-detecting bitmap, and determines the data hotness based on the mapped hierarchy. Nonetheless, key-based hotness identification can only identify keys that are frequently accessed, and determining the spatial locality of hot keys is formidable. TRIAD [6] integrates several cold and hot key detection methods to retain hot data in memory components by counting the number of updates to the KV pairs in memory. Again, this key granularity cannot perceive the spatial locality of hot keys. In addition, Anneser et al. [53] devised a scheme that dynamically samples incoming query requests, tracks accesses to the indexed data, and maintains fine-grained statistical data. This technique furnishes coding migration to optimize both spatial usage and performance. Levandoski. [54] recorded access and analyzed data offline to estimate access frequency. An exponential smoothing-based estimation algorithm was proposed to track the changes in hot and cold data. Tidal-Tree-Mem [10] records the access frequency of SSTables at the bottom of the LSM-tree, moving frequently accessed SSTables to higher positions (e.g., memory or upper levels) to suppress read amplification. Various cache replacement schemes – such as ARC [40], 2Q [41], LRU-K [55], and LIRS [56] – have been applied to memory management. One limitation is that these studies focused on fixed-sized pages. The fixed-range-based cold and hot identification captures a few hot data points within the fixed range, making the entire range hot. Cold data occupy valuable memory space. We designed in TrieKV a hotness-identification scheme to combine individual record- and range-based approaches, which avert memory space wastage caused by hot-key locality and fixed ranges.

Optimizations for efficient compaction: SILK [48] provides an I/O scheduler to delay flushing and compaction until the low-load phase is reached. During these cycles, flush and lower-level compaction are prioritized to mitigate the negative impact of write stalls on the foreground write performance. These methods reduce the probability of write stalls to some extent, but the problem persists in write-intensive scenarios. Blsm [8] employs a merge scheduler to coordinate multilevel compaction. Unfortunately, it limits only the maximum write latency and neglects the high queuing latency. ADOC [57] introduced a tuning framework to alleviate write stalls by automatically adjusting the thread count and batch size. The framework balances the data flow between components, reducing data overflow in the LSM-tree. TrieKV prioritizes compaction in read-intensive buckets, which improves read performance in a KV store. Moreover, TrieKV manages flush, compaction, and prefetch threads to avoid competition and synchronization issues that impair system performance.

VII. DISCUSSION

Balance between performance and data persistence in the PMem-based KV stores: In modern storage architectures, the optimal balance between performance and data persistence can be achieved by utilizing the DRAM-PMem system [18], [27], [28], [58]. With its ultra-high-speed data access, DRAM is suitable for handling high-frequency write and latency-sensitive tasks. PMem can store frequently read indexes and critical data through its performance and persistence advantages. By efficiently utilizing both storage devices, we can achieve optimized system performance and reduce the system's costs. The mixed DRAM-PMem framework is critical in applications where cost-effectiveness, performance, and persistence should be considered simultaneously. For example, transaction logs use DRAM to handle high-frequency writes and persist with PMem through an asynchronous mechanism to ensure data security. In big-data platforms, we can achieve fast response and cost optimization by storing hot data in DRAM and transferring cold data to PMem. In addition, in high-performance computing, temporary data is fully used in low-latency DRAM, while critical results are stored in the PMem to prevent data loss. Database indexes are stored in PMem to improve read performance, while frequently updated data is stored in DRAM to reduce latency. It can enhance system flexibility and efficiency by using an adaptive mechanism to dynamically adjust DRAM and PMem configurations based on real-time data access patterns, workload changes, and performance.

Hot-region recognition module in TrieKV: In the hot-region recognition module in TrieKV, an extra pointer is used in the index node to point to the hotness information, which can reduce the tracking overhead in the hot-region recognition module. In addition, when adjusting the level of hot-region recognition, we need to modify the extra pointer in the index nodes, which

Authorized licensed use limited to: Auburn University. Downloaded on March 14,2025 at 20:35:39 UTC from IEEE Xplore. Restrictions apply.

increases the complexity and maintenance cost of the index structure. In future work, we try to decouple the hot-region recognition module from the index structure to avoid the write amplification issue caused by the module. Therefore, it can be easily integrated into existing KV stores.

VIII. Conclusion

To harness the speed and high bandwidth of DRAM, the persistence of PMem and cost-effective of SSDs, we proposed TrieKV – a high-performance key-value (KV) store powered by hybrid memory and SSDs. TrieKV seamlessly integrates index management, data storage, and persistence in a hierarchical memory-disk architecture. Embracing a holistic design philosophy, TrieKV employs a singular trie-structured index that spans all in-memory KV pairs, enabling consistent identification of access patterns based on data hotness. TrieKV's dynamic allocation strategy optimizes data placement across the DRAM, PMem, and SSD layers, tailored to the unique hotness profile and persistence demands of each KV pair. This approach ensures peak throughput and minimal response times. An extensive evaluation reveals that TrieKV is adroit at shortening write latency, surpassing the existing systems such as HyperLevelDB, RocksDB, NoveLSM, and LightKV.

References

[1] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, pp. 351–385, 1996.

[2] S. Ghemawat and J. Dean, "Leveldb," 2016. [Online]. Available: https://github.com/Level/leveldown/issues/298

[3] Facebook, "Rocksdb, a persistent key-value store for fast storage enviroments," 2019. [Online]. Available: http://rocksdb.org/

[4] F. Chang et al., "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 1–26, 2008.

[5] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Operating Syst. Rev.*, vol. 44, no. 2, pp. 35–40, 2010.

[6] O. Balmau et al., "Triad: Creating synergies between memory, disk and log in log structured key-value stores," in *Proc. 2017 USENIX Annu. Tech. Conf.*, 2017, pp. 363–375.

[7] K. Huang, Z. Jia, Z. Shen, Z. Shao, and F. Chen, "Less is more: De-amplifying I/Os for key-value stores with a log-assisted LSM-tree," in *Proc. IEEE 37th Int. Conf. Data Eng.*, 2021, pp. 612–623.

[8] R. Sears and R. Ramakrishnan, "BLSM: A general purpose log structured merge tree," in *Proc. 2012 ACM SIGMOD Int. Conf. Manage. Data*, 2012, pp. 217–228.

[9] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "Pebblesdb: Building key-value stores using fragmented log-structured merge trees," in *Proc. 26th Symp. Operating Syst. Princ.*, 2017, pp. 497–514.

[10] C. Ma, H. Yang, S. Wu, Y. Wang, and R. Mao, "Tidal-tree-mem: Toward read-intensive key-value stores with tidal structure based on LSM-tree," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 42, no. 2, pp. 423–436, Feb. 2023.

[11] Facebook, "Block cache," 2023. [Online]. Available: https://github.com/facebook/rocksdb/wiki/Block-Cache

[12] C. Luo and M. J. Carey, "Breaking down memory walls: Adaptive memory management in LSM-based storage systems," *Proc. VLDB Endowment*, vol. 14, no. 3, pp. 241–254, 2020.

[13] F. Wu, M.-H. Yang, B. Zhang, and D. H. Du, "AC-key: Adaptive caching for LSM-based key-value stores," in *Proc. 2020 USENIX Conf. Usenix Annu. Tech. Conf.*, 2020, pp. 603–615.

[14] N. Dayan, M. Athanassoulis, and S. Idreos, "Monkey: Optimal navigable key-value store," in *Proc. 2017 ACM Int. Conf. Manage. Data*, 2017, pp. 79–94.

[15] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Redesigning LSMs for nonvolatile memory with novelsm," in *Proc. 2018 USENIX Annu. Tech. Conf.*, 2018, pp. 993–1005.

[16] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y.-R. Choi, "SLM-DB: Single-level key-value store with persistent memory," in *Proc. 17th USENIX Conf. File Storage Technol. (FAST 19)*, 2019, pp. 191–205.

[17] F. Xia, D. Jiang, J. Xiong, and N. Sun, "HiKV: A hybrid index key-value store for DRAM-NVM memory systems," in *Proc. USENIX Annu. Tech. Conf.*, 2017, pp. 349–362.

[18] S. Han, D. Jiang, and J. Xiong, "LightKV: A cross media key value store with persistent memory to cut long tail latency," in *Proc. 36th Int. Conf. Massive Storage Syst. Technol.*, 2020, pp. 1–13.

[19] HyperLevelDB, "A fork of leveldb intended to meet the needs of hyperdex while remaining compatible with LevelDB," 2014. [Online]. Available: https://github.com/rescrv/HyperLevelDB

[20] C. Luo and M. J. Carey, "LSM-based storage techniques: A survey," *VLDB J.*, vol. 29, no. 1, pp. 393–418, 2020.

[21] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proc. 12th ACM SIGMETRICS/PERFORMANCE Joint Int. Conf. Meas. Model. Comput. Syst.*, 2012, pp. 53–64.

[22] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, modeling, and benchmarking RocksDB key-value workloads at facebook," in *Proc. 18th USENIX Conf. File Storage Technol.*, 2020, pp. 209–223.

[23] E. Bortnikov, A. Braginsky, E. Hillel, I. Keidar, and G. Sheffi, "Accordion: Better memory organization for LSM key-value stores," *Proc. VLDB Endowment*, vol. 11, no. 12, pp. 1863–1875, 2018.

[24] Flurry, "Flurry analytics," 2013. [Online]. Available: https://flurry.com

[25] P. Alcorn, "Intel optane dimm pricing: 695 for 128GB, 2595 for 256GB, 7816 for 512GB (update),"2022. [Online]. Available: https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html

[26] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *Proc. 18th USENIX Conf. File Storage Technol.*, 2020, pp. 169–182.

[27] B. Yan et al., "Revisiting the design of LSM-tree based OLTP storage engine with persistent memory," *Proc. VLDB Endowment*, vol. 14, no. 10, pp. 1872–1885, 2021.

[28] W. Zhang, X. Zhao, S. Jiang, and H. Jiang, "Chameleondb: A key-value store for optane persistent memory," in *Proc. 16th Eur. Conf. Comput. Syst.*, 2021, pp. 194–209.

[29] "Ultra-low latency with samsung Z-nand SSD," 2017. [Online]. Available: http://www.samsung.com/us/labs/pdfs/collateral/Samsung_ZNAND_Technology_Brief_v5.pdf

[30] M. Böther, O. Kißig, L. Benson, and T. Rabl, "Drop it in like it's hot: An analysis of persistent memory as a drop-in replacement for NVME SSDs," in *Proc. 17th Int. Workshop Data Manage. New Hardware*, 2021, pp. 1–8.

[31] Samsung, "Pmem 1643a SAS 960 G," 2024. [Online]. Available: https://semiconductor.samsung.com/ssd/enterprise-ssd/pm1643-pm1643a/

[32] Intel, "Optane SSD DC P5800X series," 2024. [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/201861/intel-optane-ssd-dc-p5800x-series-400gb-2-5in-pcie-x4-3d-xpoint/specifications.html

[33] T.-D. Nguyen and S.-W. Lee, "PB-NVM: A high performance partitioned buffer on NVDIMM," *J. Syst. Archit.*, vol. 97, pp. 20–33, 2019.

[34] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: Artful indexing for main-memory databases," in *Proc. IEEE 29th Int. Conf. Data Eng.*, 2013, pp. 38–49.

[35] J. Bonwick et al., "The slab allocator: An object-caching kernel memory allocator," in *Proc. USENIX Summer 1994 Tech. Conf. USENIX Summer 1994 Tech. Conf.*, Boston, MA, USA, 1994, Art. no. 6.

[36] Y. Wu, K. Park, R. Sen, B. Kroth, and J. Do, "Lessons learned from the early performance evaluation of intel optane DC persistent memory in DBMs," in *Proc. 16th Int. Workshop Data Manage. New Hardware*, 2020, pp. 1–3.

[37] J. Wang, Y. Lu, Q. Wang, M. Xie, K. Huang, and J. Shu, "Pacman: An efficient compaction approach for log-structured key-value store on persistent memory," in *Proc. 2022 USENIX Annu. Tech. Conf.*, 2022, pp. 773–788.

[38] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 143–154.

[39] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, "Linkbench: A database benchmark based on the facebook social graph," in *Proc. 2013 ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 1185–1196.

[40] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," *Fast*, vol. 3, no. 2003, pp. 115–130, 2003.

Authorized licensed use limited to: Auburn University. Downloaded on March 14,2025 at 20:35:39 UTC from IEEE Xplore. Restrictions apply.

[41] D. Shasha and T. Johnson, "2Q: A low overhead high performance buffer management replacement algoritm," in *Proc. 20th Int. Conf. Very Large Databases*, 1994, pp. 439–450.
[42] J. Izraelevitz et al., "Basic performance measurements of the intel optane DC persistent memory module," 2019, *arXiv: 1903.05714*.
[43] T. Yao et al., "Matrixkv: Reducing write stalls and write amplification in LSM-tree based KV stores with a matrix container in NVM," in *Proc. 2020 USENIX Conf. Usenix Annu. Tech. Conf.*, 2020, pp. 17–31.
[44] Z. Duan, J. Yao, H. Liu, X. Liao, H. Jin, and Y. Zhang, "Revisiting log-structured merging for KV stores in hybrid memory systems," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2023, pp. 674–687.
[45] S. Sarkar, D. Staratzis, Z. Zhu, and M. Athanassoulis, "Constructing and analyzing the LSM compaction design space (updated version)," 2022, *arXiv:2202.04522*.
[46] S. Alsubaiee, M. J. Carey, and C. Li, "LSM-based storage and indexing: An old idea with timely benefits," in *Proc. 2nd Int. ACM Workshop Manag. Mining Enriched Geo-Spatial Data*, 2015, pp. 1–6.
[47] X. Zhao, S. Jiang, and X. Wu, "Wipdb: A write-in-place key-value store that mimics bucket sort," in *Proc. IEEE 37th Int. Conf. Data Eng.*, 2021, pp. 1404–1415.
[48] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona, "Silk: Preventing latency spikes in log-structured merge key-value stores," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 753–766.
[49] S. Chen and Q. Jin, "Persistent b -trees in non-volatile main memory," *Proc. VLDB Endowment*, vol. 8, no. 7, pp. 786–797, 2015.
[50] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, "Wort: Write optimal radix tree for persistent memory storage systems," in *Proc. 15th USENIX Conf. File Storage Technol.*, 2017, pp. 257–270.
[51] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory," in *Proc. 2016 Int. Conf. Manage. Data*, 2016, pp. 371–386.
[52] L. Benson, H. Makait, and T. Rabl, "Viper: An efficient hybrid PMem-DRAM key-value store," *Proc. VLDB Endowment*, vol. 14, no. 9, pp. 1544–1556, 2021.
[53] C. Anneser, A. Kipf, H. Zhang, T. Neumann, and A. Kemper, "Adaptive hybrid indexes," in *Proc. 2022 Int. Conf. Manage. Data*, 2022, pp. 1626–1639.
[54] J. J. Levandoski, P.-Å. Larson, and R. Stoica, "Identifying hot and cold data in main-memory databases," in *Proc. IEEE 29th Int. Conf. Data Eng.*, 2013, pp. 26–37.
[55] E. J. O'neil, P. E. O'neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," *ACM Sigmod Rec.*, vol. 22, no. 2, pp. 297–306, 1993.
[56] S. Jiang and X. Zhang, "LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 30, no. 1, pp. 31–42, 2002.
[57] J. Yu, S. H. Noh, Y.-R. Choi, and C. J. Xue, "ADOC: Automatically harmonizing dataflow between components in log-structured key-value stores for improved performance," in *Proc. 21st USENIX Conf. File Storage Technol.*, 2023, pp. 65–80.
[58] L. Chen et al., "Workload-aware log-structured merge key-value store for NVM-SSD hybrid storage," in *Proc. IEEE 39th Int. Conf. Data Eng.*, 2023, pp. 2207–2219.



Hui Sun received the PhD degree from Huazhong University of Science and Technology, in 2014. He is an associate professor of Computer Science at Anhui University. His research interests include computer systems, file systems, performance evaluation, and I/O architectures.



Deyan Kong is currently working toward the MS degree in Anhui University. His main research interests include storage systems and Key-Value store management.



Song Jiang (Senior Member, IEEE) received the BS and MS degrees in computer science from the University of Science and Technology of China, in 1993 and 1996, respectively, and the PhD degree in computer science from the College of William and Mary, in 2004. He is currently a professor in the Department of Computer Science and Engineering, University of Texas at Arlington. His research interests include in the areas of operating systems, file and storage systems, and high performance computing.



Yinliang Yue received the bachelor's degree in computer science from the Harbin Institute of Technology, China, in 2005, and the PhD degree in computer architecture from the Huazhong University of Science and Technology, China, in 2011. He is currently a professor with the Zhongguancun Laboratory. His research interests include computer architecture and storage systems.



Xiao Qin (Senior Member, IEEE) received the BS and MS degrees in computer science from Huazhong University of Science and Technology, in 1992 and 1999, respectively, and the PhD degree in computer science from the University of Nebraska-Lincoln, in 2004. He is currently an alumni professor and the director of Graduate Programs in the Department of Computer Science and Software Engineering, Auburn University. He won an NSF CAREER award in 2009. His research interests include parallel and distributed systems, storage systems, and performance evaluation.

Authorized licensed use limited to: Auburn University. Downloaded on March 14,2025 at 20:35:39 UTC from IEEE Xplore. Restrictions apply.