MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM

Ting Yao, Yiwen Zhang, and Jiguang Wan, *Huazhong University of Science and Technology;* Qiu Cui and Liu Tang, *PingCAP;* Hong Jiang, *UT Arlington;* Changsheng Xie, *Huazhong University of Science and Technology;* Xubin He, *Temple University*

https://www.usenix.org/conference/atc20/presentation/yao

This paper is included in the Proceedings of the 2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

Open access to the Proceedings of the 2020 USENIX Annual Technical Conference is sponsored by USENIX.

MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with a Matrix Container in NVM

Ting Yao[1], Yiwen Zhang[1], Jiguang Wan[1*], Qiu Cui[2], Liu Tang[2], Hong Jiang[3], Changsheng Xie[1], and Xubin He[4]

[1]*WNLO, Huazhong University of Science and Technology, China*
Key Laboratory of Information Storage System, Ministry of Education of China
[2]*PingCAP, China*
[3]*University of Texas at Arlington, USA*
[4]*Temple University, USA*

Abstract

Popular LSM-tree based key-value stores suffer from suboptimal and unpredictable performance due to write amplification and write stalls that cause application performance to periodically drop to nearly zero. Our preliminary experimental studies reveal that (1) write stalls mainly stem from the significantly large amount of data involved in each compaction between L_0-L_1 (i.e., the first two levels of LSM-tree), and (2) write amplification increases with the depth of LSM-trees. Existing works mainly focus on reducing write amplification, while only a couple of them target mitigating write stalls.

In this paper, we exploit non-volatile memory (NVM) to address these two limitations and propose MatrixKV, a new LSM-tree based KV store for systems with multi-tier DRAM-NVM-SSD storage. MatrixKV's design principles include performing smaller and cheaper L_0-L_1 compaction to reduce write stalls while reducing the depth of LSM-trees to mitigate write amplification. To this end, four novel techniques are proposed. First, we relocate and manage the L_0 level in NVM with our proposed *matrix container*. Second, the new *column compaction* is devised to compact L_0 to L_1 at fine-grained key ranges, thus substantially reducing the amount of compaction data. Third, MatrixKV increases the width of each level to decrease the depth of LSM-trees thus mitigating write amplification. Finally, the *cross-row hint search* is introduced for the matrix container to keep adequate read performance. We implement MatrixKV based on RocksDB and evaluate it on a hybrid DRAM/NVM/SSD system using Intel's latest 3D Xpoint NVM device Optane DC PMM. Evaluation results show that, with the same amount of NVM, MatrixKV achieves 5× and 1.9× lower 99^{th} percentile latencies, and 3.6× and 2.6× higher random write throughput than RocksDB and the state-of-art LSM-based KVS NoveLSM respectively.

*Corresponding author. Email: jgwan@hust.edu.cn

1 Introduction

Persistent key-value stores are increasingly critical in supporting a large variety of applications in modern data centers. In write-intensive scenarios, log-structured merge trees (LSM-trees) [49] are the backbone index structures for persistent key-value (KV) stores, such as RocksDB [24], LevelDB [25], HBase [26], and Cassandra [35]. Considering that random writes are common in popular OLTP workloads, the performance of random writes, especially sustained and/or bursty random writes, is a serious concern for users [2, 41, 51]. This paper takes random write performance of KV stores as a major concern. Popular KV stores are deployed on systems with DRAM-SSD storage, which intends to utilize fast DRAM and persistent SSDs to provide high-performance database accesses. However, limitations such as cell sizes, power consumption, cost, and DIMM slot availability prevent the system performance from being further improved via increasing DRAM size [4, 23]. Therefore, exploiting non-volatile memories (NVMs) in hybrid systems is widely considered as a promising mechanism to deliver higher system throughput and lower latencies.

LSM-trees [49] store KV items with multiple exponentially increased levels, e.g., from L_0 to L_6. To better understand LSM-tree based KV stores, we experimentally evaluated the popular RocksDB [24] with a conventional system of DRAM-SSD storage, and made observations that point to two challenging issues and their root causes. First, **write stalls** lead to application throughput periodically dropping to nearly zero, resulting in dramatic fluctuations of performance and long-tail latencies, as shown in Figures 2 and 3. The troughs of system throughput indicate write stalls. Write stalls induce highly unpredictable performance and degrade the quality of user experiences, which goes against NoSQL systems' design goal of predictable and stable performance [53, 57]. Moreover, write stalls substantially lengthen the latency of request processing, exerting high tail latencies [6]. Our experimental studies demonstrate that the main cause of write stalls is the large amount of data processed in each L_0-L_1 compaction. The L_0-

L_1 compaction involves almost all data in both levels due to the unsorted L_0 (files in L_0 are overlapped with key ranges). The all-to-all compaction takes up CPU cycles and SSD bandwidth, which slows down the foreground requests and results in write stalls and long-tail latency. Second, **write amplification** (WA) degrades system performance and storage devices' endurance. WA is directly related to the depth of the LSM-tree as a deeper tree resulting from a larger dataset increases the number of compactions. Although a large body of research aims at reducing LSM-trees' WA [20,36,41,44,45,51], only a couple of published studies concern mitigating write stalls [6,31,53]. Our study aims to address both challenges simultaneously.

Targeting these two challenges and their root causes, this paper proposes MatrixKV, an LSM-tree based KV store for systems with DRAM-NVM-SSD storage. The design principle behind MatrixKV is leveraging NVM to (1) construct cheaper and finer granularity compaction for L_0 and L_1, and (2) reduce LSM-trees' depth to mitigate WA. The key enabling technologies of MatrixKV are summarized as follows:

Matrix container. The matrix container manages the unsorted L_0 of LSM-trees in NVM with a receiver and a compactor. The receiver adopts and retains the MemTable flushed from DRAM, one MemTable per row. The compactor selects and merges a subset of data from L_0 (with the same key range) to L_1, one column per compaction.

Column compaction. A column compaction is the fine-grained compaction between L_0 and L_1, which compacts a small key range a time. Column compaction reduces write stalls because it processes a limited amount of data and promptly frees up the column in NVM for the receiver to accept data flushed from DRAM.

Reducing LSM-tree depth. MatrixKV increases the size of each LSM-tree level to reduce the number of levels. As a result, MatrixKV reduces write amplification and delivers higher throughput.

Cross-row hint search. MatrixKV gives each key a pointer to logically sort all keys in the matrix container thus accelerating search processes.

2 Background and Motivation

In this section, we present the necessary background on NVM, LSM-trees, LSM-based KV stores, and the challenges and motivations in optimizing LSM-based KV stores with NVMs.

2.1 Non-volatile Memory

Service providers have constantly pursued faster database accesses. They aim at providing users with a better quality of service and experience without a significant increase in the total cost of ownership (TCO). With the emergence and development of new storage media such as phase-change memory [8,33,48,52], memristors [55], 3D XPoint [28], and STT-MRAM [21], enhancing storage systems with NVMs becomes a cost-efficient choice. NVM is byte-addressable, persistent, and fast. It is expected to provide DRAM-like performance, disk-like persistency, and higher capacity than DRAM at a much lower cost [9,16,61]. Compared to SSDs, NVM is expected to provide 100× lower read and write latencies and up to ten times higher bandwidth [3,10,14,22].

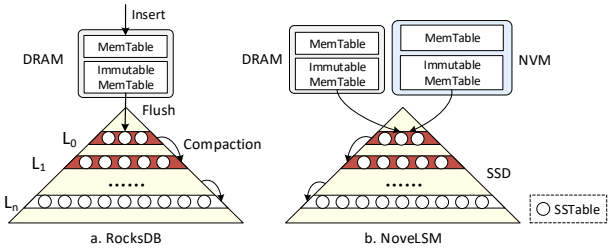


Figure 1: The structure of RocksDB and NoveLSM.

NVM works either as a persistent block storage device accessed through PCIe interfaces or as main memory accessed via memory bus [1,38]. Existing research [31] shows that the former only achieve marginal performance improvements, wasting NVM's high media performance. For the latter, NVM can supplant or complement DRAM as a single-level memory system [27,58,61,65], a system of NVM-SSD [30], or a hybrid system of DRAM-NVM-SSD [31]. In particular, systems with DRAM-NVM-SSD storage are recognized as a promising way to utilize NVMs due to the following three reasons. First, NVM is expected to co-exist with large-capacity SSDs for the next few years [32]. Second, compared to DRAM, NVM still has 5 times lower bandwidth and 3 times higher read latency [28]. Third, a hybrid system balances the TCO and system performance. As a result, MatrixKV focuses on efficiently using NVMs as persistent memory in a hybrid system of DRAM, NVMs, and SSDs.

2.2 Log-structured Merge Trees

LSM-trees [29,49] defer and batch write requests in memory to exploit the high sequential write bandwidth of storage devices. Here we explain a popular implementation of LSM-trees, the widely deployed SSD-based RocksDB [24]. As shown in Figure 1 (a), RocksDB is composed of a DRAM component and an SSD component. It also has a write-ahead log in SSDs protecting data in DRAM from system failures.

To serve write requests, writes are first batched in DRAM by two skip-lists (MemTable and Immutable MemTable). Then, the immutable MemTable is flushed to L_0 on SSDs generating Sorted String Tables (SSTables). To deliver a fast flush, L_0 is unsorted where key ranges overlap among dif-

ferent SSTables. SSTables are compacted from L_0 to deeper levels (L_1, L_2...L_n) during the lifespan of LSM-trees. Compaction makes each level sorted (except L_0) thus bounding the overhead of reads and scans [53].

To conduct a compaction, (1) an SSTable in L_i (called a victim SSTable) and multiple SSTables in L_{i+1} who has overlapping key ranges (called overlapped SSTables) are picked as the compaction data. (2) Other SSTables in L_i that fall in this compaction key ranges are selected reversely. (3) Those SSTables identified in steps (1) and (2) are fetched into memory, to be merged and sorted. (4) The regenerated SSTables are written back to L_{i+1}. Since L_0 is unsorted and each SSTable in L_0 spans a wide key range, the L_0-L_1 compaction performs step (1) and (2) back and forth involving almost all SSTables in both levels, leading to a large all-to-all compaction.

To serve read requests, RocksDB searches the MemTable first, immutable MemTable next, and then SSTables in L_0 through L_n in order. Since SSTables in L_0 contain overlapping keys, a lookup may search multiple files at L_0 [36].

2.3 LSM-tree based KV stores

Existing improvements on LSM-trees includes: reducing write amplification [19, 36, 44, 46, 51, 62–64], improving memory management [7, 39, 56], supporting automatic tuning [17, 18, 40], and using LSM-trees to target hybrid storage hierarchies [5, 47, 50]. Among them, random write performance is a common concern since it is severely hampered by compactions. In the following, we discuss the most related studies of our work in three categories: those reducing write amplification, addressing write stalls, and utilizing NVMs.

Reducing WA: PebblesDB [51] mitigates WA by using guards to maintain partially sorted levels. Lwc-tree [62] provides lightweight compaction by appending data to SSTables and only merging the metadata. WiscKey [36] separates keys from values, which only merges keys during compactions thus reducing WA. The key-value separation solution brings the complexities of garbage collection and range scans and only benefits large values. LSM-trie [59] de-amortizes compaction overhead with hash-range based compaction. VTtree [54] uses an extra layer of indirection to avoid reprocessing sorted data at the cost of fragmentation. TRIAD [44] reduces WA by creating synergy between memory, disk, and log. However, almost all these efforts overlook performance variances and write stalls.

Reducing write stalls: SILK [6] introduces an I/O scheduler which mitigates the impact of write stalls to clients' writes by postponing flushes and compactions to low-load periods, prioritizing flushes and lower level compactions, and preempting compactions. These design choices make SILK exhibits ordinary write stalls on sustained write-intensive and long peak workloads. Blsm [53] proposes a new merge scheduler, called "spring and gear", to coordinate compactions of multiple levels. However, it only bounds the maximum write processing latency while ignoring the large queuing latency [43]. KVell [37] makes KV items unsorted on disks to reduce CPU computation cost thus mitigating write stalls for NVMe SSD based KV stores, which is inapplicable to systems with general SSDs.

Improving LSM-trees with NVMs: SLM-DB [30] proposes a single level LSM-tree for systems with NVM-SSD storage. It uses a B^+-tree in NVM to provide fast read for the single level LSM-tree on SSDs. This solution comes with the overhead of maintaining the consistency between B^+-trees and LSM-trees. MyNVM [23] leverages NVM as a block device to reduce the DRAM usage in SSD based KV stores. NoveLSM [31] is the state-of-art LSM-based KV store for systems with hybrid storage of DRAM, NVMs, and SSDs. NVMRocks [38] aims for an NVM-aware RocksDB, similar to NoveLSM, which adopts persistent mutable MemTables on NVMs. However, as we verified in § 2.4.3, mutable NVM MemTables only reduce access latencies to some extent while generating a negative effect of more severe write stalls.

Since we build MatrixKV for systems with multi-tier DRAM-NVM-SSD storage and redesign LSM-trees to exploit the high performance NVM, NoveLSM [31] is considered the most relevant to our work. We use NoveLSM as our main comparison in evaluations. In addition, we also evaluate PebblesDB and SILK on NVM-based systems since they are state-of-art solutions for reducing WA or write stalls but their original designs are not for the hybrid systems.

2.4 Challenges and Motivations

To explore the challenges in LSM-tree based KV stores, we conduct a preliminary study on the SSD-based RocksDB. In this experiment, an 80 GB dataset of 16bytes-4KB key-value items is written/loaded to RocksDB in uniformly random order. The evaluation environments and other parameters are described in § 5. We record random write throughput every ten seconds as shown in Figure 2. The experimental results expose two challenging issues. **Challenge 1, Write stalls**. System performance experiences peaks and troughs, and the troughs of throughput manifest as write stalls. The significant fluctuations indicate unpredictable and unstable performance. **Challenge 2, Write amplification**. WA causes performance degradation. System performance (i.e., the average throughput) shows a downward trend with the growth of the dataset size since the number of compactions increases with the depth of LSM-trees, bringing more WA.

2.4.1 Write Stalls

In an LSM-based KV store, there are three types of possible stalls as depicted in Figure 1(a). (1) Insert stalls: if MemTable fills up before the completion of background flushes, all insert operations to LSM-trees are stalled [31]. (2) Flush stalls: if L_0 has too many SSTables and reaches a size limit, flushes to

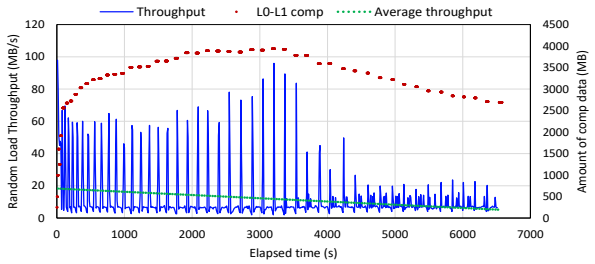


Figure 2: RocksDB's random write performance and L_0-L_1 compactions. *The blue line shows the random write throughput measured in every 10 seconds. The green line shows the average throughput. Each red line represents the duration and amount of data processed in a L_0-L_1 compaction.*

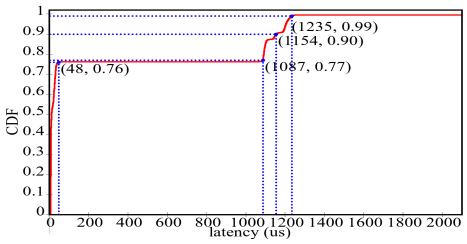


Figure 3: The CDF of latencies of the 80 GB write requests.

storage are blocked. (3) Compaction stalls: too many pending compaction bytes block foreground operations. All these stalls have a cascading impact on write performance and result in write stalls.

Evaluating these three types of stalls individually by recording the period of flushes and compactions at different levels, we find that the period of L_0-L_1 compaction approximately matches write stalls observed, as shown in Figure 2. Each red line represents a L_0-L_1 compaction, where the length along the x-axis represents the latency of the compaction and the right y-axis shows the amount of data processed in the compaction. The average amount of compaction data is 3.10 GB. As we elaborate in § 2.2, since L_0 allows overlapping key ranges between SSTables, almost all SSTables in both levels join the L_0-L_1 compaction. A large amount of compaction data leads to heavy read-merge-writes, which takes up CPU cycles and the SSD bandwidth, thus blocking foreground requests and making L_0-L_1 compaction the primary cause of write stalls.

Write stalls not only are responsible for the low system throughput, but also induce high write latency leading to the long-tail latency problem. Figure 3 shows the cumulative distribution function (CDF) of the latency for each write request during the 80 GB random load process. Although the latency of 76% of the write requests is less than 48 us, the write latency of the 90^{th}, 99^{th}, and 99.9^{th} percentile reaches 1.15, 1.24, and 2.32 ms respectively, a two-order magnitude increase. The high latency significantly degrades the quality of user experiences, especially for latency-critical applications.

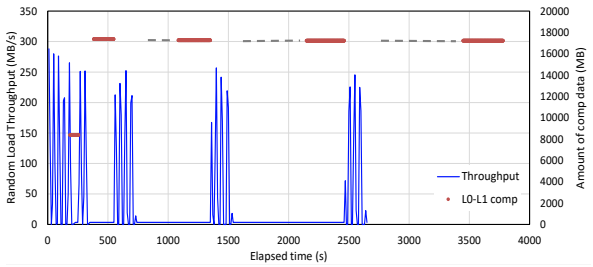


Figure 4: NoveLSM's random write performance and L_0-L_1 compactions. *Comparing to RocksDB in Figure 2, the average period of write stalls is increased.*

2.4.2 Write Amplification

Next, we analyze the second observation, i.e., system throughput degrades with the increase in dataset size. Write amplification (WA) is defined as the ratio between the amount of data written to storage devices and the amount of data written by users. LSM-tree based KV stores have long been criticized for their high WA due to frequent compactions. Since the sizes of adjacent levels from low to high increase exponentially by an amplification factor ($AF = 10$), compacting an SSTable from L_i to L_{i+1} results in a WA factor of AF on average. The growing size of the dataset increases the depth of an LSM-tree as well as the overall WA. For example, the WA factor of compacting from L_1 to L_2 is AF, while the WA factor of compacting from L_1 to L_6 is over $5 \times AF$. The increased WA consumes more storage bandwidth, competes with flush operations, and ultimately slows down application throughput. Hence, system throughput decreases with higher write amplification caused by the increased depth of LSM-trees.

2.4.3 NoveLSM

NoveLSM [31] exploits NVMs to deliver high throughput for systems with DRAM-NVM-SSD storage, as shown in Figure 1(b). The design choices of NoveLSM include: (1) adopting NVMs as an alternative DRAM to increase the size of MemTable and immutable MemTable; (2) making the NVM MemTable mutable to allow direct updates thus reducing compactions. However, these design choices merely postpone the write stalls. When the dataset size exceeds the capacity of NVM MemTables, flush stalls still happen, blocking foreground requests. Furthermore, the enlarged MemTables in NVM are flushed to L_0 and dramatically increase the amount of data in L_0-L_1 compactions, resulting in even more severe write stalls. The worse write stalls magnify performance variances and hurt user experiences further.

We evaluate NoveLSM (with 8 GB NVM) by randomly writing the same 80 GB dataset. Test results in Figure 4 show that NoveLSM reduces the overall loading time by 1.7× compared to RocksDB (Figure 2). However, the period of write stalls is significantly longer. This is because the amount

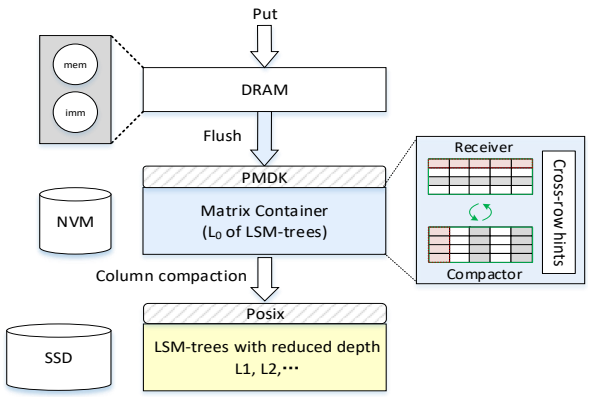


Figure 5: MatrixKV's architectural overview. *MatrixKV is a KV store for systems consisting of DRAM, NVMs, and SSDs.*

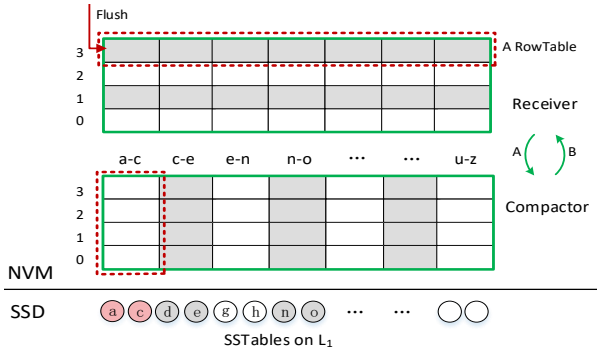


Figure 6: Structure of matrix container. *The receiver absorbs flushed MemTables, one per row. Each row is reorganized as a RowTable. The compactor merges L_0 with L_1 in fine-grained key ranges, one range at a time, referred to as* column compaction. *In Process A, the receiver becomes the compactor once RowTables fill its capacity. In Process B, each column compaction frees a column.*

of data involved in each L_0-L_1 compaction is over 15 GB, which is 4.86× larger than that of RocksDB. A write stall starts when compaction threads call for the L_0-L_1 compaction. Then, the compaction waits and starts until other pending compactions with higher priorities complete (i.e., the grey dashed lines). Finally, performance rises again as the compaction completes. In general, NoveLSM exacerbates write stalls.

From the above analysis, we conclude that the main cause of write stalls is the large amount of data involved in L_0-L_1 compactions, and the main cause of increased WA is the deepened depth of LSM-trees. The compounded impact of write stalls and WA deteriorates system throughput and lengthens tail latency. While NoveLSM attempts to alleviate these issues, it actually exacerbates the problem of write stalls. Motivated by these observed challenging issues, we propose MatrixKV that aims at providing a stable low-latency KV store via intelligent use of NVMs, as elaborated in the next section.

3 MatrixKV Design

In this section, we present MatrixKV, an LSM-tree based key-value store for systems with multi-tier DRAM-NVM-SSD storage. MatrixKV aims to provide predictable high performance through the efficient use of NVMs with the following four key techniques, i.e., the matrix container in NVMs to manage the L_0 of LSM-trees (§ 3.1), column compactions for L_0 and L_1 (§ 3.2), reducing LSM-tree levels (§ 3.3), and the cross-row hint search (§ 3.4). Figure 5 shows the overall architecture of MatrixKV. From top to bottom, (1) DRAM batches writes with MemTables, (2) MemTables are flushed to L_0 that is stored and managed by the matrix container in NVMs, (3) data in L_0 are compacted to L_1 in SSDs through column compactions, and (4) SSDs store the remaining levels of a flattened LSM-tree.

3.1 Matrix Container

LSM-tree renders all-to-all compactions for L_0 and L_1 because L_0 has overlapping key ranges among SSTables. The heavy L_0-L_1 compactions are identified as the root cause of write stalls as demonstrated in § 2.4. NoveLSM [31] exploits NVM to increase the number and size of MemTables. However, it actually exacerbates write stalls by having a larger L_0 and keeping the system bottleneck, L_0-L_1 compactions, on lower-speed SSDs. Hence, the principle of building an LSM-tree based KV store without write stalls is to reduce the granularity of L_0-L_1 compaction via high-speed NVMs.

Based on this design principle, MatrixKV elevates L_0 from SSDs to NVMs and reorganizes L_0 into a matrix container to exploit the byte-addressability and fast random accesses of NVMs. Matrix container is a data management structure for the L_0 of LSM-trees. Figure 6 shows the organization of a matrix container, which comprises one receiver and one compactor.

Receiver: In the matrix container, the receiver accepts and retains MemTables flushed from DRAM. Each such MemTable is serialized as a single row of the receiver and organized as a *RowTable*. RowTables are appended to the matrix container row by row with an increasing sequence number, i.e., from 0 to n. The size of the receiver starts with one RowTable. When the receiver size reaches its size limit (e.g., 60% of the matrix container) and the compactor is empty, the receiver stops receiving flushed MemTables and dynamically turns into the compactor. In the meantime, a new receiver is created for receiving flushed MemTables. There is no data migration for the logical role change of the receiver to the compactor.

RowTable: Figure 7(a) shows the RowTable structure consisting of data and metadata. To construct a RowTable, we first serialize KV items from the immutable MemTable in the

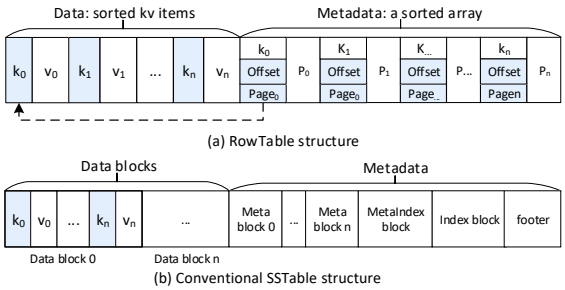


Figure 7: RowTable and conventional SSTable.

order of keys (the same as SSTables) and store them to the data region. Then, we build the metadata for all KV items with a sorted array. Each array element maintains the key, the page number, the offset in the page, and a forward pointer (i.e., p_n). To locate a KV item in a RowTable, we binary search the sorted array to get the target key and find its value with the page number and the offset. The forward pointer in each array element is used for cross-row hint searches that contribute to improving the read efficiency within the matrix container. The cross-row hint search will be discussed in § 3.4. Figure 7(b) shows the structure of conventional SSTable in LSM-trees. SSTables are organized with the basic unit of blocks in accordance with the storage unit of devices such as SSDs and HDDs. Instead, RowTable takes an NVM page as its basic unit. Other than that, RowTables are only different from SSTables in the organization of metadata. As a result, the construction overhead of SSTables and RowTables is similar.

Compactor: The compactor is used for selecting and merging data from L_0 to L_1 in SSDs at a fine granularity. Leveraging the byte addressability of NVMs and our proposed RowTables, MatrixKV allows cheaper compactions that merge a specific key range from L_0 with a subset of SSTables at L_1 without needing to merge all of L_0 and all of L_1. This new L_0-L_1 compaction is referred to as column compaction (detailed in § 3.2). In the compactor, KV items are managed by logical columns. A column is a subset of key spaces with a limited amount of data, which is the basic unit of the compactor in column compactions. Specifically, KV items from different RowTables that fall in the key range of a column compaction logically constitute a column. The amount of these KV items is the size of a column, which is not strictly fixed but at a threshold determined by the size of column compactions.

Space management: After compacting a column, the NVM space occupied by the column is freed. To manage those freed spaces, we simply apply the paging algorithm [3]. Since column compactions rotate the key ranges, at most one page per RowTable is partially fragmented. The NVM pages fully freed after column compactions are added to the free list as a group of page-sized units. To store incoming RowTables in the receiver, we apply free pages from the free list. The 8 GB matrix container contains 2^{11} pages of 4 KB each. Each page is identified by the page number of an unsigned integer. Adding the 8 bytes pointer per list element, the metadata size for each page is 12 bytes. The metadata of the free list occupies a total space of 24 KB on NVMs at most.

It is worth noting that in the matrix container, while columns are being compacted in the compactor, the receiver can continue accepting flushed MemTables from DRAM simultaneously. By freeing the NVM space one column at a time, MatrixKV ends the write stalls forced by merging the entire L_0 with all of L_1.

3.2 Column Compaction

Column compaction is a fine-grained L_0-L_1 compaction that each time compacts only a column, i.e., a small subset of the data in a specific key range. Thus, column compaction can significantly reduce write stalls. The main workflow of column compaction can be described in the following seven steps. (1) MatrixKV separates the key space of L_1 into multiple contiguous key ranges. Since SSTables in L_1 are sorted and each SSTable is bounded by its smallest key and largest key, the smallest keys and largest keys of all the SSTables in L_1 form a sorted key list. Every two adjacent keys represent a key range, i.e., the key range of an SSTable or the gap between two adjacent SSTables. As a result, we have multiple contiguous key ranges in L_1. (2) Column compaction starts from the first key range in L_1. It selects a key range in L_1 as the compaction key range. (3) In the compactor, victim KV items within the compaction key range are picked concurrently in multiple rows. Specifically, assuming N RowTables in the compactor, k threads work in parallel to fetch keys within the compaction key range. Each thread in charge of N/k RowTables. We maintain an adequate degree of concurrent accesses on NVMs with $k = 8$. (4) If the amount of data within this key range is under the lower bound of compaction, the next key range in L_1 joins. The k threads keep forward in N sorted arrays (i.e., the metadata of the RowTables) fetching KV items within the new key range. This key range expansion process continues until the amount of compaction data reaches a size between the lower bound and the upper bound (i.e., $\frac{1}{2}AF \times S_{sst}$ and $AF \times S_{sst}$ respectively). The two bounds guarantee the adequate overhead of a column compaction. (5) Then a column in the compactor is logically formed, i.e., KV items in N RowTables that fall in the compaction key range make up a logical column. (6) Data in the column are merged and sorted with the overlapped SSTables of L_1 in memory. (7) Finally, the regenerated SSTables are written back to L_1 on SSDs. Column compaction continues between the next key range of L_1 and the next column in the compactor. The key ranges of column compaction rotate in the whole key space to keep LSM-trees balanced.

We show an example of column compaction in Figure 8. First, MatrixKV picks the SSTable with key range 0-3 in L_1 as the candidate compaction SSTable. Then, we search the metadata arrays of the four RowTables. If the amount

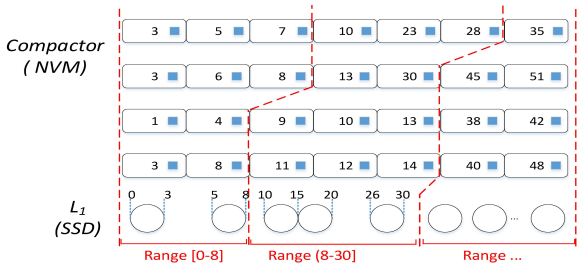


Figure 8: Column compaction: an example. *There are 4 RowTables in the compactor. Each circle represents an SSTable on L_1. Columns are logically divided (red dashed lines) according to the key range of compaction.*

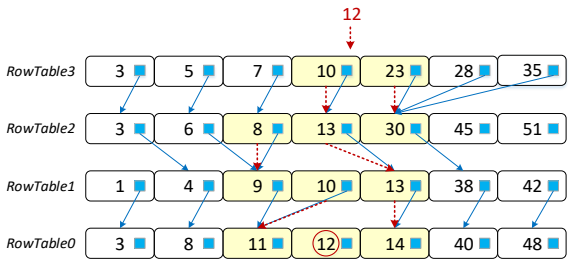


Figure 9: Cross-row hint search. *This figure shows an example of searching the target key ($k = 12$) with forward pointers of each array element.*

of compaction data within key range 0-3 is under the lower bound, the next key range (i.e., key range 3-5) joins to form a larger key range 0-5. If the amount of compaction data is still beneath the lower bound, the next key range 5-8 joins. Once the compaction data is larger than the lower bound, a logical column is formed for the compaction. The first column compaction compacts the column at the key range of 0-8 with the first two SSTables in L_1.

In general, column compaction first selects a specific key range from L_1, and then compacts with the column in the compactor that shares the same key range. Comparing to the original all-to-all compaction between L_0 and L_1, column compaction compacts at a much smaller key range with a limited amount of data. Consequently, the fine-grained column compaction shortens the compaction duration, resulting in reduced write stalls.

3.3 Reducing LSM-tree Depth

In conventional LSM-trees, the size limit of each level grows by an amplification factor of $AF = 10$. The number of levels in an LSM-tree increase with the amount of data in the database. Since compacting an SSTable to a higher level results in a write amplification factor of AF, the overall WA increases with the number of levels (n) in the LSM-tree, i.e., WA=n*AF [36]. Hence, the other design principle of MatrixKV is to reduce the depth of LSM-trees to mitigate WA. MatrixKV reduces the number of LSM-tree levels by increasing the size limit of each level at a fixed ratio making the AF of adjacent levels unchanged. As a result, for compactions from L_1 and higher levels, the WA of compacting an SSTable to the next level remains the same AF but the overall WA is reduced with due to fewer levels.

Flattening conventional LSM-trees with wider levels brings two negative effects. First, since the enlarged L_0 has more SSTables that overlap with key ranges, the amount of data in each L_0-L_1 compaction increases significantly, which not only adds the compaction overhead but also lengthens the duration of write stalls. Second, traversing the larger unsorted L_0 decreases the search efficiency. MatrixKV addresses the first problem with the fine-grained column compaction. The amount of data involved in each column compaction is largely independent of the level width as a column contains a limited amount of data. For the second problem, MatrixKV proposes the cross-row hint search (§ 3.4) to compensate for the increased search overhead due to the enlarged L_0. It is worth noting that locating keys in fewer levels reduces the lookup time on SSDs, since SSTables from L_1 to L_n are well-sorted.

3.4 Cross-row Hint Search

In this section, we discuss solutions for improving the read efficiency in the matrix container. In the L_0 of MatrixKV, each RowTable is sorted and different RowTables are overlapped with key ranges. Building Bloom filters for each table is a possible solution for reducing search overheads. However, it brings costs on the building process and exhibits no benefit to range scans. To provide adequate read and scan performances for MatrixKV, we build cross-row hint searches.

Constructing cross-row hints: When we build a RowTable for the receiver of the matrix container, we add a forward pointer for each element in the sorted array of metadata (Figure 7). Specifically, for a key x in RowTable i, the forward pointer indexes the key y in the preceding RowTable $i-1$, where the key y is the first key not less than x (i.e., $y \geq x$). These forward pointers provide hints to logically sort all keys in different rows, similar to the fractional cascading [11, 53]. Since each forward pointer only records the array index of the preceding RowTable, the size of a forward pointer is only 4 bytes. Thus, the storage overhead is very small.

Search process in the matrix container: A search process starts from the latest arrived RowTable i. If the key range of RowTable i does not overlap the target key, we skip to its preceding RowTable $i-1$. Else, we binary search RowTable i to find the key range (i.e., bounded by two adjacent keys) where the target key resides. With the forward pointers, we can narrow the search region in prior RowTables, $i-1, i-2, \ldots$ continually until the key is found. As a result, there is no need to traverse all tables entirely to get a key or scan a key range.

Cross-row hint search improves the read efficiency of L_0 by significantly reducing the number of tables and elements involved in a search process.

An example of cross-row hint search is shown in Figure 9. The blue arrows show the forward pointers providing cross-row hints. Suppose we want to fetch a target key $k = 12$ in the matrix container, we first binary search RowTable 3 to get a narrowed key range of key=10 to key=23. Then their hints lead us to the key 13 and 30 in RowTable 2 (the red arrows). The preceding key is added into the search region when the target key is not included in the key range of the two hint keys. Next, we binary search between key=8 and key=30. Failing to find the target key, we move to the prior RowTable 1, then RowTable 0, with the forward pointers. Finally, the target key 12 is obtained in RowTable 0.

4 Implementation

We implement MatrixKV based on the popular KV engine RocksDB [24] from Facebook. The LOC on top of RocksDB is 4117 lines [1]. As shown in Figure 5, MatrixKV accesses NVMs via the PMDK library and accesses SSDs via the POSIX API. The persistent memory development kit (PMDK) [1, 60] is a library based on the direct access feature (DAX). Next, we briefly introduce the write and read processes and the mechanism for consistency as follows.

Write: (1) Write requests from users are inserted into a write-ahead log on NVMs to prevent data loss from system failures. (2) Data are batched in DRAM, forming MemTable and immutable MemTable. (3) The immutable MemTable is flushed to NVM and stored as a RowTable in the receiver of the matrix container. (4) The receiver turns into the compactor logically if the number of RowTables reaches a size limit (e.g., 60% of the matrix container) and the compactor is empty. This role change has no real data migrations. (5) Data in the compactor is column compacted with SSTables in L_1 column by column. In the meantime, a new receiver receives flushed MemTables. (6) In SSDs, SSTables are merged to higher levels via conventional compactions as RocksDB does. Compared to RocksDB, MatrixKV is completely different from step 3 through step 5.

Read: MatrixKV processes read requests in the same way as RocksDB. The read thread searches with the priority of DRAM>NVMs>SSDs. In NVMs, the cross-row hint search contributes to faster searches among different RowTables of L_0. The read performance can be further improved by concurrently searching in different storage devices [31].

Consistency: Data structures in NVM must avoid inconsistency caused by system failures [12, 13, 34, 42, 58]. For MatrixKV, writes/updates for NVM only happen in two processes, flush and column compaction. For flush, immutable MemTables flushed from DRAM are organized as RowTables and written to NVM in rows. If a failure occurs in the middle of writing a RowTable, MatrixKV can re-process all the transactions that were recorded in the write-ahead log. For column compaction, MatrixKV needs to update the state of RowTables after each column compaction. To achieve consistency and reliability with low overhead, MatrixKV adopts the versioning mechanism of RocksDB. RocksDB records the database state with a manifest file. The operations of compaction are persisted in the manifest file as version changes. If the system crashes during compaction, the database goes back to its last consistent state with versioning. MatrixKV adds the state of RowTables into the manifest file, i.e., the offset of the first key, the number of keys, the file size, and the metadata size, etc. MatrixKV uses lazy deletion to guarantee that stale columns invalidated by column compactions are not deleted until a consistent new version is completed.

[1] MatrixKV source code is publicly available at https://github.com/PDS-Lab/MatrixKV.

Table 1: FIO 4 KB read and write bandwidth

	SSDSC2BB800G7	Optane DC PMM
Rnd write	68 MB/s	1363 MB/s
Rnd read	250 MB/s	2346 MB/s
Seq write	354 MB/s	1444 MB/s
Seq read	445 MB/s	2567 MB/s

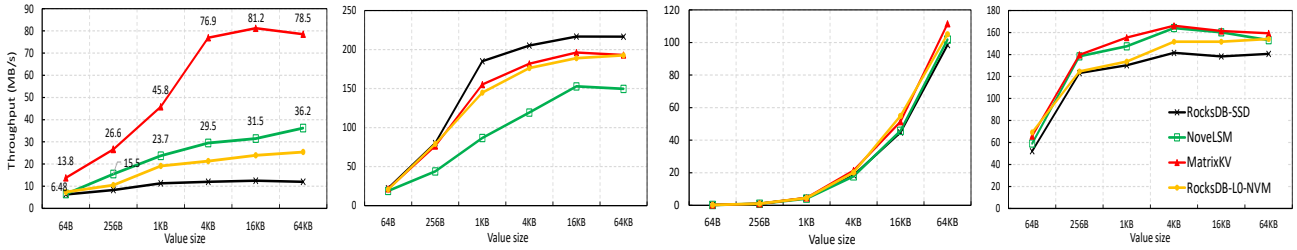
5 Evaluation

In this section, we run extensive experiments to demonstrate the key accomplishments of MatrixKV. (1) MatrixKV obtains better performance on various types of workloads and achieves lower tail latencies (§ 5.2). (2) The performance benefits of MatrixKV come from reducing write stalls and write amplification by its key enabling techniques (§ 5.3).

5.1 Experiment Setup

All experiments are run on a test machine with two Genuine Intel(R) 2.20GHz 24-core processors and 32 GB of memory. The kernel version is 64-bit Linux 4.13.9 and the operating system in use is Fedora 27. The experiments use two storage devices, an 800 GB Intel SSDSC2BB800G7 SSD and 256 GB NVMs of two 128 GB Intel Optane DC PMM [28]. Table 1 lists their maximum single-thread bandwidth, evaluated with the versatile storage benchmark tool FIO.

We mainly compare MatrixKV with NoveLSM and RocksDB (including RocksDB-SSD and RocksDB-L0-NVM). RocksDB-SSD represents the conventional RocksDB on a DRAM-SSD hierarchy. The other three KV stores are for systems with DRAM-NVM-SSD storage. They use 8 GB NVM to be consistent with the setup in NoveLSM's paper and force the majority of the 80 GB test data to be flushed to SSDs. RocksDB-L0-NVM simply enlarges L_0 into 8 GB and stores it in NVM. MatrixKV reorganizes the 8 GB L_0



(a) Random write (b) Sequential write (c) Random read (d) Sequential read

Figure 10: Performance on Micro-benchmarks with different value sizes.

in NVM and enlarges the L_1 in SSDs into the same 8 GB. NoveLSM employs NVM to store two MemTables (2*4 GB). Test results from this configuration can also demonstrate that MatrixKV achieves system performance improvement with the economical use of NVMs. Finally, we evaluate PebblesDB and SILK for systems with DRAM-NVM storage since they are the representative studies on LSM-tree improvement but are not originally designed for systems with multi-tier storage. Unless specified otherwise, the evaluated KV stores assume the default configuration of RocksDB, i.e., 64 MB MemTables/SSTables, 256 MB L_1 size, and AF of 10. The default key-value sizes are 16 bytes and 4 KB.

5.2 Overall performance evaluation

In this section, we first evaluate the overall performance of the four KV stores using db_bench, the micro-benchmark released with RocksDB. Then, we evaluate the performance of each KV store with the YCSB macro-benchmarks [15].

Write performance: We evaluate the random write performance by inserting KV items totaling 80 GB in a uniformly distributed random order. Figure 10(a) shows the random write throughput of four KV stores as a function of value size. The performance difference between RocksDB-SSD and RocksDB-L0-NVM suggests that simply placing L_0 in NVM brings about an average improvement of 65%. We use RocksDB-L0-NVM and NoveLSM as baselines of our evaluation. MatrixKV improves random write throughput over RocksDB-L0-NVM and NoveLSM in all value sizes. Specifically, MatrixKV's throughput improvement over RocksDB-L0-NVM ranges from 1.86× to 3.61×, and MatrixKV's throughput improvement over NoveLSM ranges from 1.72× to 2.61. Taking the commonly used value size of 4 KB as an example, MatrixKV outperforms RocksDB-L0-NVM and NoveLSM by 3.6× and 2.6× respectively. RocksDB-L0-NVM delivers relatively poor performance since putting L_0 in NVM only brings a marginal improvement. NoveLSM uses a large mutable MemTable in NVM to handle a portion of update requests thus slightly reducing WA. However, for both RocksDB and NoveLSM, the root causes of write stalls and WA remain unaddressed, i.e., the all-to-all L_0-L_1 compaction and the deepened depth of LSM-trees.

We evaluate sequential write performance by inserting a total of 80 GB KV items in sequential order. From the test results in Figure 10(b), we make three main observations. First, sequential write throughput is higher than random write throughput for the four KV stores as sequential writes incur no compaction. Second, RocksDB-SSD performs the best since the other three KV stores have an extra NVM tier, requiring data migration from NVMs to SSDs. Three, MatrixKV and RocksDB-L0-NVM have better sequential write throughput than NoveLSM since contracting RowTable/SSTables in NVMs is cheaper than updating the skip list of NoveLSM's large mutable MemTable.

Read performance: Random/sequential read performances are evaluated by reading one million KV items from the 80 GB randomly loaded database. To obtain the read performance free from the impact of compactions, we start the reading test after the tree becomes well-balanced. Figure 10(c) and (d) show the test results of random reads and sequential reads. Since NVM only accommodates 10% of the dataset, the read performance in SSDs dominates the overall read performance. Besides, since a balanced tree is well-sorted from L_1 to L_n on SSDs, the four KV stores exhibit similar read throughputs. MatrixKV does not degrade read performance and even has a slight advantage in sequential reads for two reasons. First, the cross-row hint search reduces the search overhead of the enlarged L_0. Second, MatrixKV has fewer LSM-tree levels, resulting in less search overhead on SSDs.

Macro-benchmarks: Now we evaluate four KV stores with YCSB [15], a widely used macro-benchmark suite delivered by Yahoo!. We first write an 80 GB dataset with 4KB values for loading, then evaluate workload A-F with one million KV items respectively. From the test results shown in Figure 11, we draw three main conclusions. First, MatrixKV gets the most advantage from write/load dominated workloads, i.e., load, and workload A and F. MatrixKV is 3.29× and 2.37× faster than RocksDB-L0-NVM and NoveLSM on the load workload (i.e., random write). Second, MatrixKV maintains adequate performance over read-dominated workloads, i.e., workloads B to E. Third, NoveLSM and MatrixKV behave better on workload D due to the latest distribution, where they both hit more in NVMs and thus MatrixKV can

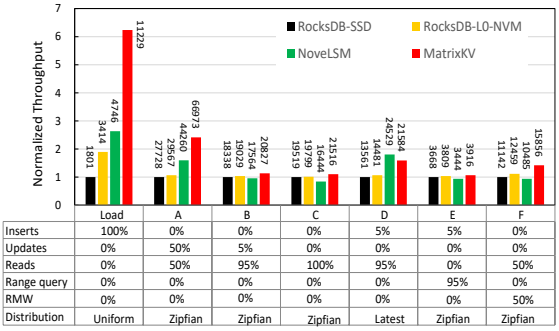


Figure 11: Macro-benchmarks. *The y-axis shows the throughput of each KV store normalized to RocksDB-SSD. The number on each bar indicates the throughput in ops/s.*

Table 2: Tail Latency

Latency (us)	avg.	90%	99%	99.9%
RocksDB-SSD	974	566	11055	17983
NoveLSM	450	317	2080	2169
RocksDB-L0-NVM	477	528	786	1112
MatrixKV	**263**	**247**	**405**	**663**

benefit more from cross-row hints.

Tail latency: Tail latency is especially important for LSM-tree based KV stores, since they are widely deployed in production environments to provide services for write-heavy workloads and latency-critical applications. We evaluate the tail latency with the same methodology used in SILK [6], i.e., using the YCSB-A workload and setting request arrival rate at around 20K requests/s. Table 2 shows the average, 90^{th}, 99^{th}, and 99.9^{th} percentile latencies of four key-value stores. MatrixKV significantly reduces latencies in all cases. The 99^{th} percentile latency of MatrixKV is 27×, 5×, and 1.9× lower than RocksDB-SSD, NoveLSM, and RocksDB-L0-NVM respectively. The test results demonstrate that by reducing write stalls and WA, MatrixKV improves the quality of user experience with much lower tail latencies.

5.3 Performance Gain Analysis

To understand MatrixKV's performance improvement over random write workloads, we investigate the main challenges of LSM-trees (§ 5.3.1) and the key enabling techniques of MatrixKV (§ 5.3.2).

5.3.1 Main Challenges

In this section, we demonstrate that MatrixKV does address the main challenges of LSM-trees, i.e., write stalls and WA.

Write Stalls: We record the throughput of the four KV stores in every ten seconds during their 80 GB random write process (similar to Figures 2 and 4) to visualize write stalls. From the performance variances shown in Figure 12, we draw three observations. (1) MatrixKV takes a shorter time to process the same 80GB random write since it has higher

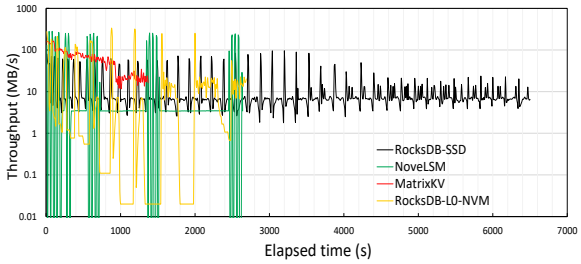


Figure 12: Throughput fluctuation as a function of time. *The random write performance fluctuates where the troughs on curves signify the occurrences of possible write stalls.*

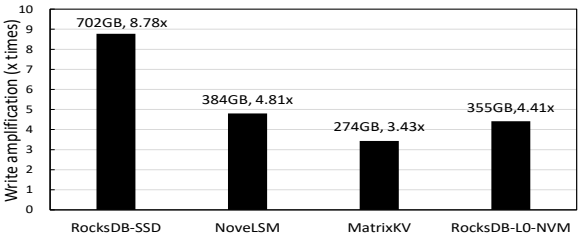


Figure 13: Write amplification of 80 GB random writes.*The numbers on each bar show the amount of data written to SSDs and the WA ratio respectively.*

random write throughput than other KV stores (as demonstrated in § 5.2). (2) Both RocksDB and NoveLSM suffer from write stalls due to the expensive L_0-L_1 compaction. NoveLSM takes longer to process a L_0-L_1 compaction because L_0 maintains large MemTables flushed from NVMs. Comparing to RocksDB-SSD, RocksDB-L0-NVM has lower throughput during write stalls, which means that it blocks foreground requests more severely because of the enlarged L_0. (3) MatrixKV achieves the most stable performance. The reason is that we reduce write stalls by the fine-grained column compaction which guarantees a small amount of data processed in each L_0-L_1 compaction.

Write Amplification: We measure the WA of four systems on the same experiment of randomly writing 80 GB dataset. Figure 13 shows the WA factor measured by the ratio of the amount of data written to SSDs and the amount of data coming from users. The WA of MatrixKV, NoveLSM, and RocksDB-L0-NVM are 2.56×, 1.83×, and 1.99× lower than RocksDB-SSD respectively. MatrixKV has the smallest WA since it reduces the number of compactions by lowering the depth of LSM-trees.

5.3.2 MatrixKV Enabling Techniques

Column compaction: To demonstrate the efficiency of column compaction, we record the amount of data involved, the duration of every L_0-L_1 compaction for four KV stores in the same 80 GB random write experiment. As shown in Figure 14, MatrixKV conducts 467 column compactions, each 0.33 GB,

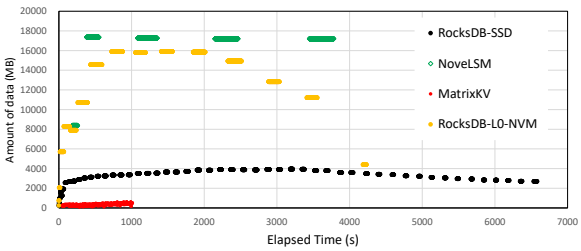


Figure 14: The L_0-L_1 compaction. *Each line segment indicates an L_0-L_1 compaction. The y-axis shows the amount of data involved in the compaction and the length along x-axis shows the duration of the compaction.*

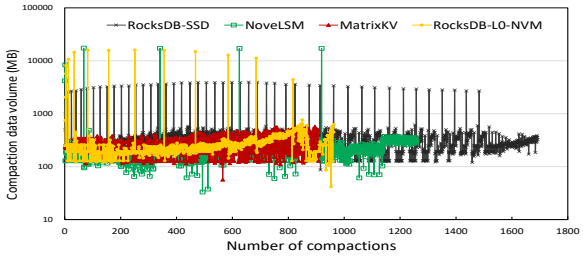
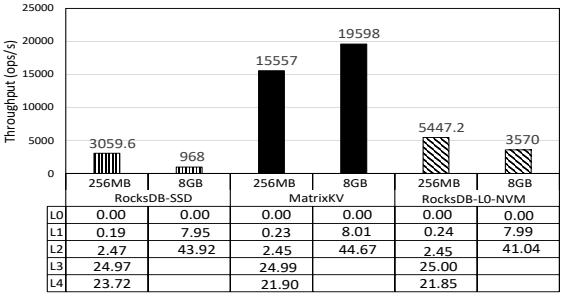


Figure 15: Compaction analysis. *This figure shows the amount of data of every individual compaction during the 80 GB random write.*

written a total of 153 GB data. RocksDB-SSD processes 52 compactions, each 3.1 GB on average, written a total of 157 GB data. MatrixKV processes more fine-grained L_0-L_1 compactions, where each has the least amount of data and the shortest compaction duration. As a result, column compactions have only a negligible influence on foreground requests and finally significantly reduce write stalls. NoveLSM actually exacerbates write stalls since the enlarged MemTables flushed from NVM significantly increase the amount of data processed in each L_0-L_1 compaction.

Overall compaction efficiency: We further record the overall compaction behaviors of four KV stores by recording the amount of data for every compaction during the random write experiment. From the test results shown in Figure 15, we draw four observations. First, MatrixKV has the smallest number of compactions, attributed to the reduced LSM-tree depth. Second, all compactions in MatrixKV process similar amount of data since we reduce the amount of compaction data on L_0-L_1 and does not increase that on other levels. Third, NoveLSM and RocksDB-L0-NVM have fewer compactions than RocksDB-SSD. The reasons are: (1) NoveLSM uses large mutable MemTables to serve more write requests and absorb a portion of update requests, and (2) RocksDB-L0-NVM has an 8 GB L_0 in NVM to store more data. Fourth, the substantial amount of compaction data in NoveLSM and RocksDB stems from the L_0-L_1 compaction.

Reducing LSM-tree depth: To evaluate the technique of flattening LSM-trees, we change level sizes for both RocksDB



	RocksDB-SSD 256MB	RocksDB-SSD 8GB	MatrixKV 256MB	MatrixKV 8GB	RocksDB-L0-NVM 256MB	RocksDB-L0-NVM 8GB
L0	0.00	0.00	0.00	0.00	0.00	0.00
L1	0.19	7.95	0.23	8.01	0.24	7.99
L2	2.47	43.92	2.45	44.67	2.45	41.04
L3	24.97		24.99		25.00	
L4	23.72		21.90		21.85	

Figure 16: Reducing LSM-tree depth. *The y-axis shows random write throughputs of RocksDB and MatrixKV when L_1 is 256 MB/8 GB. The table below shows the data distribution among levels (in GB).*

and MatrixKV. The first configuration is $L_1 = 256MB$ (the default L_1 size of RocksDB). The second configuration is $L_1 = 8GB$. The following levels exponentially increased at the ratio of AF=10. Figure 16 shows the throughput of randomly writing an 80 GB dataset. The table under the figure shows the data distribution on different levels after balancing LSM-trees. The test results demonstrate that both RocksDB and MatrixKV reduce the number of levels by enlarging level sizes, i.e., from 5 to 3. However, they exert opposite influences on system performance. RocksDB-SSD and RocksDB-L0-NVM reduce their random write throughputs by 3× and 1.5× respectively as level sizes increase. The reason is that the enlarged L_1 significantly increases the amount of compaction data between L_0 and L_1. RocksDB-L0-NVM is slightly better than RocksDB-SSD since it puts L_0 in NVMs. For MatrixKV, the throughput increases 25% since the fine granularity column compaction is independent of level sizes. Furthermore, the MatrixKV with 256 MB L_1 shows the performance improvement of only addressing write stalls.

Cross-row hint search: To evaluate the technique of cross-row hint search, we first randomly write an 8 GB dataset with 4 KB value size to fill the L_0 in NVMs for MatrixKV and RocksDB-L0-NVM. Then we search for one million KV items from NVMs in uniformly random order. This experiment makes NVMs accommodate 100% of the dataset to fully reflect the efficiency of cross-row hint searches. The random read throughput of RocksDB-L0-NVM and MatrixKV are 9 MB/s and 157.9 MB/s respectively. Hence, compared to simply placing L_0 in NVMs, the cross-row hint search improves the read efficiency by 17.5 times.

5.4 Extended Comparisons on NVMs

To further verify that MatrixKV's benefits are not solely due to the use of fast NVMs, we evaluate more KV stores on the DRAM-NVM hierarchy, i.e., RocksDB, NoveLSM, PebblesDB, SILK, and MatrixKV, where DRAM stores MemTables, and all other components are stored on NVMs.

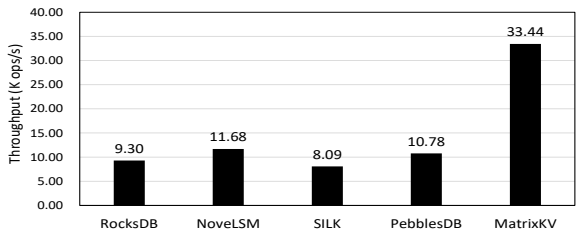


Figure 17: Throughput on NVM based KV stores.

Table 3: Tail latency on NVM-based KV stores

Latency (us)	avg.	90%	99%	99.9%
RocksDB	385	523	701	864
NoveLSM	377	250	808	917
SILK	351	445	575	747
PebblesDB	335	1103	1406	1643
MatrixKV	**209**	**310**	**412**	**547**

Throughput: Figure 17 shows the performance for randomly writing an 80 GB dataset. MatrixKV achieves the best performance among all KV stores. It demonstrates that the enabling techniques of MatrixKV are appropriate for NVM devices. Using NVM as a fast block device, PebblesDB does not show much improvement over RocksDB. SILK is slightly worse than RocksDB since its design strategies have limited advantages over intensive writes.

Tail latency: Tail latencies are evaluated with YCSB-A workload as in § 5.2. Since NVM has a significantly better performance than SSDs, we speed up the requests from clients (60K requests/s). Test results in Table 3 show that with the persistent storage of NVMs most KV stores provide adequate tail latencies. However, MatrixKV still achieves the shortest tail latency.

6 Conclusion

In this paper, we present MatrixKV, a stable low-latency key-value store based on LSM-trees. MatrixKV is designed for systems with multi-tier DRAM-NVM-SSD storage. By lifting the L_0 to NVM, managing it with the matrix container, and compacting L_0 and L_1 with the fine granularity column compaction, MatrixKV reduces write stalls. By flattening the LSM-trees, MatrixKV mitigates write amplification. MatrixKV also guarantees adequate read performance with cross-row hint searches. MatrixKV is implemented on a real system based on RocksDB. Evaluation results demonstrate that MatrixKV significantly reduces write stalls and achieves much better system performance than RocksDB and NoveLSM.

7 Acknowledgement

We thank our shepherd Patrick Stuedi, the anonymous reviewers, and Damien Le Moal for their insightful comments and guidance. We appreciate Yun Liu, Hua Jiang, and Tao Zhong from Intel for the hardware support and maintenance. We also thank Zhiwen Liu and Chenghao Zhu for their efforts in the open-source project.This work was sponsored in part by the National Key Research and Development Program of China No.2018YFB10033005, the Creative Reaserch Group Project of NSFC No.61821003, and the National Natural Science Foundation of China under Grant No.61472152. This work is also partially supported by the US NSF under Grant No.CCF-1704504 and No.CCF-1629625.

References

[1] Persistent memory development kit, 2019. https://github.com/pmem/pmdk.

[2] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP 09)*, pages 1–14, 2009.

[3] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. *Operating systems: Three easy pieces*, volume 151. Arpaci-Dusseau Books Wisconsin, 2014.

[4] Joy Arulraj and Andrew Pavlo. How to build a non-volatile memory database management system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1753–1758. ACM, 2017.

[5] Anurag Awasthi, Avani Nandini, Arnab Bhattacharya, and Priya Sehgal. Hybrid hbase: Leveraging flash ssds to improve cost per throughput of hbase. In *Proceedings of the 18th International Conference on Management of Data*, pages 68–79, 2012.

[6] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. Silk: Preventing latency spikes in log-structured merge key-value stores. In *2019 USENIX Annual Technical Conference (ATC 19)*, 2019.

[7] Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis, and Igor Zablotchi. Flodb: Unlocking memory in persistent key-value stores. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 80–94, 2017.

[8] Meenakshi Sundaram Bhaskaran, Jian Xu, and Steven Swanson. Bankshot: Caching slow storage in fast non-volatile memory. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, 2013.

[9] Geoffrey W Burr, Bülent N Kurdi, J Campbell Scott, Chung Hon Lam, Kailash Gopalakrishnan, and Rohit S Shenoy. Overview of candidate device technologies for

storage-class memory. *IBM Journal of Research and Development*, 52(4.5):449–464, 2008.

[10] Adrian M Caulfield, Arup De, Joel Coburn, Todor I Mollow, Rajesh K Gupta, and Steven Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 385–395, 2010.

[11] Bernard Chazelle and Leonidas J Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(1-4):133–162, 1986.

[12] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 11)*, pages 105–118, 2011.

[13] Nachshon Cohen, David T Aksun, Hillel Avni, and James R Larus. Fine-grain checkpointing with in-cache-line logging. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 19)*, pages 441–454, 2019.

[14] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 133–146, 2009.

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC 10)*, 2010.

[16] George Copeland, Tom W Keller, Ravi Krishnamurthy, and Marc G Smith. The case for safe ram. In *VLDB*, pages 327–335, 1989.

[17] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 79–94, 2017.

[18] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*, pages 505–520, 2018.

[19] Niv Dayan and Stratos Idreos. The log-structured merge-bush & the wacky continuum. In *Proceedings of the 2019 International Conference on Management of Data*, pages 449–466, 2019.

[20] Biplob Debnath, Sudipta Sengupta, and Jin Li. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 25–36, 2011.

[21] Alexander Driskill-Smith. Latest advances and future prospects of stt-ram. In *Non-Volatile Memories Workshop*, pages 11–13, 2010.

[22] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, page 15. ACM, 2014.

[23] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing dram footprint with nvm in facebook. In *Proceedings of the Thirteenth EuroSys Conference*, page 42. ACM, 2018.

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

[25] Sanjay Ghemawat and Jeff Dean. Leveldb, 2016. https://github.com/google/leveldb.

[26] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand S Aiyer, Liyin Tang, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Analysis of hdfs under hbase: a facebook messages case study. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, 2014.

[27] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200, 2018.

[28] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[29] HV Jagadish, PPS Narayan, Sridhar Seshadri, S Sudarshan, and Rama Kanneganti. Incremental organization for data recording and warehousing. In *VLDB*, volume 97, pages 16–25, 1997.

[30] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. Slm-db: Single-level key-value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 191–205, 2019.

[31] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning lsms for nonvolatile memory with novelsm. In *2018 USENIX Annual Technical Conference (ATC 18)*, 2018.

[32] Hyojun Kim, Sangeetha Seshadri, Clement L Dickey, and Lawrence Chiu. Phase change memory in enterprise storage systems: silver bullet or snake oil? *ACM SIGOPS Operating Systems Review*, 48(1):82–89, 2014.

[33] Wook-Hee Kim, Beomseok Nam, Dongil Park, and Youjip Won. Resolving journaling of journal anomaly in android i/o: multi-version b-tree with lazy split. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 273–285, 2014.

[34] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. High-performance transactions for persistent memories. *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 16)*, pages 399–411, 2016.

[35] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. In *The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware*, 2009.

[36] Lu Lanyue, Pillai Thanumalayan Sankaranarayana, Arpaci-Dusseau Andrea C, and Arpaci-Dusseau Remzi H. WiscKey: separating keys from values in ssd-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016.

[37] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 447–461, 2019.

[38] Jianhong Li, Andrew Pavlo, and Siying Dong. Nvmrocks: Rocksdb on non-volatile memory systems, 2017.

[39] Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu. Elasticbf: elastic bloom filter with hotness awareness for boosting read performance in large key-value stores. In *2019 USENIXAnnual Technical Conference (USENIX ATC 19)*, pages 739–752, 2019.

[40] Hyeontaek Lim, David G Andersen, and Michael Kaminsky. Towards accurate and fast evaluation of multi-stage log-structured designs. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 149–166, 2016.

[41] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the 23th ACM Symposium on Operating Systems Principles (SOSP 11)*, pages 1–13, 2011.

[42] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. Dudetm: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 17)*, pages 329–343, 2017.

[43] Chen Luo and Michael J Carey. On performance stability in lsm-based storage systems (extended version). *arXiv preprint arXiv:1906.09667*, 2019.

[44] Balmau Oana Maria, Didona Diego, Guerraoui Rachid, Zwaenepoel Willy, Yuan Huapeng, Arora Aashray, Gupta Karan, and Konka Pavan. Triad: creating synergies between memory, disk and log in log structured key-value stores. In *2017 USENIX Annual Technical Conference (ATC 17)*, 2017.

[45] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. Nvmkv: A scalable, lightweight, ftl-aware key-value store. In *2015 USENIX Annual Technical Conference (ATC 15)*, 2015.

[46] Fei Mei, Qiang Cao, Hong Jiang, and Jingjun Li. Sifrdb: A unified solution for write-optimized key-value stores in large datacenter. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 477–489, 2018.

[47] Prashanth Menon, Tilmann Rabl, Mohammad Sadoghi, and Hans-Arno Jacobsen. Optimizing key-value stores for hybrid storage architectures. In *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering*, pages 355–358, 2014.

[48] Iulian Moraru, David G Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, 2013.

[49] Patrick ONeil, Edward Cheng, Dieter Gawlick, and Elizabeth ONeil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[50] Patrick E O'Neil and Gerhard Weikum. A log-structured history data access method (lham). In *HPTS*, page 0. Citeseer, 1993.

[51] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 497–514. ACM, 2017.

[52] Simone Raoux, Geoffrey W Burr, Matthew J Breitwisch, Charles T Rettner, Y-C Chen, Robert M Shelby, Martin Salinga, Daniel Krebs, S-H Chen, H-L Lung, et al. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2008.

[53] Russell Sears and Raghu Ramakrishnan. blsm: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD 12)*, 2012.

[54] Pradeep Shetty, Richard P Spillane, Ravikant Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building workload-independent storage with vt-trees. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 17–30, 2013.

[55] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The missing memristor found. *nature*, 453(7191):80, 2008.

[56] Dejun Teng, Lei Guo, Rubao Lee, Feng Chen, Siyuan Ma, Yanfeng Zhang, and Xiaodong Zhang. Lsbm-tree: Re-enabling buffer caching in data management for mixed reads and writes. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 68–79, 2017.

[57] Doug Terry. *Transactions and Scalability in Cloud Databases—Can't We Have Both?* USENIX Association, Boston, MA, 2019.

[58] Michael Wu and Willy Zwaenepoel. envy: a non-volatile, main memory storage system. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 94)*, pages 86–97, 1994.

[59] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. Lsm-trie: An lsm-tree-based ultra-large key- value store for small data. In *Proceedings of the USENIX Annual Technical Conference (ATC 15)*, 2015.

[60] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. Finding and fixing performance pathologies in persistent memory software stacks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 19)*, pages 427–439, 2019.

[61] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181, 2015.

[62] Ting Yao, Jiguang Wan, Ping Huang, Xubin He, Qingxin Gui, Fei Wu, and Changsheng Xie. A light-weight compaction tree to reduce i/o amplification toward efficient key-value stores. In *Proceedings of the 33rd International Conference on Massive Storage Systems and Technology (MSST 2017)*, 2017.

[63] Ting Yao, Jiguang Wan, Ping Huang, Yiwen Zhang, Zhiwen Liu, Changsheng Xie, and Xubin He. Geardb: A gc-free key-value store on hm-smr drives with gear compaction. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 159–171, 2019.

[64] Yinliang Yue, Bingsheng He, Yuzhe Li, and Weiping Wang. Building an efficient put-intensive key-value store with skip-tree. *IEEE Transactions on Parallel and Distributed Systems*, 28(4):961–973, 2016.

[65] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 461–476, 2018.