TriangleKV: Reducing Write Stalls and Write Amplification in LSM-Tree Based KV Stores With Triangle Container in NVM

Chen Ding, Ting Yao, Hong Jiang, *Fellow, IEEE*, Qiu Cui, Liu Tang, Yiwen Zhang, Jiguang Wan, and Zhihu Tan

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 work mainly focus on reducing write amplification, while only a couple of them target mitigating write stalls. In this paper, we exploit unique features of non-volatile memory (NVM) to address these two limitations and propose TriangleKV, a new LSM-tree based persistent KV store with multi-tier DRAM-NVM-SSD storage. TriangleKV'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 *triangle container*. Second, the new *right-angle side compaction* is devised to compact L_0 to L_1 at fine-grained key ranges, thus substantially reducing the amount of compaction data. Third, TriangleKV 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 triangle container to keep adequate read performance. We implement TriangleKV based on MatrixKV 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, TriangleKV outperforms RocksDB, NoveLSM and MatrixKV in 99th-percentile latencies by $5.5\times$, $2.1\times$ and $1.1\times$, and random write throughput by $4.9\times$, $3.5\times$ and $1.4\times$ respectively.

Index Terms—Key-value stores, LSM-tree, non-volatile memory

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) [1] are the backbone index structures for persistent key-value stores, such as RocksDB [2], LevelDB [3], HBase [4], and Cassandra [5]. 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 [6], [7], [8]. 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 [9], [10]. 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 [24, 40] 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 [2] with conventional systems 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 Figs. 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 [11], [12]. With respect to latency, write stalls

- *Chen Ding, Ting Yao, Yiwen Zhang, and Zhihu Tan are with the Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System, School of Computer Science and Technology, Huazhong University of Science and Technology, Ministry of Education of China, Wuhan 430074, China. E-mail: {cding, tingyao, zhangyiwen, stan}@hust.edu.cn.*
- *Hong Jiang is with the Computer Science and Engineering Department, University of Texas at Arlington, Arlington, TX 76019 USA. E-mail: hong.jiang@uta.edu.*
- *Qiu Cui and Liu Tang are with PingCAP, Beijing 100192, China. E-mail: {cuiqiu, tl}@pingcap.com.*
- *Jiguang Wan is with the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430074, China, and also with the Shenzhen Huazhong University of Science and Technology Research Institute, Shenzhen 518000, China. E-mail: jgwan@hust.edu.cn.*

Manuscript received 8 February 2022; revised 14 May 2022; accepted 23 June 2022. Date of publication 4 July 2022; date of current version 23 August 2022. This work was sponsored in part by the National Natural Science Foundation of China under Grant 62072196, in part by the Science, Technology and Innovation Commission of Shenzhen Municipality under Grant JCYJ20190809095001781, and in part by the Key Research and Development Program of Guangdong Province under Grant 2021B0101400003.
(Corresponding authors: Jiguang Wan and Ting Yao.)
Recommended for acceptance by J. Zhai.
Digital Object Identifier no. 10.1109/TPDS.2022.3188268

substantially lengthen the latency of request processing, exerting high tail latencies [13]. 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 almost involves all data in both levels due to the unsorted L_0 (files in L_0 are overlapped with key ranges). Thereby, a large number of I/Os on SSDs become a bottleneck which slows down the foreground requests and results in write stalls and high 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 deeper tree resulting from larger datasets increases the number of compactions. Although a large body of research aims at reducing LSM-trees' WA [7], [8], [14], [15], [16], [17], only a couple of published studies concern mitigating write stalls [12], [13], [18]. Our study aims to address both challenges simultaneously.

Targeting these two challenges and their root causes, this paper proposes TriangleKV, an LSM-tree based KV store for systems with DRAM-NVM-SSD storage. The design principle behind TriangleKV 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 TriangleKV are summarized as follows:

1) *Triangle container*. The triangle container manages the unsorted L_0 of LSM-trees in NVM with a triangular data structure. This data structure adopts and retains the MemTable flushed from DRAM as its horizontal bottom side. L_0-L_1 are merged into the L_1 layer on the SSD in sequence according to the sub-key value range, forming one vertical right-angle side per compaction.
2) *Right-angle side compaction*. A right-angle side compaction is the fine-grained compaction between L_0 and L_1, which compacts a small key range at a time. Right-angle side compaction reduces write stalls because it processes a limited amount of data and promptly frees up the vertical right-angle side in NVM for the triangular data structure to accept data flushed from DRAM.
3) *Reducing LSM-tree depth*. TriangleKV increases the size of each LSM-tree level to reduce the number of levels. As a result, TriangleKV reduces write amplification and delivers higher throughput.
4) *Cross-row hint search*. TriangleKV gives each key a pointer to logically sort all keys in the triangle 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 [19], [20], [21], [22], memristors [23], 3D XPoint [24], and STT-MRAM [25], 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 [26], [27], [28]. Compared to SSDs, NVM is expected to provide $100\times$ lower read and write latencies and up to ten times higher bandwidth [29], [30], [31], [32].

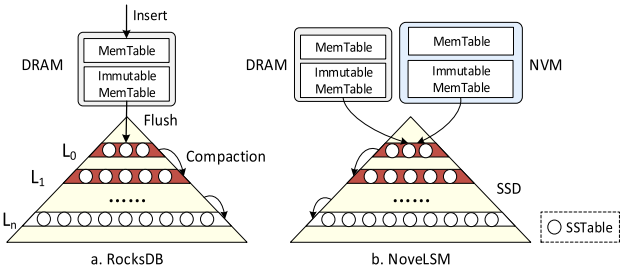


Fig. 1. The structure of RocksDB and NoveLSM.

NVM works either as a persistent block storage device accessed through PCIe interfaces or as main memory accessed through memory bus [33], [34]. Existing research [18] 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 [26], [35], [36], [37], a hybrid system of NVM-SSD [38], or a hybrid system of DRAM-NVM-SSD [18]. 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 [39]. Second, compared to DRAM, NVM still has 5 times lower bandwidth and 3 times higher read latency [24]. Third, a hybrid system balances the TCO and system performance. As a result, TriangleKV 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 [1], [40] 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 [2]. As shown in Fig. 1a, RocksDB is composed of a DRAM component and an SSD component. The DRAM component consists of two sorted skip-list named MemTable and Immutable MemTable. The SSD components contains multiple Sorted String Tables(SSTables) and a write-ahead log protecting data in DRAM from system failures.

To serve write request, the new KV pair is first appended to an write-ahead log to enable recovery. It is then added to the MemTable, which is sorted by keys. Once the MemTable becomes full, it is set to immutable and a new MemTable is allocated to absorb new writes. In the meanwhile, a background thread is started to convert the Immutable MemTable into an SSTable and flush the SSTable to L_0 on SSD. Like the Immutable MemTable, the SSTable also keeps the key in order. Therefore, the keys within each SSTable in L0 are

ordered. But the key ranges of different SSTables in L_0 may overlap, so they appear unordered as a whole. 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 [12].

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 as new victim SSTables. Then, the compaction process goes back to step (1) and uses the new victim SSTables to find more overlapping SSTables in L_{i+1}. (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 [17].

2.3 LSM-Tree Based KV Stores

We categorize existing studies on LSM-tree based KV stores into three types, i.e., those addressing write stalls, reducing write amplification, and utilizing NVMs.

Reducing Write Stalls. SILK [13] 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 has little performance improvement and exhibits ordinary tail latency on sustained write-intensive and long peak workloads. Blsm [12] proposes a new merge scheduler to coordinate compactions of multiple levels. However, the L_0-L_1 compaction still stalls foreground requests severely due to the all-to-all compaction. KVell [41] designs a KV store specialized for NVMe SSDs. The analysis and corresponding solutions for write stalls are not applicable to systems with general SSDs due to different device properties.

Reducing WA. PebblesDB [8] mitigates WA by using guards to maintain partially sorted levels. Lwc-tree [42] provides lightweight compaction by appending data to SSTables and only merging the metadata. WiscKey [17] separates keys from values, which only merges keys during compactions thus reducing WA. LSM-trie [43] de-amortizes compaction overhead with hash-range based compaction. VTtree [44] uses an extra layer of indirection to avoid reprocessing sorted data at the cost of fragmentation. TRIAD [15] reduces WA by creating synergy between memory, disk, and log.

Improving LSM-Trees With NVMs. SLM-DB [38] proposes a single level LSM-tree on a hybrid system of NVM-SSD. It uses a B^+-tree in NVM to provide fast read for the single level LSM-tree on SSDs. Similar to the key-value separation solution, this solution comes with the complexities of garbage collection, range scan, and maintaining the consistency between B^+-trees and LSM-trees. MyNVM [10] leverages NVM as a block device to reduce the DRAM usage in the SSD based KV store. NoveLSM [18] is the state-of-art LSM-based key-value store for systems with hybrid storage of DRAM-NVM-SSD. NVMRocks [33] aims for an NVM-aware RocksDB, which adopts persistent mutable MemTables on NVM, similar to NoveLSM. However, as we verified in Section 2.4.3, mutable NVM MemTables only reduces access latencies to some extent while generating a negative effect of more severe write stalls. MatrixKV [45] lifts L_0 to NVM and manages it with a matrix container and, with the fine-granularity column compaction between L_0 and L_1, MatrixKV reduces write stalls. However, as illustrated in Section 2.4.4, the column compaction is inefficient and can cause additional writes.

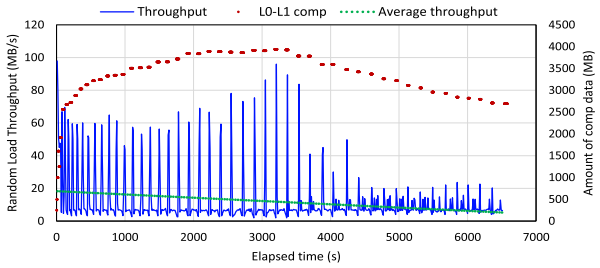


Fig. 2. RocksDB's random write performance and L_0-L_1 compactions. *The blue line shows the random write throughput measured in every 10 seconds. The green line shows the average throughput. Each red line represents the duration and amount of data processed in a L_0-L_1 compaction.*

Since TriangleKV builds on MatrixKV and targets systems with multi-tier DRAM-NVM-SSD storage, NoveLSM [18] is considered the most relevant to our work. We use MatrixKV and NoveLSM as our main comparison baseline 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 system.

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-4 KB key-value items is written/loaded to RocksDB in a uniformly random order. The evaluation environments and other parameters are described in Section 5. We record random write throughput every ten seconds as shown in Fig. 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 Fig. 1a. (1) Insert stalls: if MemTable fills up before the completion of background flushes, all insert

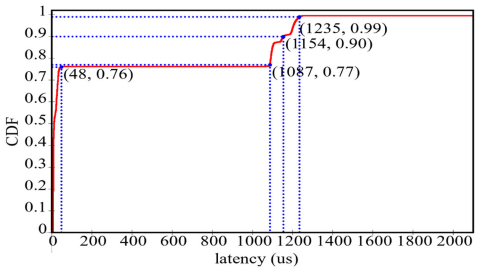


Fig. 3. The CDF of latencies of the 80 GB write requests.

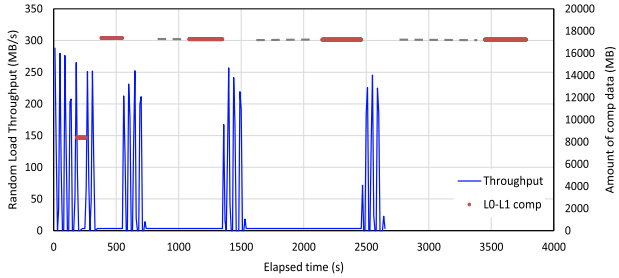


Fig. 4. NoveLSM's random write performance and L_0-L_1 compactions. *Comparing to RocksDB in Fig. 2, the average period of write stalls is increased.*

operations to LSM-trees are stalled [18]. (2) Flush stalls: if L_0 has too many SSTables and reaches a size limit, flushes to 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 Fig. 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 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 Section 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 blocks foreground requests and makes 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. Fig. 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 90th, 99th, and 99.9th 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.

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 ($AF = 10$ [2]), 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. And, 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 [18] exploits NVMs to deliver high throughput for systems with DRAM-NVM-SSD storage, as shown in Fig. 1b. 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 longer compaction latencies and more severe write stalls. The worse write stall problem magnifies performance variability and hurts user experiences further.

We evaluate NoveLSM (with 8 GB NVM) by randomly writing the same 80 GB dataset. Test results in Fig. 4 show that NoveLSM reduces the overall loading time by $1.7\times$ compared to RocksDB (Fig. 2). However, the period of write stalls is significantly longer. This is because the amount of data involved in each L_0-L_1 compaction is over 15 GB, which is $4.86\times$ larger than that of RocksDB. A write stall starts when compaction threads call for the L_0-L_1 compaction. Then, the compaction waits until other pending compactions with a higher priority 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.

2.4.4 Matrixkv

Motivated by these observed challenging issues, MatrixKV [45] is designed to provide a stable low-latency KV store via intelligent use of NVM. As shown in Fig. 5, MatrixKV still manages MemTable and immutable MemTable in DRAM and put WAL in NVM. One may think putting MemTable and immutable MemTable in NVM can further eliminate

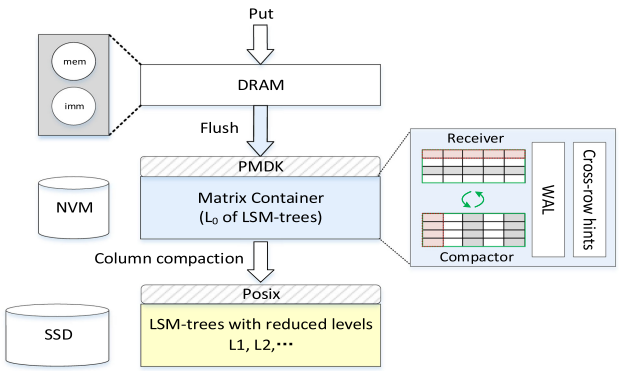


Fig. 5. The structure of MatrixKV. *MatrixKV uses matrix container to manage L_0 in NVM, and reduces the number of levels in LSM-Tree on SSD.*

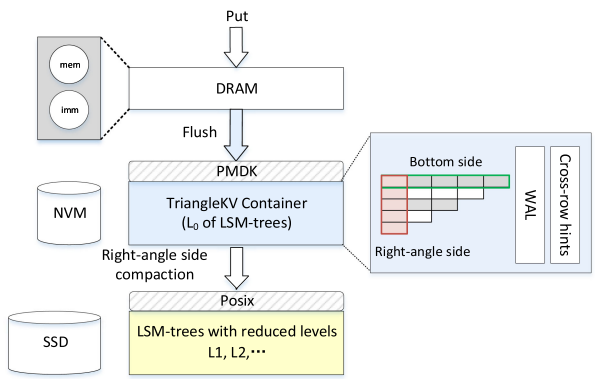


Fig. 7. TriangleKV's architectural overview. *TriangleKV is KV store for systems consisting of DRAM, NVMs, and SSDs.*

WAL overhead. But in our test, this method performed worse due to the following two reasons: (1) As illustrated in Section 2.1, compared to DRAM, NVM still has 5 times lower bandwidth and 3 times higher read latency. Using NVM to store MemTables and immutable MemTables may degrade read and write performance. (2) Inserting data into MemTable needs to look up the skip-list, and involves multiple pointer modifications, these operations are also very expensive in NVM [46].

Besides, MatrixKV also lifts L_0 of LSM-Tree to NVM. Placing L0 on NVM has the following advantages: (1) NVM has higher bandwidth and much lower latency than that SSD, so flushing immutable MemTable to NVM is much faster than flushing it to SSD, which reduces insert stalls. And it also isolates the interaction between L_0-L_1 compaction and flush operations, because flush operations no longer compete with L_0-L_1 compaction for SSD bandwidth, reducing write stalls. (2) NVM is byte-addressable, which means L_0-L_1 compaction can be done at a very fine granularity, and the space can be released immediately after every fine-grained compaction. Putting the L_0 on SSD can't achieve that because the SSTable on SSD can't be deleted until the entire file is compacted.

MatrixKV manages L_0 using a matrix container which comprises one receiver and one compactor. Immutable memtables are first flushed to the receiver and, when the receiver reaches its size limit, it is transformed into a compactor and a new receiver is generated to absorb flushes.

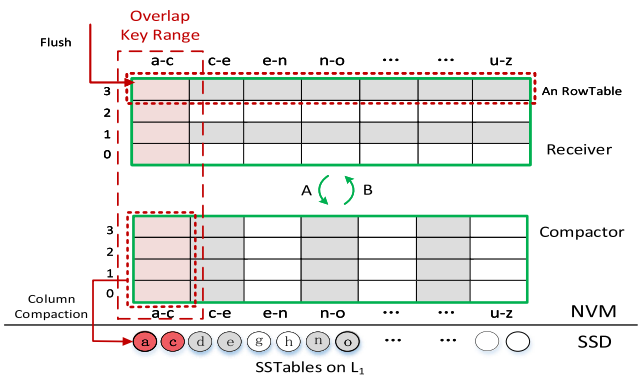


Fig. 6. Problem of matrix container. *The receiver and compactor have overlapping key ranges. The column compaction only merges key ranges in the compactor, causing inefficiency.*

Data in the compactor is compacted to L_1 in SSDs through a fine-granularity column compaction that compacts only a small subset of the data in a specific key range each time, which significantly reduces write stalls.

However, the column compaction is only performed for the compactor, and the key ranges overlapping with the compactor in the receiver cannot participate in the compaction process. For example, as shown in Fig. 6, the key range a-c in the compactor is first compacted to the L_1 by the column compaction, and the same key range in the receiver is still in L_0. When the receiver is transformed into a compactor, the key range a-c is compacted again, which means the key range a-c was written twice in L_1, causing additional write amplification.

3 TriangleKV Design

In this section, we present TriangleKV, an LSM-tree based key-value store for systems with multi-tier DRAM-NVM-SSD storage. TriangleKV builds and improves on MatrixKV by replacing the latter's matrix data structure with a more efficient management structure in L_0, a triangular data structure. TriangleKV has the following four key enabling techniques, i.e., triangle container in NVMs to manage the L_0 of LSM-trees (Section 3.1), right-angle side compaction for L_0 and L_1 (Section 3.2), level reduction of LSM-tree (Section 3.3), and cross-row hint search (Section 3.4). Fig. 7 shows the overall architecture of TriangleKV, where, from top to bottom, (1) DRAM batches writes with MemTables, (2) MemTables are flushed to L_0 that is stored and managed by a triangle container in NVMs, (3) data in L_0 are compacted to L_1 in SSDs through right-angle side compactions, and (4) SSDs store the remaining levels of a flattened LSM-tree.

3.1 Triangle Container

LSM-tree requires all-to-all compactions for L_0 and L_1 because L_0 has overlapping key ranges among SSTables. The substantial amount of I/Os due to L_0-L_1 compactions are identified as the root cause of write stalls, as demonstrated in Section 2.4. NoveLSM [18] 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 the lower-speed SSDs. Hence, the design principle of an LSM-tree based KV store with minimum or no write stalls is to

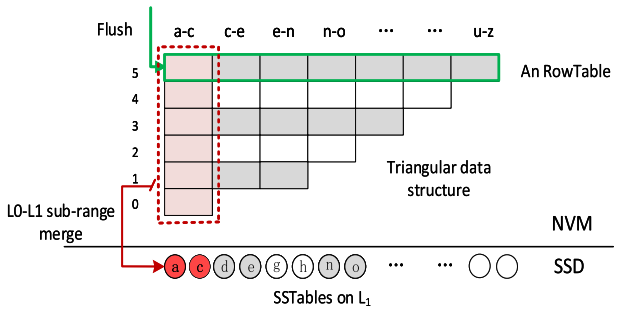


Fig. 8. Structure of triangle container. *The triangular data structure absorbs flushed MemTables, one per row. Each row is reorganized as a RowTable. The L_0-L_1 sub-range merge operation merges L_0 with L_1 in fine-grained key ranges, one range at a time, referred to as* right-angle side compaction.

reduce the granularity of L_0 and L_1 compaction and the number of I/Os on SSDs.

Based on this design principle, TriangleKV elevates L_0 from SSDs to NVMs and reorganizes L_0 into a triangle container to exploit the byte-addressability and high-speed random accesses of NVMs. Triangle container is a data management structure for L_0 of LSM-trees. Fig. 8 shows the organization of a triangle container.

Triangular Data Structure. In the triangle container, the Immutable MemTable flushed from DRAM is stacked in NVM by "rows". Each such MemTable is reorganized as a *RowTable* according to the key order. RowTables are stacked up as the bottom side of the triangular data structure with an increasing sequence number, i.e., from 0 to n. The size of the triangular data structure starts with one RowTable. When the triangular data structure size reaches its size limit (e.g., 60% of the amount of NVM), a sub-range merge operation starts with L_0-L_1. It is the data in the vertical right-angle side of the triangular data structure that participate in each such sub-range merge operation, that is, the sub-key value range that has not been merged for the longest time. The sub-range merging operation of L_0-L_1 is performed synchronously with the flush operation of DRAM, so that the data structure inside the NVM is maintained in a triangular balanced state.

Fig. 9 shows an example of the dynamic changes of the triangular data structure during the flush and L_0-L_1 sub-range merge processes. There are a total of 6 RowTables in Fig. 9a, with serial numbers from 0 to 5. Assuming that the total key value space is a-f, the key value range being merged is a. After the L_0-L_1 layer merges the key value range a, all the data in this range is written to the SSD. Therefore, each RowTable in the NVM no longer contains

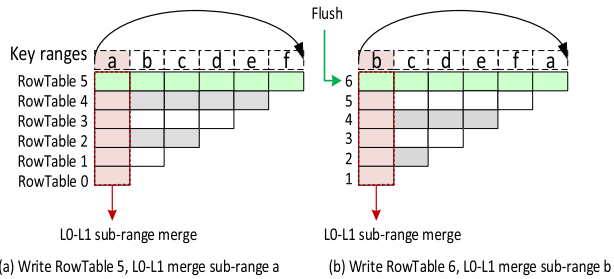


Fig. 9. Dynamic changes of triangular data structure during flush and L_0-L_1 sub-range merge process.

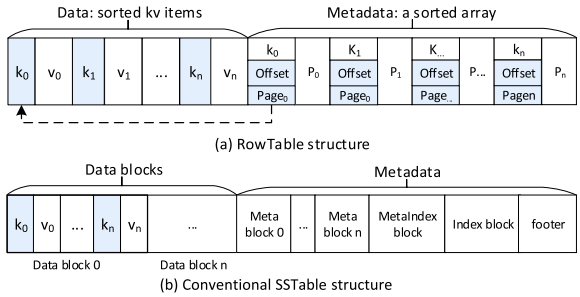


Fig. 10. RowTable and conventional SSTable.

the data of the key value range a. As shown in Fig. 9b, when the DRAM flushes another Immutable MemTable, a RowTable 6 is added to the NVM, and the key value range of RowTable 6 is a-f. At this time, only RowTable 6 contains the key value range a. The L_0-L_1 sub-ranges merge is rotated to the key value range b, so that the data in the range b in each RowTable will all be merged into the L_1 layer.

In the two sets of DRAM Flush and L_0-L_1 sub-range merge operations illustrated in Fig. 9, RowTable is flushed from DRAM to NVM, and is continuously appended to the triangular data structure as it's bottom side; the L_0-L_1 sub-range merge operation is rotated through the key range, and is always performed in the sub-key range that has not been merged for the longest time, eliminating the longest vertical right-angle side of the triangle. Therefore, the L_0 layer in the NVM forms a dynamic triangle, which is gives rise to the name "triangular data structure". In what we follows we introduce RowTable, L_0-L_1 sub-range merging algorithm, and the space management scheme for the triangular data structure.

RowTable. Fig. 10a shows the RowTable structure consisting of data and metadata. To construct a RowTable, we first serialize KV items from the immutable MemTable in the 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. 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 triangle container. The cross-row hint search will be discussed in Section 3.4. Fig. 10b 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.

L_0-L_1 Sub-Range Merge. One main function of the triangle container is to select and merge data from L_0 to L_1 in SSDs at a fine granularity. Leveraging the byte addressability of NVMs and our proposed RowTables, TriangleKV 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 right-angle side compaction (detailed in Section 3.2). In

the triangle container, KV items are managed by logical triangular data structure. A right-angle side of the triangular data structure is a subset of key spaces with a limited amount of data, which is the basic unit of the triangle container in right-angle side compactions. Specifically, KV items from different RowTables that fall in the key range of a right-angle side compaction logically constitute a right-angle side. The amount of these KV items is the size of a right-angle side, which is not strictly fixed but limited by a threshold determined by the size of the right-angle compaction.

Space Management. After compacting a right-angle side, the NVM space occupied by the right-angle side is freed. To manage those freed spaces, we simply apply the paging algorithm [29]. The NVM pages freed after right-angle compactions are added to the free list as a group of page-sized units. To store incoming RowTables in the triangle container, we apply free pages from the free list. The 8 GB triangle 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 space management includes a free list and an array list that maps the RowTable number to its corresponding metadata pages, which only occupies a total space of 24 KB on NVMs.

It is worth noting that in the triangle container, while the right-angle side of the triangular data structure are being compacted, the triangular data structure can continue accepting flushed MemTables from DRAM as its bottom side simultaneously. By freeing the NVM space one right-angle side at a time, TriangleKV ends the write stalls forced by merging the entire L_0 with all of L_1.

3.2 Right-Angle Side Compaction

Right-angle side compaction is a fine-grained L_0-L_1 compaction that each time compacts only a right-angle side, i.e., a small subset of the data in a specific key range. Thus, right-angle side compaction is able to significantly reduce write stalls. The main workflow of right-angle side compaction can be described in the following seven steps. (1) TriangleKV 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 a smallest key and a largest key, the smallest key and largest key of all the SSTables in L_1 form a sorted key list. Every two adjacent keys represent a key range, 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) Right-angle side 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 triangle container, victim KV items within the compaction key range are picked concurrently in multiple RowTables. Specifically, assuming N RowTables in the triangle container, 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 right-angle side compaction. (5) Then a right-angle side in the triangle container is logically formed, i.e., KV items in N RowTables that fall in the compaction key range make up a logic right-angle side. (6) Data in the right-angle side 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. Right-angle side compaction continues between the next key range of L_1 and the next right-angle side in the triangle container. In L_1, the key ranges of right-side compaction rotate in the whole key space to keep LSM-trees balanced.

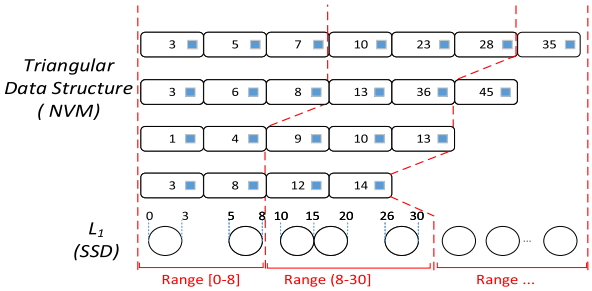


Fig. 11. Right-angle side compaction: an example. *There are 4 RowTables in the triangle data structure. Each circle represents an SSTable on L_1. Right-angle sides are logically divided (red dashed lines) according to the key range of compaction.*

We show an example of right-angle side compaction in Fig. 11. First, TriangleKV 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 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 logic right-angle side is formed for the compaction. The first right-angle side compaction compacts the right-angle side at the key range of 0-8 with the first two SSTables in L_1.

In general, right-angle side compaction first selects a specific key range from L_1, and then compacts with the right-angle side in the compactor that shares the same key range. Comparing to the original all-to-all compaction between L_0 and L_1, right-angle side compaction compacts at a much smaller key range with a limited amount of data. Consequently, the fine-grained right-angle 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 increases 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 in the LSM-tree. Hence, the other design principle of TriangleKV is to reduce the depth of LSM-trees to mitigate WA. TriangleKV 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

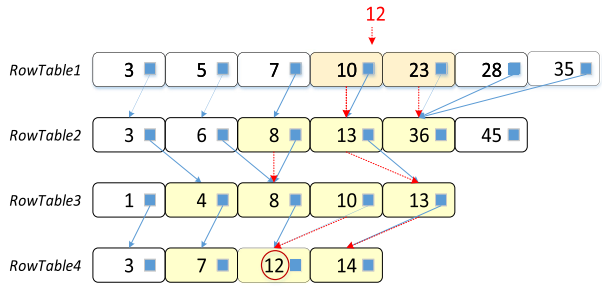


Fig. 12. Cross-row hint search. *This figure shows an example of searching the target key ($k = 12$) with forward pointers of each array element.*

levels, the WA of compacting an SSTable to the next level remains the same AF but the overall WA is reduced due to fewer levels.

Flattening conventional LSM-trees with larger level widths brings two negative effects. First, since the enlarged L_0 has more SSTables that overlap with key ranges, the amount of data in the L_0-L_1 compaction increases significantly, which not only adds the compaction overhead but also lengthens the duration of write stalls. Second, the search efficiency decreases as we must traverse a larger unsorted L_0. TriangleKV does not suffer from the first problem since right-angle side compaction makes the L_0-L_1 compaction always in fine granularity. The amount of data involved in right-angle side compaction is largely independent of the level width as a right-angle side contains a limited amount of data. The only drawback comes from the decreased efficiency of searching the larger L_0, which potentially reduces read efficiency.

3.4 Cross-Row Hint Search

In this section, we discuss solutions for improving the read efficiency in the triangle container. In the L_0 of TriangleKV, 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 TriangleKV, we build cross-row hint searches.

Constructing Cross-Row Hints. When we build a RowTable in the triangle container, we add a forward pointer for each element in the sorted array of metadata (Fig. 10). 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 [12], [47], [48]. 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 Triangle 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 Fig. 12. The blue arrows show the forward pointers providing cross-row hints. Suppose we want to fetch a target key $k = 12$ in the triangle 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.

TABLE 1
FIO 4 KB Read and Write Bandwidth

	SSDSC2BB800G7	Optane DC PMM
Rnd write	168 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

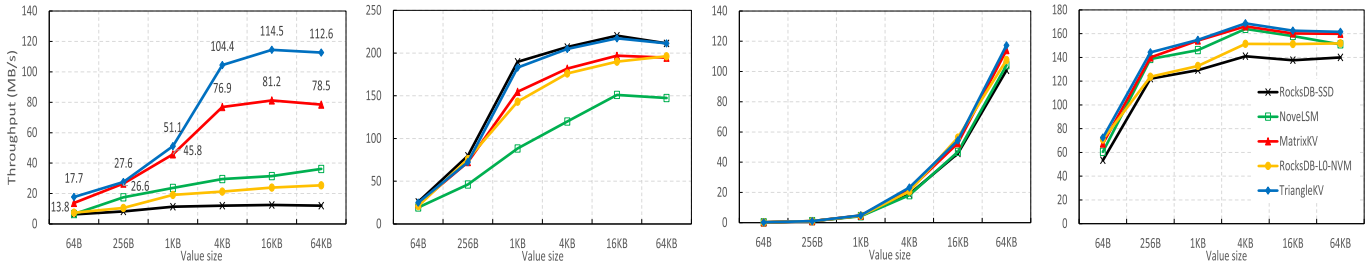
4 Implementation

We implement TriangleKV based on the popular KV engine RocksDB [2] from Facebook. The LOC on top of RocksDB is 4552 lines. The implementation of TriangleKV will be open source. As shown in Fig. 7, TriangleKV accesses NVMs via the PMDK library and accesses SSDs via the POSIX API. The persistent memory development kit (PMDK) [34], [49] 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 triangle container, forming the bottom-side of the triangular data structure. (4) Data in the triangular data structure is right-angle side compacted with SSTables in L_1 one by one. (5) In SSDs, SSTables are merged to higher levels via conventional compactions as RocksDB does. Compared to RocksDB, TriangleKV is completely different from step 3 through step 4.

Read. TriangleKV 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 search in different storage devices [18].

Consistency. Data structures in NVM must avoid inconsistency caused by system failures [37], [50], [51], [52], [53]. For TriangleKV, writes/updates for NVM only happen in two processes, flush and right-angle side compaction. For flush, immutable MemTables flushed from DRAM are organized as RowTables and written to NVM in rows. If a failure



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

Fig. 13. Performance on Micro-benchmarks with different value sizes.

occurs in the middle of writing a RowTable, TriangleKV can re-process all the transactions that were recorded in the write-ahead log. For right-angle side compaction, TriangleKV needs to update the state of RowTables after each right-angle side compaction. To achieve consistency and reliability with low overhead, TriangleKV 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. TriangleKV 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 size of metadata, etc. TriangleKV uses lazy deletion to guarantee that stale right-angle sides invalidated by right-angle side compaction are not deleted until a consistent new version is completed.

5 Evaluation

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

5.1 Experiment Setup

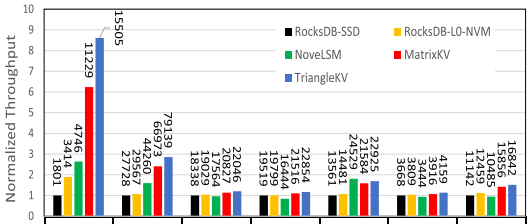
All experiments are run on a test machine with two Genuine Intel(R) 2.20 GHz 24-core processors and 32 GB of memory. The kernel version is 64-bit Linux 4.13.9 and the operating system in use is Fedora 27. The experiments use two storage devices, an 800 GB Intel SSDSC2BB800G7 SSD and 256 GB NVMs of two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

We mainly compare TriangleKV with MatrixKV, NoveLSM and RocksDB (including RocksDB-SSD and RocksDB-L0-NVM). RocksDB-SSD represents the conventional RocksDB on a DRAM-SSD hierarchy. The other four KV stores are for systems with DRAM-NVM-SSD storage. Their NVM in use is 8 GB, being consistent with the experiments in Nove-LSM's paper. RocksDB-L0-NVM simply enlarges L_0 into 8 GB and stores it in NVM. TriangleKV stores the reorganized 8 GB L_0 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 TriangleKV achieves system performance improvement with an 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 five key-value stores using db_bench, the micro-benchmark released with RocksDB. Then, we evaluate the performance of each KV store with the YCSB macro-benchmarks [54].

Write Performance. We evaluate the random write performance by inserting KV items totaling 80 GB in a uniformly distributed random order. Fig. 13a shows the random write throughput of five 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, NoveLSM and MatrixKV as baselines of our evaluation. TriangleKV improves random write throughput over RocksDB-L0-NVM, NoveLSM and MatrixKV in all value sizes. Taking the commonly used value size of 4 KB as an example, TriangleKV outperforms RocksDB-L0-NVM, NoveLSM and MatrixKV by $4.9\times$, $3.5\times$, $1.4\times$ respectively. RocksDB-L0-NVM delivers relatively poor performance since simply putting L_0 in NVM does not change write stalls and WA. NoveLSM slightly reduces WA by fewer compactions since its large mutable MemTable in NVM handles a portion of update requests. However, for both of them, the root causes of write stalls and WA remain unaddressed, i.e., the large amount of data involved in L_0-L_1 compaction and the deepened depth of LSM-trees. NoveLSM actually exacerbates write stalls since the enlarged MemTables flushed from NVM significantly increase the amount of data processed in L_0-L_1 compaction. MatrixKV reduces write stalls by using fine-grained column compaction and reduces WA by reducing the number of levels of LSM-tree, thus it has better performance than of RocksDB-L0-NVM and NoveLSM. However, since the column compaction only compact data in the compactor, MatrixKV has additional redundant writes, so its performance is worse than TriangleKV. TriangleKV benefits less from small KV items, because smaller KV size means larger number for KV items, at which point the CPU overhead of compactions becomes non-negligible.



	Load	A	B	C	D	E	F
Inserts	100%	0%	0%	0%	5%	5%	0%
Updates	0%	50%	5%	0%	0%	0%	0%
Reads	0%	50%	95%	100%	95%	0%	50%
Rangequery	0%	0%	0%	0%	0%	95%	0%
RMW	0%	0%	0%	0%	0%	0%	50%
Distribution	Uniform	Zipfian	Zipfian	Zipfian	Latest	Zipfian	Zipfian

Fig. 14. 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.* .

We evaluate sequential write performance by inserting a total of 80 GB KV items in sequential order. Fig. 13b shows that the five KV stores exhibit similar sequential write performances, because sequential writes do not trigger compaction. RocksDB-SSD performs the best since other KV stores have an NVM tier, requiring data migration from NVMs to SSDs. TriangleKV has similar performance to RocksDB-SSD, since TriangleKV also directly writes the data to the SSD when it judges that the workload is sequential write.

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. The test results in Fig. 13c indicate that the five KV stores have similar random read throughput. Fig. 13d shows the test results of sequential reads. Since a balanced tree is well-sorted in each level (from L_1 to L_n), the five KV stores exhibit almost the same sequential read throughput. The design of TriangleKV does not degrade read performance and even has a slight advantage in random reads for two reasons. First, the cross-row hint search reduces the search overhead of the enlarged L_0. Second, TriangleKV has fewer LSM-tree levels, resulting in less search overhead on SSDs.

Macro-Benchmarks. Now we evaluate five KV stores with YCSB [54], a widely used macro-benchmark suite delivered by Yahoo!. We first write an 80 GB dataset with 4 KB values for loading, then evaluate workload A-F with one million KV items respectively. From the test results shown in Fig. 14, we draw three main conclusions. First, TriangleKV benefits the most from write/load dominated workloads, i.e., load, and workload A and F. TriangleKV is $4.54\times$, $3.27\times$ and $1.38\times$ faster than RocksDB-L0-NVM, NoveLSM and MatrixKV under these load workloads (i.e., random write). Second, TriangleKV maintains adequate performance over read-dominated workloads, i.e., workloads B to E. Third, NoveLSM, MatrixKV and TriangleKV behave better on workload D due to the latest distribution, where they both hit more in NVMs and thus TriangleKV can 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 [13], i.e., using the YCSB-A workload and setting request arrival rate at around 20 K requests/s. Table 2 shows the average, 90th, 99th, and 99.9th percentile latencies of five key-value stores. TriangleKV significantly reduces latencies in all cases. The 99th percentile latency of TriangleKV is $29\times$, $5.5\times$, and $2.1\times$ lower than RocksDB-SSD, NoveLSM, and RocksDB-L0-NVM respectively. TriangleKV and MatrixKV have similar performance. The test results demonstrate that by reducing write stalls and WA, TriangleKV improves the quality of user experience with much lower tail latencies.

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
TriangleKV	**254**	**227**	**379**	**635**

5.3 Performance Gain Analysis

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

5.3.1 Main Challenges

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

Write Stalls. We record the random write throughput for the five KV stores in every ten seconds (similar to Figs. 2 and 4) to visualize write stalls. Fig. 15 shows the performance variation of five KV stores. From the test results, we draw the following observations. (1) TriangleKV is the fastest for randomly writing the same 80 GB dataset (as demonstrated in Section 5.2). (2) MatrixKV is slower than TriangleKV since the former's multi-component structure causes more writes. (3) NoveLSM is faster than RocksDB since the large and mutable MemTable in NVM contributes to serving more write requests at a higher speed. (4) 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. (5) TriangleKV achieves the most stable performance. The reason is that we reduce write stalls by the fine-grained right-angle side compaction which guarantees a small amount of data processed in each L_0-L_1 compaction.

Write Amplification. We measure the WA of five systems on the same experiment of randomly writing 80 GB dataset. Fig. 16 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 TriangleKV, MatrixKV, NoveLSM, and RocksDB-L0-NVM are $3.26\times$, $2.56\times$, $1.83\times$, and $1.99\times$ lower than RocksDB-SSD respectively. Write

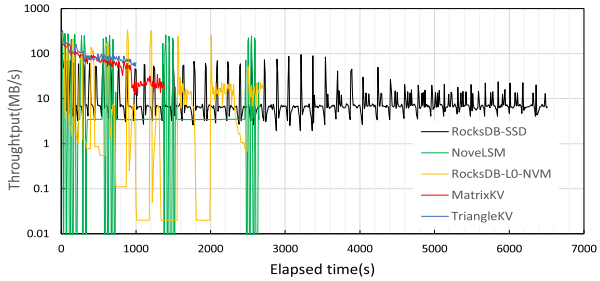


Fig. 15. Throughput fluctuation as a function of time. *The random write performance fluctuates where the troughs on curves signify the occurrences of possible write stalls.* .

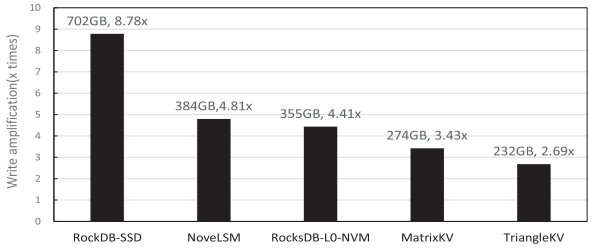


Fig. 16. Write amplification of 80 GB random writes. *The numbers on each bar show the amount of data written to SSDs and the WA ratio respectively.*

amplification is mainly decided by compactions. The reasons for TriangleKV's smallest WA are threefold. First, it reduces the amount of data processed per L_0-L_1 compaction with right-angle side compactions. Second, it reduces the number of compactions by lowering the depth of LSM-trees. Third, it reduces writes by using a dynamic triangular data structure.

5.3.2 TriangleKV Enabling Techniques

Right-Angle Side Compaction. To demonstrate the efficiency of right-angle side compaction, we record the amount of data involved, the starting and ending times of every L_0-L_1 compaction for five KV stores during the 80 GB random write experiment. As shown in Fig. 17, TriangleKV has the smallest volume of compaction data and the shortest compaction duration. As a result, the fine-grained right-angle side compaction occupies less bandwidth of SSDs, has only a negligible influence on foreground requests, and finally significantly reduces write stalls. MatrixKV also has the same advantages, but the column compaction only compact data in the compactor, so more writes are required.

Overall Compaction Efficiency. We further record the overall compaction behaviors of five KV stores by recording the amount of data for every compaction during the random write experiment. From the test results shown in Fig. 18, we draw five observations. First, TriangleKV has the smallest number of compactions, attributed to the dynamic triangular data structure and reduced LSM-tree depth. Second, TriangleKV reduces 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

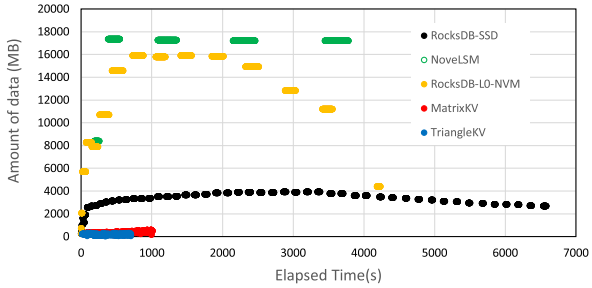


Fig. 17. The L_0-L_1 compaction. *Each line segment indicates an L_0-L_1 compaction. The y-axis shows the amount of data involved in the compaction and the length along x-axis shows the duration of the compaction.* .

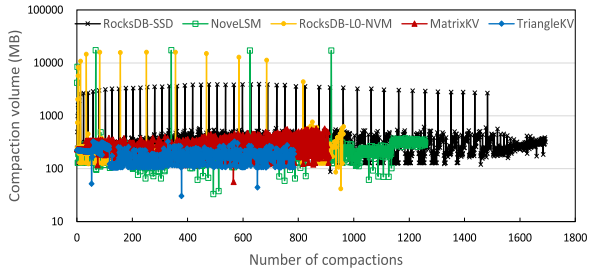
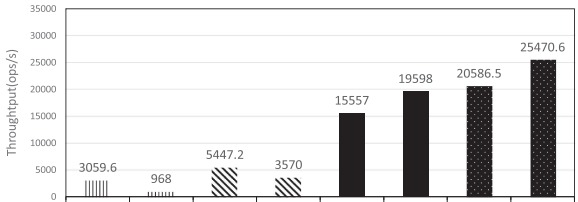


Fig. 18. Compaction analysis. *This figure shows the amount of data of every individual compaction during the 80 GB random write.*



	RocksDB-SSD		RocksDB-L0-NVM		MatrixKV		TriangleKV	
	256M	8G	256M	8G	256M	8G	256M	8G
L0	0	0	0	0	0	0	0	0
L1	0.19	7.95	0.23	8.01	0.24	7.99	0.24	7.99
L2	2.47	43.92	2.45	44.67	2.45	41.04	2.44	44.95
L3	24.97		24.99		25		24.96	
L4	23.72		21.9		21.85		22.28	

Fig. 19. Reducing LSM-tree depth. *The y-axis shows random write throughputs of RocksDB, MatrixKV and TriangleKV when L_1 is 256 MB/8 GB. The table below shows the data distribution among levels (in GB).*

and RocksDB stems from the L_0-L_1 compaction. Fifth, TriangleKV has less compactions than MatrixKV, since TriangleKV reduces additional writes, the total amount of compaction data is less than that of MatrixKV, while TriangleKV and MatrixKV have the same amount of data in each compaction, so the number of compactions of TriangleKV is smaller than that of MatrixKV.

Reducing LSM-Tree Depth. To evaluate the technique of flattening LSM-trees, we change level sizes for both RocksDB, MatrixKV and TriangleKV. The first configuration is $L_1 = 256\ MB$ (the default L_1 size of RocksDB). The second configuration is $L_1 = 8\ GB$. The following levels grow at the same ratio of AF = 10. Fig. 19 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, MatrixKV and TriangleKV reduce the number of levels by enlarging level sizes, i.e., from 5 to 3. However, they exert opposite influences on system performance. For RocksDB-SSD and RocksDB-L0-NVM, their random write throughputs are reduced by $3\times$ and $1.5\times$

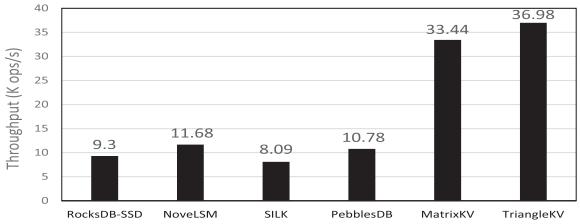


Fig. 20. Throughput on NVM based KV stores.

respectively when 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 TriangleKV, the throughput increases 24% since the fine-granularity right-angle side compaction is independent of level sizes. Furthermore, the TriangleKV with 256 MB L_1 shows the performance improvement of only addressing write stalls. MatrixKV exhibits similar results as TriangleKV.

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 level in NVMs for TriangleKV and RocksDB-L0-NVM. Then we search for one million KV items from NVMs in uniformly random order. The random read throughput of RocksDB-L0-NVM and TriangleKV are 9 MB/s and 162.6 MB/s respectively. Hence, compared to simply placing L_0 in NVMs, the cross-row hint search improves the read efficiency by 18 times.

5.4 Extended Comparisons on NVMs

To further verify that TriangleKV'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, MatrixKV and TriangleKV, where DRAM stores MemTables, and all other components are stored on NVMs.

Throughput. Fig. 20 shows the performance for randomly writing an 80 GB dataset. TriangleKV achieves the best performance among all KV stores. It demonstrates that the enabling techniques of TriangleKV 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 also evaluated with YCSB-A workload as in Section 5.2. Since NVM has a significantly better performance than SSDs, we speed up the requests from clients (60 K requests/s). Test results in Table 3 show that with the persistent storage of NVMs most KV stores provide adequate tail latencies. TriangleKV achieves even shorter tail latency than SILK.

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
TriangleKV	197	291	399	523

6 Conclusion

In this paper, we present TriangleKV, a stable low-latency key-value store based on LSM-trees. TriangleKV is designed for systems with multi-tier DRAM-NVM-SSD storage. By lifting the L_0 to NVM, managing it with the triangle container, and compacting L_0 and L_1 with the fine granularity right-angle-side compaction, TriangleKV reduces write stalls. By flattening the LSM-trees, TriangleKV mitigates write amplification. TriangleKV also guarantees adequate read performance with cross-row hint searches. TriangleKV builds and improves on MatrixKV to reduce unnecessary writes by using a dynamic triangular data structure. Evaluation results demonstrate that TriangleKV significantly reduces system stalls and achieves much better system performance than RocksDB and NoveLSM. TriangleKV will be *open source*.

References

[1] P. ONeil, E. Cheng, D. Gawlick, and E. ONeil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.
[2] Facebook, Rocksdb, a persistent key-value store for fast storage enviroments, 2019. [Online]. Available: http://rocksdb.org/
[3] S. Ghemawat and J. Dean, "Leveldb," 2016. [Online]. Available: https://github.com/Level/leveldown/issues/298
[4] T. Harter *et al.*, "Analysis of HDFS under HBase: A facebook messages case study," in *Proc. 12th USENIX Conf. File Storage Technol.*, 2014, pp. 199–212.
[5] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," in *Proc. 3rd ACM SIGOPS Int. Workshop Large Scale Distrib. Syst. Middleware*, 2009, pp. 35–40.
[6] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "FAWN: A fast array of wimpy nodes," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Princ.*, 2009, pp. 1–14.
[7] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "SILT: A memory-efficient, high-performance key-value store," in *Proc. 23th ACM Symp. Operating Syst. Princ.*, 2011, pp. 1–13.
[8] 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.
[9] J. Arulraj and A. Pavlo, "How to build a non-volatile memory database management system," in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 1753–1758.
[10] A. Eisenman *et al.*, "Reducing DRAM footprint with NVM in facebook," in *Proc. 13th EuroSys Conf.*, 2018, pp. 101–109.
[11] D. Terry, "Transactions and scalability in cloud databasescan't we have both?," Boston, MA, USA: USENIX Assoc., 2019.
[12] R. Sears and R. Ramakrishnan, "bLSM: A general purpose log structured merge tree," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2012, pp. 217–228.
[13] 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.
[14] B. Debnath, S. Sengupta, and J. Li, "SkimpyStash: Ram space skimpy key-value store on flash-based storage," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2011, pp. 25–36.
[15] B. O. Maria *et al.*, "Triad: Creating synergies between memory, disk and log in log structured key-value stores," in *Proc. USENIX Annu. Tech. Conf.*, 2017, pp. 363–375.
[16] L. Marmol, S. Sundararaman, N. Talagala, and R. Rangaswami, "Nvmkv: A scalable, lightweight, ftl-aware key-value store," in *Proc. USENIX Annu. Tech. Conf.*, 2015, pp. 207–219.
[17] L. Lanyue, P. T. Sankaranarayana, A.-D. A. C., and A.-D. R. H., "WiscKey: Separating keys from values in SSD-conscious storage," in *Proc. 14th USENIX Conf. File Storage Technol.*, 2016, pp. 133–148.
[18] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Redesigning LSMs for nonvolatile memory with NoveLSM," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 993–1005.
[19] S. Raoux *et al.*, "Phase-change random access memory: A scalable technology," *IBM J. Res. Dev.*, vol. 52, no. 4.5, pp. 465–479, 2008.

[20] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia, P. Ranganathan, and N. Binkert, "Consistent, durable, and safe memory management for byte-addressable non volatile main memory," in *Proc. First ACM SIGOPS Conf. Timely Results Operating Syst.*, 2013, pp. 1–17.
[21] M. S. Bhaskaran, J. Xu, and S. Swanson, "Bankshot: Caching slow storage in fast non-volatile memory," in *Proc. 1st Workshop Interact. NVM/FLASH Operating Syst. Workloads*, 2013, pp. 73–81.
[22] W.-H. Kim, B. Nam, D. Park, and Y. Won, "Resolving journaling of journal anomaly in android I/O: Multi-version B-tree with lazy split," in *Proc. 12th USENIX Conf. File Storage Technol.*, 2014, pp. 273–285.
[23] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature*, vol. 453, no. 7191, 2008, Art. no. 80.
[24] J. Izraelevitz *et al.*, "Basic performance measurements of the intel optane DC persistent memory module," 2019, *arXiv:1903.05714*.
[25] A. Driskill-Smith, "Latest advances and future prospects of STT-RAM," in *Proc. Non-Volatile Memories Workshop*, 2010, pp. 11–13.
[26] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "NV-Tree: Reducing consistency cost for NVM-based single level systems," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 167–181.
[27] G. Copeland, T. W. Keller, R. Krishnamurthy, and M. G. Smith, "The case for safe RAM," in *Proc. 15th Int. Conf. Very Large Data Bases*, 1989, pp. 327–335.
[28] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy, "Overview of candidate device technologies for storage-class memory," *IBM J. Res. Dev.*, vol. 52, no. 4.5, pp. 449–464, 2008.
[29] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*. Hunter Hl Madison, WI, USA: Arpaci-Dusseau Books Wisconsin, 2014, vol. 151.
[30] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson, "Moneta: A high-performance storage array architecture for next-generation, non-volatile memories," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2010, pp. 385–395.
[31] J. Condit *et al.*, "Better I/O through byte-addressable, persistent memory," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Princ.*, 2009, pp. 133–146.
[32] S. R. Dulloor *et al.*, "System software for persistent memory," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, Art. no. 15.
[33] J. Li, A. Pavlo, and S. Dong, NVMRocks: RocksDB on non-volatile memory systems, 2017. [Online]. Available: http://istc-bigdata.org/index.php/nvmrocks-rocksdb-on-non-volatile-memory-systems/
[34] Persistent memory development kit, 2019. [Online]. Available: https://github.com/pmem/pmdk
[35] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in byte-addressable persistent B-tree," in *Proc. 16th USENIX Conf. File Storage Technol.*, 2018, pp. 187–200.
[36] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 461–476.
[37] M. Wu and W. Zwaenepoel, "envy: A non-volatile, main memory storage system," in *Proc. 6th Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 1994, pp. 86–97.
[38] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y. ri Choi, "SLM-DB: Single-level key-value store with persistent memory," in *Proc. 17th USENIX Conf. File Storage Technol.*, 2019, pp. 191–205.
[39] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu, "Phase change memory in enterprise storage systems: Silver bullet or snake oil?," *ACM SIGOPS Operating Syst. Rev.*, vol. 48, no. 1, pp. 82–89, 2014.
[40] H. Jagadish, P. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti, "Incremental organization for data recording and warehousing," in *Proc. 23rd Int. Conf. Very Large Data Bases*, 1997, pp. 16–25.
[41] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel, "KVell: The design and implementation of a fast persistent key-value store," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, 2019, pp. 447–461.
[42] T. Yao *et al.*, "A light-weight compaction tree to reduce I/O amplification toward efficient key-value stores," in *Proc. IEEE 33rd Symp. Massive Storage Syst. Technol.*, 2017, pp. 1–13.
[43] X. Wu, Y. Xu, Z. Shao, and S. Jiang, "LSM-Trie: An LSM-tree-based ultra-large key- value store for small data," in *Proc. USENIX Annu. Tech. Conf.* (ATC 15), 2015.
[44] P. Shetty, R. P. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok, "Building workload-independent storage with vt-trees," in *Proc. 11th USENIX Conf. File Storage Technol.*, 2013, pp. 17–30.
[45] T. Yao *et al.*, "MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 17–31.
[46] Y. Chen, Y. Lu, K. Fang, Q. Wang, and J. Shu, "uTree: A persistent B -tree with low tail latency," *Proc. VLDB Endowment*, vol. 13, no. 12, pp. 2634–2648, 2020.
[47] B. Chazelle and L. J. Guibas, "Fractional cascading: I. a data structuring technique," *Algorithmica*, vol. 1, no. 1–4, pp. 133–162, 1986.
[48] B. Kuszmaul, "How TokuDB fractal tree indexes work," TokuTek, Lexington, MA, USA, Tech. Rep., 2010.
[49] J. Xu, J. Kim, A. Memaripour, and S. Swanson, "Finding and fixing performance pathologies in persistent memory software stacks," in *Proc. 24th Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2019, pp. 427–439.
[50] J. Coburn *et al.*, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *Proc. 16th Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2011, pp. 105–118.
[51] N. Cohen, D. T. Aksun, H. Avni, and J. R. Larus, "Fine-grain checkpointing with in-cache-line logging," in *Proc. 24th Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2019, pp. 441–454.
[52] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "High-performance transactions for persistent memories," *Proc. 21st Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2016, pp. 399–411.
[53] M. Liu *et al.*, "DudeTM: Building durable transactions with decoupling for persistent memory," in *Proc. 22nd Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2017, pp. 329–343.
[54] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. ACM Symp. Cloud Comput.*, 2010, pp. 143–154.

Chen Ding received the bachelor's degree in computer science and technology from the Huazhong University of Science and Technology, China, in 2019. He is currently working towards the PhD degree with the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, China. His research interests include emerging storage devices (i.e., non-volatile memory, SSD, etc.), key-value systems, and learned index.

Ting Yao received the bachelor's degree in the Internet of Things from the School of Computer Science and Technology, Northwest University, China, in 2015, and the PhD degrees in computer architecture from the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, China, in 2020. Her research interests include emerging storage devices (i.e., non-volatile memory, SMR, etc.), key-value systems, and computer architecture.

Hong Jiang (Fellow, IEEE) received the BSc and MASc degrees in computer engineering from the Huazhong University of Science and Technology, and the University of Toronto, respectively, and the PhD degree in computer science from the Texas A&M University. He is currently chair and Wendell H. Nedderman Endowed professor with the Computer Science and Engineering Department, University of Texas at Arlington. Prior to joining UTA, he served as a program director with National Science Foundation (2013–2015) and he was with University of Nebraska-Lincoln since 1991, where he was Willa Cather professor of computer science and engineering. He has graduated 17 PhD students and supervised 20 post-doctoral fellows and visiting scholars. His present research interests include computer architecture, computer storage systems and parallel I/O, high-performance computing, Big Data computing, and cloud and edge computing. He is a topic editor and an associate editor of the *IEEE Transactions on Computers*. He has more than 300 publications in major journals and international Conferences in these areas, including *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *Proceedings of IEEE*, *ACM Transactions on Architecture and Code Optimization*, *ACM TRANSACTIONS ON STORAGE*, USENIX ATC, FAST, EUROSYS, ISCA, MICRO, SOCC, LISA, SIGMETRICS, ICDE, DATE, ICDCS, IPDPS, MIDDLEWARE, OOPLAS, ECOOP, SC, ICS, HPDC, INFOCOM, ICPP, etc. He is a member of ACM.

Qiu Cui worked for PeaPod, and his main work in PingCAP is to participate in the design and development of open source NewSQL database TiDB from scratch, and now he is mainly responsible for PingCAP community and talent building and technical exchange and cooperation related matters.

Liu Tang the chief architect of PingCAP, has rich development experience in distributed, high performance and high availability, and is now working on TiDB, the next generation distributed database, and TiKV, the distributed storage. He is now working on the next generation distributed database TiDB and distributed storage TiKV, and is committed to providing a complete HTAP solution in the field of infrastructure database to free productivity. He is an open source enthusiast and author of the well-known open source software LedisDB, go-mysql, etc.

Yiwen Zhang received the bachelor's degree in computer science and technology from the Huazhong University of Science and Technology, China, in 2017. He is currently working towards the PhD degree with the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, China. His research interests include emerging storage devices (i.e., non-volatile memory, SSD, etc.), key-value systems, and file systems.

Jiguang Wan received the bachelor's degree in computer science from Zhengzhou University, China, in 1996, the MS and PhD degrees in computer science from Huazhong University of Science and Technology (HUST), China, in 2003 and 2007, respectively. He is currently a professor with the Wuhan National Laboratory for Optoelectronics (WNLO), Huazhong University of Science and Technology. His research interests include computer architecture, networked storage system, key-value storage system, parallel and distributed system.

Zhihu Tan received the BS, MS, and PhD degree in computer architecture from the Huazhong University of Science and Technology, China, in 1996, 1998, and 2008, respectively. He is currently working as a professor with the Key Laboratory of Data Storage System, Huazhong University of Science and Technology. His research interests include network storage, storage cache and disk I/O, high availability computing.