PartitionKV: Redesigning LSM-tree KV Stores on NVMs with Adaptive Partitioning for Reducing Write Stalls and Amplification

XINGYE HUANG* and JINYU WU*, Xidian University, China
XIAOFANG XIA†, JIANGTAO CUI†, and HUI LI, Xidian University, China
LIANG WANG, Northwestern Polytechnical University, China
FENG ZHANG, Renmin University of China, China

In write-intensive applications, the log-structured merge (LSM) trees are widely used as the basic index structure of key-value (KV) stores. Existing works integrate NVM into traditional DRAM-SSD architecture to improve write performance. However, these works still suffer from significant write stalls and amplification, mainly due to the inefficient L_0-L_1 compaction caused by the unordered nature of data in L_0 of LSM-trees. To address these issues, we propose PartitionKV, a novel LSM-tree based KV store designed for the DRAM-NVM-SSD storage architecture, which has three main design characteristics: (1) First, we design an ordered partition layer comprising multiple partitions to replace the Memtable components and L_0 of original LSM-trees. Incoming KVs are directly persisted into NVM Logs of designated partitions based upon keys. This design minimizes unnecessary data rewriting during compaction and can double as a write-ahead log, significantly reducing write amplification. (2) Second, we introduce an adaptive partitioning strategy that dynamically splits or merges partitions based on the number of overlapping SSTables, ensuring that an optimal amount of data is involved in each compaction. (3) Third, we propose a multi-threaded compaction strategy where multiple threads leverage two priority lists to efficiently coordinate concurrent data compaction between the partition layer and L_1. By effectively integrating these two strategies, PartitionKV accelerates NVM space release and reduces write stalls significantly. We implement PartitionKV based on RocksDB and conduct extensive experiments to evaluate its performance. Results show that PartitionKV achieves 3.63× and 4.06× higher random write throughput than FlatLSM and MatrixKV, respectively.

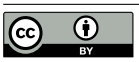
CCS Concepts: • **Information systems → Database management system engines**.

Additional Key Words and Phrases: LSM-tree, NVM, Partition, Write Stalls, Write Amplification

ACM Reference Format:
Xingye Huang, Jinyu Wu, Xiaofang Xia, Jiangtao Cui, Hui Li, Liang Wang, and Feng Zhang. 2026. PartitionKV: Redesigning LSM-tree KV Stores on NVMs with Adaptive Partitioning for Reducing Write Stalls and Amplification. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 62 (February 2026), 26 pages. https://doi.org/10.1145/3786676

*Both authors contributed equally to this research.
†These authors are co-corresponding authors.

Authors' Contact Information: Xingye Huang, xingyehuang@stu.xidian.edu.cn; Jinyu Wu, jyw@stu.xidian.edu.cn, Xidian University, Xi'an, Shaanxi, China; Xiaofang Xia, xiaofangxia89@gmail.com; Jiangtao Cui, cuijt@xidian.edu.cn; Hui Li, hli@xidian.edu.cn, Xidian University, Xi'an, Shaanxi, China; Liang Wang, liangwang@nwpu.edu.cn, Northwestern Polytechnical University, Xi'an, Shaanxi, China; Feng Zhang, fengzhang@ruc.edu.cn, Renmin University of China, Beijing, China.



ACM 2836-6573/2026/2-ART62
https://doi.org/10.1145/3786676

1 Introduction

Nowadays, persistent key-value (KV) stores are becoming increasingly important for supporting various applications in modern data centers, such as social networks [11], e-commerce platforms [53] and search engines [13]. Many of these applications are write-intensive, and hence these KV stores usually employ a write-friendly log-structured merge tree (LSM-tree) as the basic index structure, such as LevelDB [24], RocksDB [36], HBase [6] and Cassandra [32]. The LSM-tree significantly enhances write performance primarily through its out-of-place update strategy, which converts random I/Os into sequential I/Os. Concretely, insertion, update, and deletion operations are implemented via sequential write operations—by appending the key followed by either its value or a special deletion.

Most LSM-tree based KV stores are deployed on a DRAM–SSD architecture, where written KVs are first buffered in DRAM and then flushed to SSD as SSTables. These SSTables are organized into multiple levels (L_0, L_1, ..., L_n). To mitigate the space consumption caused by outdated KVs, L_i-L_{i+1} compaction is triggered at appropriate time to merge a portion of SSTables from L_i (and L_{i+1}) by replacing outdated KVs with their newer versions. It is well known that the compaction policy adopted by the LSM-tree has a significant impact on read performance, write performance, space utilization, and write amplification, essentially representing a trade-off among these aspects. Two basic policies exist: leveling, which favors reads, and tiering, which favors writes. Hybrid policies combine both, typically applying tiering to upper levels and leveling to lower ones. While leveling (except L_0) is suitable for read-heavy and space-constrained scenarios, it suffers from degraded write performance under random writes due to: (1) **write amplification** caused by rewriting KVs during L_i-L_{i+1} compaction; (2) **write stalls** primarily resulting from slow compaction, especially from L_0-L_1 compaction, which is caused by the excessive number of SSTables involved in both levels. More details are provided in Section 2.3.

In order to alleviate the write amplification introduced by the leveling policy, many studies have proposed novel compaction strategies along with structural modifications to the lower levels that employ this policy, including Spooky [18], Wisckey [35], BlockDB [47], and ALDC [12]. However, these studies rarely focus on the write amplification caused by L_0–L_1 compaction; in other words, only the write amplification from L_1 to L_n are partially mitigated. In parallel, only a few studies have examined the impact of write stalls on the overall write performance of KV stores, especially those caused by L_0-L_1 compaction, such as TRIAD [7], SILK [8] and CruiseDB [33]. However, due to the significantly lower read and write performance of SSDs compared to DRAM, it is infeasible to eliminate key range overlaps among SSTables in L_0 without incurring significant transformation costs, resulting in a limited reduction of write stalls.

With the development of hardware technology, non-volatile memory (NVM) today offers storage capacities close to SSDs, access speeds comparable to DRAM, and is byte-addressable [21–23, 39]. It is also a persistent memory that can be accessed through the memory bus. These features bring new possibilities for solving the above issues in LSM-trees by integrating NVM into the DRAM-SSD architecture. Some existing studies have proposed new data management structures in NVM to replace L_0, along with customized compaction strategies, to mitigate the impact of L_0-L_1 compaction on the write performance of KV stores, such as NoveLSM [30], MatrixKV [55], TriangleKV [19] and FlatLSM [25]. However, these approaches suffer from one or more of the following limitations, which hinder their effectiveness in reducing write stalls and write amplification: (1) **Limitation 1**: high transformation overhead introduced by the proposed data management structures; (2) **Limitation 2**: during L_0–L_1 compaction, KVs with overlapping key ranges in NVM (i.e., the new L_0) cannot be compacted together, leading to repeated involvement of the same L_1 SSTables in multiple compactions; (3) **Limitation 3**: coarse-grained L_0–L_1 compaction slows down the

overall compaction process and delays the release of NVM space, which blocks new KV writes and exacerbates write stalls.

To address the above limitations, in this paper we present PartitionKV, a novel LSM-tree based KV storage specially designed for DRAM-NVM-SSD hybrid storage architecture, which has three main design characteristics: (1) **Character 1**: we propose to apply an **ordered partition layer** comprising multiple partitions with non-overlapping and consecutive key ranges to replace Memtable components and L_0 of original LSM-trees. Incoming KVs are directly persisted into NVM Logs of designated partitions and indexed by skiplists in DRAM. This design does not require additional transformation overhead, while enabling KVs within the same partition (i.e., the same key range) to be compacted together, thereby reducing the number of times SSTables in L_1 are involved in compactions. It helps address **Limitations 1 ~ 2**. (2) **Character 2:** we propose an **adaptive partitioning strategy** that dynamically splits or merges partitions based on the number of overlapping SSTables. Specifically, if a partition has a large number of overlapping SSTables, it is logically split into two smaller partitions; otherwise, it is logically merged into an adjacent partition with few overlapping SSTables. This strategy enables dynamic adjustment of each compaction granularity without rewriting KVs and helps address **Limitations 3**. (3) **Character 3**: we propose a **multi-threaded compaction strategy** to further accelerate space reclamation, whereby multiple threads concurrently execute separate compaction tasks between the partition layer and L_1. Specifically, each available compaction thread first traverses a high-priority list, followed by a low-priority list, to select appropriate partitions for compaction. This strategy also mitigates **Limitation 3**. Contributions of this work are highlighted as follows:

- We propose a new LSM-tree based KV store called PartitionKV, which improves the system's random write performance significantly.
- We design an ordered partition layer to replace Memtable and L_0, which enables compaction of KVs within the same key range with minimal overhead and fewer L_1 SSTables involved, thereby reducing writing amplification.
- We introduce an adaptive partitioning strategy that achieves fine-grained compaction by logically splitting or merging partitions, thereby reducing write stalls.
- We propose a multi-threaded compaction strategy that allows multiple threads to concurrently compact different partitions, further reducing write stalls.
- Extensive experiments are conducted to evaluate PartitionKV, and the results show that its random write performance is 3.63× and 4.06× of that of FlatLSM and MatrixKV, respectively, while its average write latency is 1.94× and 2.32× lower.

2 Background and Motivation

2.1 Log-structured Merge Trees

We take RocksDB, which is widely used in graph processing [10], stream processing [14] and OLTP workloads [35], to describe the specific design of the LSM-tree. As shown in Figure 1(a), RocksDB consists of a Mutable Memtable and an Immutable Memtable in DRAM as well as Sorted String Tables (SSTables), a WAL and a MANIFEST in SSD. The Mutable Memtable and Immutable Memtable are implemented based on skiplists. The Mutable Memtable is responsible for accepting inserted KVs, and the Immutable Memtable is responsible for flushing KVs from DRAM to L_0. SSTables organize KVs in the order of keys. As the volume of written data increases, SSTables are continuously compacted from L_0 to deeper levels (L_1, L_2, ..., L_n). Therefore, the LSM-tree presents a multi-level structure in SSD, where the capacity of L_{i+1} is generally 10 times that of L_i. The WAL records write requests, and the system can recover the Mutable Memtable and Immutable Memtable using the WAL after a system crash. The MANIFEST file is a metadata log that records all changes

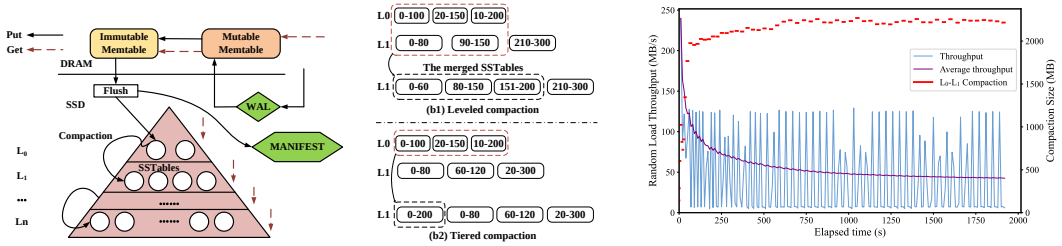


Fig. 1. (a) The structure of RocksDB. (b) RocksDB's leveled and tiered compactions. (c) RocksDB's random write performance and L_0–L_1 compactions. The light blue line shows the random write throughput measured every 10 seconds, the violet line shows the average throughput, and each light red line represents the duration and amount of data processed in a L_0–L_1 compaction.

to the database's versioned state, enabling RocksDB to be restored to the latest known consistent state on a restart.

Write KVs. When handling write operations, the KVs are first written to the WAL and then inserted into the Mutable Memtable. If the Mutable Memtable is full, it is converted into an Immutable Memtable, and a new Mutable Memtable is created to receive KVs. The Immutable Memtable is flushed to L_0 by the background compaction thread, and one or more SSTables are generated after the Flush is completed.

Read KVs. Since the LSM-tree applies an out-of-place update strategy, its upper levels store the most recently written KVs. When a read request arrives, RocksDB first searches the Mutable Memtable, then the Immutable Memtable, and finally the SSTables from L_0 to L_n in order. This process leads to a read amplification problem.

Background compaction. In RocksDB, the leveling and tiering policies correspond to leveled compaction and tiered compaction, respectively. As shown in Figure 1(b1), in leveled compaction, RocksDB merges the SSTables in both L_i and L_{i+1} whose key range are overlapping, and writes the newly generated SSTables into L_{i+1}. As a result, the KVs in each level (except for L_0) are unique. In contrast, as shown in Figure 1(b2), tiered compaction selects a group of adjacent SSTables with similar sizes within L_i for merging, and writes the resulting SSTables into L_{i+1}. Consequently, within each level, there exist KVs with the same key (i.e., both new and old versions). We can conclude that leveled compaction generally provides better read performance due to the uniqueness of keys in each level (except for L_0). However, since SSTables in L_{i+1} are involved in the compaction, it leads to significantly higher write amplification compared to tiered compaction. In RocksDB, to handle write bursts, no compaction operation is performed during the Flush process, which means that L_0 adopts the tiered compaction by default — a behavior that cannot be changed by user configuration. However, this results in the SSTables in L_0 being unordered and potentially covering a wide range of keys. When the user specifies that other levels adopt leveled compaction (the default setting in RocksDB), compaction between L_0 and L_1 can sometimes involve nearly all SSTables in both levels, leading to slow compaction and eventually causing write stalls.

2.2 Non-volatile Memory

New types of NVM are byte-addressable and memory-bus accessible, such as phase-change memory [9], ReRAM [50], memristors [45], 3D XPoint [27], and STT-MRAM [20]. These memories can be accessed through the memory interface using *mmap()/memcpy()*, or through the file system interface using *read()/write()* [40]. Compared to DRAM, NVM's bandwidth is 5× lower and its latency is 3× higher [27, 52]. However, NVM provides larger capacity than DRAM at a lower cost and with lower energy consumption [15, 52]. Compared with SSDs, NVM can be directly accessed

by the CPU via the memory bus, without relying on DRAM as an intermediary. Besides, it offers comparable capacity to SSDs, while providing 10× higher bandwidth and 100× lower latency [5]. Moreover, ensuring data consistency in NVM mainly involves addressing two challenges: write reordering and partial persistence. The former can be resolved by enforcing ordered memory writes using memory fence and cache flush instructions [26, 30], while the latter can be mitigated through mechanisms such as Write-Ahead Logging, Copy-on-Write, or Log-Structured Updates.

Research Significance: Although Intel officially discontinued Optane Persistent Memory in 2022, the rapidly growing demands of the AI industry have driven the emergence of new memory solutions. In 2024, Samsung introduced a series of Compute Express Link (CXL)-based devices, including the CXL Memory Module – DRAM (CMM-D) [42], CXL Memory Module – PMEM/Hybrid (CMM-H) [43], and CXL Memory Module – Memory Box (CMM-B) [41]. Among them, the CMM-H device integrates DRAM and NAND flash, providing byte-addressability and persistence capabilities similar in function to those of Optane persistent memory, as analyzed as follows: (1) Its internal DRAM provides byte-addressability and allows the host application to directly control or influence its cache management, thus enabling the prefetching and eviction of specific pages [44]. As evaluated in [44], under heavy loads, the system with CMM-H demonstrates lower latency at equivalent memory bandwidth levels compared to a DRAM-only system. (2) Moreover, it includes an internal energy source and supports CXL Global Persistent Flush (GPF) and Sudden Power Loss (SPL), ensuring the persistence of data in DRAM. These characteristics, which indicate that it can be used to target Intel Optane as well as NVDIMM customers [43], are critical for the design and implementation of PartitionKV. This implies that **the partition layer of the PartitionKV can be deployed on the DRAM within the CMM-H device**. Moreover, PartitionKV may require certain adjustments to better leverage its cache mechanism. For example, during write operations, when the cache space becomes insufficient, data pages from cold partitions can be selectively evicted to the SSD to free up space for incoming KVs. Before compacting cold partitions, the evicted pages can be prefetched from the SSD, thereby leveraging SSD's large capacity.

Additionally, in applications, if we do not have NVM available, the PartitionKV would still work with its persistent part in a NVMe SSD, but its overall performance would degrade greatly. However, the underlying partitioning and key-range management principles can provide useful insights for designing other systems aimed at enhancing random write performance.

2.3 Motivation

Write Stalls. Write stalls refer to delays or interruptions in the process of write operations, which can significantly cause frequent throughput degradation and long tail latency. To assess the impact of L_0-L_1 compaction on system throughput, we conduct an experiment on RocksDB, in which 80GB of KVs are randomly written into RocksDB, with the key and value sizes being set as 16B and 4096B, respectively. From the experimental result shown in Figure 1(c), we observe that the average throughput of RocksDB decreases over time, with periodic fluctuations, where the troughs in the throughput curve indicate write stalls. During L_0-L_1 compaction, RocksDB's throughput drops to the bottom of the valley. Once the compaction is completed, the throughput returns to its peak. The reason why write stalls caused by L_0-L_1 compaction have a significant impact on system throughput is that the disordered structure of L_0 causes nearly all SSTables in both L_0 and L_1 to participate in the compaction process. This results in excessive consumption of system resources - including memory, disk bandwidth, and CPU - slowing down compaction and causing continuous accumulation of L_0 files. The large number of overlapping L_0 files further degrades read performance and exacerbates subsequent L_0-L_1 compactions. Therefore, in RocksDB, the size of L_0 is typically set to be very small (usually a few hundred MB), and write stalls are triggered once the amount of data in L_0 exceeds a user-specified threshold. Moreover, although RocksDB introduces

sub-compaction to address the slow L_0-L_1 compaction issue, its efficiency remains limited because many SSTables in L_1 are repeatedly involved in compactions with newly flushed SSTables in L_0.

Write amplification. Write amplification is defined as the ratio between the amount of data written to storage devices and the amount of data written by users. During each L_i-L_{i+1} compaction, the KVs in the SSTables of both L_i and L_{i+1} are first read into DRAM, and then the KVs in L_{i+1} are replaced by those from L_i with identical keys. Subsequently, the merged data is written back to SSDs. Since some KVs actually do not need to be updated, this results in unnecessary data rewriting. Apparently, reducing the proportion of KVs that do not require update during the compaction process can reduce unnecessary data rewriting.

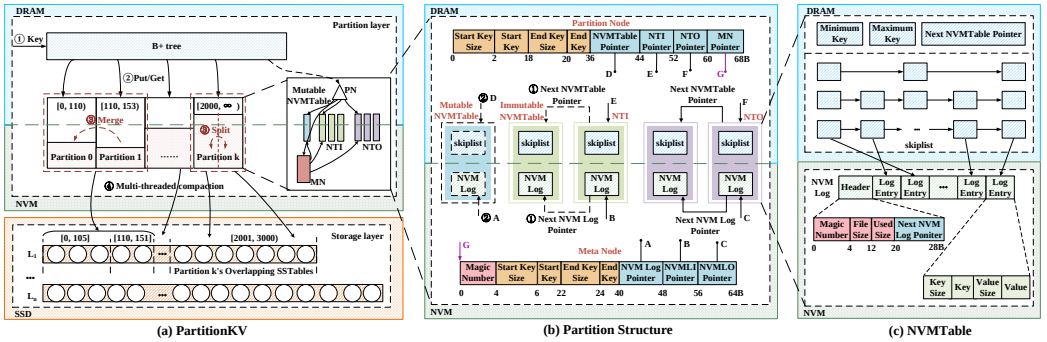
WAL overhead. As aforementioned, the WAL is used to ensure the recoverability of KVs stored in DRAM, which involves actual write operations to the SSD. Moreover, if the WAL recording written KVs is further used as a part of L_0, we do not need to further flush KVs in the Immutable Memtable into L_0. That is to say, the Flush operation can be eliminated, which helps reduce data rewriting.

3 PartitionKV Design

In this section, we present PartitionKV which aims to utilize the characteristics of NVM to reduce write amplification and write stalls caused by L_0-L_1 compaction, thereby improving the write performance. As shown in Figure 2(a), **PartitionKV** is based upon the DRAM-NVM-SSD three-tier storage architecture, and **can be primarily divided into a storage layer and a partition layer.** The storage layer located on SSDs maintains the same structure as the SSTables in L_1 to L_n of the original LSM-tree, with L_0 removed. The partition layer mainly consists of two components: (1) **multiple partitions spanning across both DRAM and NVM** which are used to replace the Memtable and L_0 for receiving written KVs, with the structure introduced in Subsection 3.1; (2) **a B+ tree located in DRAM** for indexing the partitions. When writing or reading a KV, PartitionKV first indexes the B+ tree to locate the partition whose key range covers the key of the KV. Then, it writes the KV to the partition or begins to search from this partition. Each partition's key range overlaps with the key ranges of a certain number of SSTables in L_1. These overlapping SSTables may participate in the compaction between the partition and L_1. To ensure an appropriate number of overlapping SSTables involved in each compaction, we propose an **adaptive partitioning strategy** to dynamically split or merge partitions according to the number of overlapping SSTables, as demonstrated in Subsection 3.2. When the number of KVs written to a partition reaches a predefined threshold, this partition is compacted into the L_1 to free up NVM space, in case that the NVM capacity are exhausted and subsequent KV writes are blocked. To accelerate data migration from the partition layer to L_1, we propose a **multi-threaded compaction strategy**, by which multiple threads are allocated to concurrently execute separate compaction tasks, as demonstrated in Subsection 3.3.

3.1 Partition Structure

Although some studies have attempted to use new data management structures on NVM to replace L_0 or Memtable in order to alleviate write stalls and write amplification, these approaches still require further improvement. For example, MatrixKV [55] introduces an ordered Matrix container to replace the disordered L_0 in original LSM-trees. However, it requires serializing the Immutable Memtable into a Matrix container and still uses the WAL only for Memtable recovery, resulting in expensive transformation overhead and write amplification caused by WAL. In contrast, NoveLSM [30] removes the WAL by introducing a persistent skiplist structure in NVM to replace the Memtable for handling KV writes; however, the L_0 is still disordered, and hence the L_0-L_1 compactions still involve a large amount of data. FlatLSM [25] directly appends KVs to NVM while maintaining



(a) PartitionKV (b) Partition Structure (c) NVMTable

Fig. 2. The overall structure of PartitionKV, which is a KV store based upon the DRAM-NVM-SSD three-tier storage architecture. The shaded area of the partition represents the number of SSTables in L_1 whose key ranges overlap with the partition's key range. The larger the shaded area, the more SSTables are covered.

skiplists in DRAM, further eliminating the need for a Memtable. It also proposes a multi-threaded compaction similar to sub-compaction [37] in RocksDB to accelerate large-granularity compaction. However, it cannot compact KVs within the same key range together, resulting in SSTables in L_1 participating in compactions multiple times.

To better address the issues of write stalls and write amplification, in this paper **we propose to replace the Memtable and L_0 with a partition layer consisting of multiple partitions and a B+ tree. Each partition, which is assigned with a specific key range, receives KVs within their key ranges. Note that key ranges of all partitions are unique, non-overlapping and consecutive.** This design significantly reduces write amplification through the following (1) KVs within the same key range can be compacted together. This implies that the proportion of KVs with identical keys (i.e., necessary data rewriting) during each compaction increases, and accordingly unnecessary data rewriting is reduced; and (2) each partition contains components called NVM Logs that function as WALs while maintaining skiplists in DRAM to index the KVs stored in the NVM Logs, which eliminates the Flush operation.

As depicted in Figure 2(b), a partition mainly consists of a Partition Node (PN), a Mutable Non-volatile Memory Table (NVMTable), an Immutable NVMTable list of itself (NTI), an Immutable NVMTable list of other partitions (NTO), and a Meta Node (MN). *The PN records the metadata of a partition, such as its key range and pointers to other components within the partition. The Mutable NVMTable is used to receive incoming written KVs. Once it becomes full, it is converted into an Immutable NVMTable and appended to the tail of the NTI. When a partition is split or merged, the NTI is converted into the NTO. The MN records critical partition state information for system recovery.* Their specific structures and functionalities are elaborated as follows:

3.1.1 Partition Node. As shown in Figure 2(b), the PN consists of: (1) a 16-byte Start/End Key which together defines the key range of this partition, along with a 2-byte Start/End Key Size which records the size of the Start/End Key; (2) four pointers, including an 8-byte NVMTable Pointer, an 8-byte NTI Pointer, an 8-byte NTO Pointer and an 8-byte MN Pointer, which point to the Mutable NVMTable, the NTI, the NTO and the MN within the partition, respectively.

When write or read requests arrive, PartitionKV first locates the partition whose key range contains the requesting key through an B+ tree in DRAM. It then use the information within the PN to locate the Mutable NVMTable in this partition to write KVs or start querying from it, followed by NTI and NTO.

3.1.2 NVMTable. As indicated in paper [25], maintaining skiplist structures on NVM involves a large number of small-granularity writes, which results in poor write performance. Thus, FlatLSM

[25] separates the KVs from the index nodes by writing the KVs directly to the NVM and maintaing skiplists in DRAM that point to the written KVs, which helps reduce small-granularity writes on NVM. Following this philosophy, we design NVMTable which contains a Minimum Key, a Maximum Key, a Next NVMTable Pointer, a skiplist which resides in the DRAM, as well as a NVM Log that resides in the NVM, as shown in Figure 2(c). The Minimum Key and the Maximum Key together indicates the key range of the written KVs in NVMTable; the Next NVMTable Pointer links adjacent Immutable NVMTables to form an Immutable NVMTable list; the skiplist points to the KVs in the NVM Log.

The detailed structure of the NVM Log are depicted in the lower part of Figure 2(c). The NVM Log contains a 28-byte Header and multiple Log Entries, which are demonstrated as follows: (1) The Header records the NVM Log's metadata, which contains: a) a 4-byte Magic Number. If it is not filled with the hex value of "PMEM", the system will identify the NVM Log as being invalid and automatically reclaim the space; b) an 8-byte File Size and an 8-byte Used Size, which record the size of NVM Log and the amount of space already used, respectively; c) an 8-byte Next NVM Log Pointer, which is used to persist the linkage of NVM Logs across Immutable NVMTables in NVM. This ensures that even if a system crash occurs and the linkage of Immutable NVMTables in NTI and NTO—maintained by the Next NVMTable Pointer—is lost, the NTI/NTO can still be reconstructed. (2) Each log entry, used to persist written KVs, consists of four fields: Key Size, Key, Value Size, and Value.

When a new KV is written to the corresponding partition, it is first appended to the NVM Log of Mutable NVMTable as a Log Entry, and then the field of Used Size in the Header is updated accordingly. After the KV has been successfully persisted to NVM, corresponding index nodes are inserted into the skiplist, and the Minimum and Maximum Key fields are updated accordingly.

Notably, the difference of the PMTable in paper [25] and the proposed NVMTable lies in that we add the Next NVMTable Pointer in the DRAM and the Next NVM Log Pointer of the NVM Log in the NVM, both of which are used to maintain the linkage of the Immutable NVMTables. *These structures are essential for the dynamic management of partition splitting and merging.*

3.1.3 Meta Node. To ensure partition recovery during system crashes where DRAM-stored PN information may be lost, we design a Meta Node (MN) located in NVM as a persistent storage mechanism. *The MN stores essential partition state information and must be promptly synchronized whenever structural changes occur, such as key range modifications or the creation of a new Mutable NVMTable. This design establishes a robust recovery mechanism by maintaining NVM-based state tracking that persists beyond volatile memory failures.*

As shown in the lower part of Figure 2(b), the MN consists of the following critical components: (1) a 4-byte Magic Number, which serves as the validity indicator. Specifically, if it is not filled with the hex value of "META", then the system will identify the MN as being invalid and automatically reclaims the space; (2) a 2-byte Start/End Key Size which records the size of Start/End Key, as well as a 16-byte Start/End Key which together indicates the key range of the partition; (3) an 8-byte NVM Log Pointer, which directs to the NVM Log of the Mutable NVMTable of the partition; (4) an 8-byte Pointer of the Immutable NVM Log list of itself and other partitions, called the NVMLI and NVMLO Pointers, which direct to the NVM Log of the Immutable NVMTable at the head of the NTI and NTO, respectively.

During system recovery, the pointer triad (NVM Log Pointer, the NVMLI and NVMLO Pointers) enables reconstruction of the active Mutable NVMTable state as well as the structures of NTI and NTO.

3.1.4 Immutable NVMTable list of itself & other partitions. When the used storage space of the Mutable NVMTable reaches a predefined threshold, it is first converted into an Immutable NVMTable,

which is then compacted to L_1 by an available compaction thread. However, since the compaction between partitions and L_1 involves a large amount of I/O operations, it proceeds relatively slow, while incoming KVs are written into newly allocated Mutable NVMTables at a relatively high speed, which are then transformed into Immutable NVMTables. This leads to more and more Immutable NVMTables buffering and awaiting for compaction, which forms the NTI. That is to say, *the **NTI** essentially consists of multiple Immutable NVMTables from its partition, storing KVs within the partition's key range.*

When a Mutable NVMTable is transformed into an Immutable NVMtable, two scenarios arise: **Case 1**: If the NTI is empty, the newly created Immutable NVMtable becomes the head of the NTI. Consequently, the NTI Pointer in the PN is updated to reference it, while the NVMLI Pointer in the MN is directed to its corresponding NVM Log. **Case 2**: If the NTI is not empty, the Immutable NVMtable is appended to the tail of the NTI. This is achieved by updating the Next NVMTable Pointer and Next NVM Log Pointer of the current tail entry. Additionally, whenever a new Immutable NVMtable is formed, it is necessary to determine whether the partition should be split or merged, which will be discussed in detail in Subsection 3.2. If no splitting or merging is required, a new Mutable NVMTable is allocated. The NVMTable Pointer in the PN is then updated to reference this newly allocated Mutable NVMTable, while the NVM Log Pointer in the MN is set to its corresponding NVM Log.

*The **NTO** is transformed from the NTI of other partitions that have been split or merged, and the key range of the stored KVs may exceed the key range of the partition*, which will be elaborated later in Subsection 3.2. ***Compared to NTI, these Immutable NVMTables in NTO are prioritized for compaction into L_1 together by compaction threads***, which will be elaborated in Subsection 3.3.

3.1.5 Recovery of Partitions. In the partition, the components of the PN, the skiplist of NVMTable and the Next NVMTable Pointer of NVMTable reside in DRAM. *When system crash occurs, information in these components will get lost, which can be restored using the MN and NVM Log located in the NVM.* The partition recovery proceeds as follows: (1) based on the NVM Log Pointer of the MN, KVs in the NVM Log of the NVMTable are scanned to reconstruct the corresponding skiplist, Minimum Key and Maximum Key in DRAM. This process effectively restores the Mutable NVMTable; (2) based on the NVMLI Pointer of the MN and the Next NVM Log Pointer of the NVM Log, a NVM Log list is scanned to rebuild the NTI; (3) the recovery process of NTO is the same as that of NTI; (4) PartitionKV reconstructs the PN based on the partition key range of the MN in the NVM and the recovered Mutable NVMTable, NTO and NTI.

3.2 Adaptive Partitioning Strategy

Apparently, after replacing L_0 with an ordered partition layer, each partition has a certain number of overlapping SSTables in L_1. As shown in Figure 2(a), Partition 1 and Partition k has 2 and 7 overlapping SSTables, respectively. These overlapping SSTables may participate in compaction between the partition and L_1. *When a partition is frequently written with KVs and undergoes multiple rounds of compaction, the number of overlapping SSTables in L_1 increases.* As a result, the compaction granularity grows in subsequent rounds, which may lead to severe write stalls. To avoid this issue, the partition needs to be split into smaller key ranges. However, due to the limited capacity of NVM, the total number of partitions is constrained. To ensure that there is sufficient available space for creating a new partition when an existing one needs to be split, some existing partitions needs to be merged. Fortunately, for partitions that are not frequently written with KVs, the number of overlapping SSTables in L_1 gradually decreases as they are continuously compacted into L_2. Merging such partitions with their adjacent ones does not significantly increase the compaction granularity.

To achieve this goal, **we propose an adaptive partitioning strategy that dynamically splits or merges partitions based on the number of overlapping SSTables, ensuring that an appropriate amount of data is involved in compaction.** Specifically, if a partition has a large number of overlapping SSTables, it is split into two smaller partitions; conversely, if it has a few overlapping SSTables, it is merged with its adjacent partition that has fewer overlapping SSTables.

The overall process of the adaptive partitioning strategy is stated as follows: *For any given partition of interest, whenever a Mutable NVMTable is converted into an Immutable NVMTable that is then immediately appended to the tail of the NTI, the next steps depend on the state of the NTO.* Specifically, if the NTO is not empty, a new Mutable NVMTable is allocated to receive incoming KVs; otherwise, a decision must be made on whether to split or merge the partition. If splitting or merging is required, the corresponding operations are performed accordingly; otherwise, a new Mutable NVMTable is simply created to accommodate the incoming KVs. In the following, we demonstrate how to make the decision and how the partition splitting and merging proceeds.

3.2.1 Partition Split/Merge Decision. When PartitionKV is initially launched, the partition layer contains only a single partition. As KVs continues to be written, the number of partitions, denoted by n_p, gradually increases, until it reaches a user-specified maximum number denoted by δ_{max}. *Based upon the value of n_p, we divide the running period of PartitionKV into three stages.* Specifically, if $1 \leq n_p \leq \delta_{min}$, PartitionKV is in **Stage 1**; if $\delta_{min} < n_p \leq \delta_{mid}$, it is in **Stage 2**; and if $\delta_{mid} < n_p \leq \delta_{max}$, it is in **Stage 3**, with δ_{min} and δ_{mid} being two user-specified parameters.

Let n_s denote the number of overlapping SSTables in L_1 of an interested partition. Let β_i^s and β_i^m denote the user-specified thresholds at Stage i for determining whether the interested partition should be split or merge, respectively. We apply different rules at each stage, as demonstrated as below:

Rule 1: *This rule applies in **Stage 1** when the PartitionKV is launched shortly.* In this case, we usually set β_1^s as a small number such that the partitions splits quickly. Specifically, if we have $n_s \geq \beta_1^s$, the partition of interest is split. This reduces the number of overlapping SSTables within each partition, thereby shortening compaction time and enhancing the system's write performance.

Rule 2: *This rule applies in **Stage 2** when there is still sufficient NVM storage space to create new partitions.* We use fixed thresholds to determine whether a partition should be split or merged. Specifically, if we have $n_s \geq \beta_2^s$, this partition should be split; otherwise, if we have $n_s \leq \beta_2^m$ and $n_p - 1 > \delta_{min}$, this partition should be merged, with $\beta_2^s > \beta_2^m$. The constraint $n_p - 1 > \delta_{min}$ ensures that the PartitionKV still runs in Stage 2 after a partition merging process completes.

Rule 3: *This rule applies in **Stage 3** when the available NVM space is nearly exhausted, making it difficult to create new partitions, while the number of SSTables in L_1 continues to grow.* It is necessary to further raise the thresholds for partition splitting and merging to achieve a relatively balanced increase in the number of overlapping SSTables per partition. Specifically, we determine whether the partition should be split or merged based on the coverage ratio, denoted by r_c, which is defined as the ratio of the number of overlapping SSTables of this partition to the total number of SSTables in L_1. If we have $r_c \geq \beta_3^s$ and $n_p + 1 \leq \delta_{max}$, this partition will be split; otherwise, if $r_c \leq \beta_3^m$, this partition will be merged.

3.2.2 Partition Splitting. The partition splitting proceeds as follows: *For a given partition of interest, PartitionKV first identifies the overlapping SSTables in L_1 and selects the largest key in the middle overlapping SSTable as the split key. A new partition is then created, with the key ranges of both partitions being updated based on the split key, followed by an immediate adjustment of the B+ tree index.* For example, assume that Partition 1 in Figure 3 whose key range is [0-100] needs to be split. Since its overlapping SSTables include SST1, SST2, SST3, SST4 and SST5, we set the split key as the largest key of SST3, i.e., 60. Then, a new partition, i.e., Partition 1* in Figure 3, is created with the

key range (60-100], and the key range of Partition 1 is adjusted as [0-60]. Immediately, the index key of Partition 1 is updated as 60, and a new index key 100 pointing to the PN of Partition 1* is created at the same time.

Afterwards, PartitionKV converts the NTI of the original partition into an NTO by modifying the NTO pointers in the PNs of both partitions. As shown in Figure 3, the NTI of the original Partition 1 is transformed into the NTO of the split Partition 1 and Partition 1*, pointed by their NTO Pointers; and the NTI Pointer of Partition 1 is set as null. Finally, PartitonKV updates the key ranges, NVMLI pointers, and NVMLO pointers in the MNs of both partitions to persist the partition state based on their PNs.

Through the above steps, it can be seen that *the NTI becomes a shared NTO for both partitions*, causing the KVs stored in the NTO to exceed the key range of each of these two partitions. Subsequent KVs will be written to the corresponding partition based on the key ranges of the two new partitions.

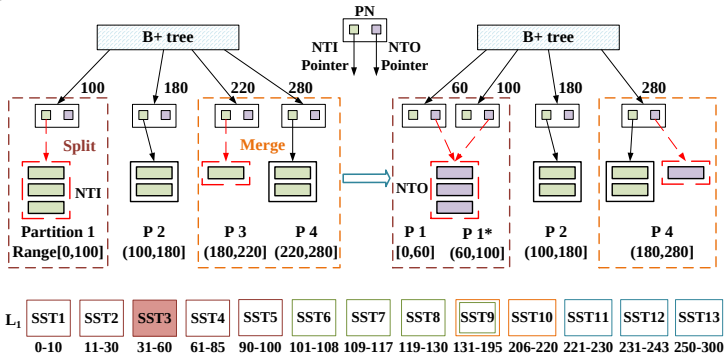


Fig. 3. Illustration examples of partition splitting/merging. The red dashed arrows represent that the corresponding NTI/NTO pointers are (to be) modified.

3.2.3 Partition Merging. The partition merging proceeds as follows: *For a given partition of interest, PartitionKV first identifies an acceptable partition, which is defined as the adjacent partition with an empty NTO and having fewer overlappling SSTables than the other adjacent one. Then, it expands the key range of the acceptable partition to the union of the key ranges of both partitions and updates the B+ tree index.* For example, assume that Partition 3 in Figure 3 needs to be merged. As shown, its left neighboring Partition 2 has an empty NTO and covers 4 overlapping SSTables, and the right neighboring Partition 4 has an empty NTO and covers 3 SSTables. Thus, the partition 4 is identified as the acceptable partition. Then, the key range of Partition 4 is expanded from (220-280] to (180-280], and the index key of Partition 3 is deleted.

Afterwards, PartitionKV converts the NTI of the original partition into NTO by modifying the NTO pointer in the PN of the acceptable partition. As shown in Figure 3, the NTI of the original Partition 3 becomes the NTO of the merged Partition 4, pointed by its NTO Pointer. Finally, PartitionKV deletes the partition to be merged and updates the MN of the acceptable partition to persist the partition state based on its PN.

Note that the partition splitting and merging operations do not require rewriting the KVs within the partitions; instead, they only involve modifying partition pointers. As a result, the time overhead for these two processes is very small and can almost be neglected.

3.3 Multi-threaded Compaction Strategy

The exhaustion of NVM or L_0 capacity in original LSM-tree blocks subsequent KV writes, causing severe write stalls. Many existing works have tried to alleviate this issue. For example, RocksDB

[36] introduces a sub-compaction strategy to accelerate L_0-L_1 compaction by dividing the involved KVs into multiple ranges, with each thread responsible for compacting the KVs within a specific range. However, when RocksDB selects SSTables in L_0 for compaction, many SSTables in L_1 are involved, which may also participate in subsequent compactions with other SSTables in L_0, leading to low efficiency. MatrixKV [55] applies a single-threaded fine-grained column compaction that compacts a column of KVs (instead of a large number of KVs) at each time to prevent delayed release of NVM space. However, MatrixKV's column compaction is single-threaded, resulting in slow data migration from NVM to SSDs.

To better alleviate write stalls, **we propose a multi-threaded compaction strategy, by which multiple threads are allocated to concurrently perform different compaction between the partition layer and L_1.** Specifically, each available thread selects an NTI or NTO for compaction. As previously discussed, the adaptive partitioning strategy ensures that each compaction of NTI or NTO involves an appropriate amount of overlapping SSTables. Since KVs with the same keys are written to the same partition, we can infer that a partition's overlapping SSTables typically do not participate in compaction with other partitions' NTI or NTO. By effectively synergizing the multi-threaded compaction strategy with adaptive partitioning, PartitionKV achieves fine-grained compaction comparable to MatrixKV while substantially outperforming RocksDB.

Note that if a partition is frequently written with KVs, the number of Immutable NVMTables in the NTI gets large, leading to slow compaction and write stalls. To mitigate this issue, when the number of Immutable NVMTables in the NTI reaches a user-specified threshold θ, PartitionKV slows down KVs ingestion and prioritizes this partition's NTI for compaction. To prioritize the compaction of the Immutable NVMTable list that was formed first and contains a larger amount of data, *we design two priority lists—a **high-priority list** and a **low-priority list**—to manage the compaction priority of NTI and NTO.* These priority lists store pairs of <NTI or NTO address, PN address>, where the corresponding NTI or NTO and PN belong to the same partition.

We introduce how to *set the compaction priority of NTIs and NTOs by placing them into appropriate lists*, as demonstrated as follows: (1) Once a partition completes splitting or merging, the original partition's NTI is converted to an NTO. PartitionKV traverses the high-priority list and then the low-priority list to find the pair storing this NTI address. Then, this pair is removed, and a new pair consisting of this NTO address and the corresponding PN address is created and appended to the high-priority list. This is because, in our design, a partition cannot be split or merged if its NTO is not empty, so all NTOs should be placed in the high-priority list in order to be compacted into L_1 as soon as possible. (2) After an Immutable NVMTable is added to the NTI, if the NTI contains only one Immutable NVMTable, PartitionKV creates a new pair consisting of this NTI address and the corresponding PN address and appends it to the low-priority list. Otherwise, PartitionKV finds the pair storing the NTI address and checks whether it is placed in the high-priority list. If not, PartitionKV removes it from the low-priority list and appends it to the high-priority list.

We also allocate multiple threads to handle the compaction between the partition layer and L_1. Each available thread traverses the high-priority list from head to tail, followed by the low priority list, to select the NTI/NTO for compaction, as follows:

(1) If the pair stores the address of an NTO, PartitionKV checks whether there is an SSTable in L_1 whose key range overlaps with the key range of the KVs stored in the NTO and is currently involved in the compaction of another NTI or NTO. If so, the pair is skipped, and the next pair is checked for qualification. Otherwise, the NTO recorded in the pair will be compacted, and this pair is removed once the compaction is complete. This is because, if multiple threads simultaneously select the same SSTable for compaction, the key ranges of the newly generated SSTables will overlap, affecting subsequent compaction and read operations.

(2) If the pair stores the address of an NTI, an additional condition is required: PartitionKV needs to find the corresponding PN based on this pair to check whether the NTO of the same partition is empty. If it is empty, the NTI is qualified for compaction; otherwise, this pair is skipped. This is because, if NTO comes from a split partition, the KVs stored in NTO are older than the KVs stored in NTI within the same partition. If the compaction of NTO is not prioritized for compaction, old data will overwrite new data. In addition, in our design, prioritizing the compaction of NTO enables partitions to be split and merged as quickly as possible.

Remark: Novelties of PartitionKV *are summarized as follows:* (1) The **partition structure** ensures that the partition layer remains ordered and that KVs with the same keys are written to the same partition, thereby reducing unnecessary KV rewriting during compaction and decreasing the frequency of compaction of overlapping SSTables in L_1. In addition, **each NVM Log serves as both the WAL and part of** L_0, eliminating the original Flush operation. These aspects together significantly reduce write amplification. (2) **Adaptive partitioning strategy** enables fine-grained compaction between the partition layer and L_1, which reduces the frequency of slow compaction and mitigates the occurrence of write stalls. (3) **Multi-threaded compaction strategy** enables concurrent compaction between the partition layer and L_1, further accelerating NVM space release and reducing write stalls.

4 Implementation

The Persistent Memory Development Kit (PMDK) [2, 51] is a collection of libraries and tools for System Administrators and Application Developers to simplify managing and accessing persistent memory devices. With the functions provided by PMDK, we use two Intel Optane Persist Memory Modules to implement PartitionKV based on RocksDB [36]. Next, we briefly introduce the write/read process, the concurrency control, NVM space management and consistency mechanisms as follows.

Write: (1) PartitionKV first indexes the B+ tree in DRAM based on the key of KV to locate the corresponding partition. Then, it writes the KV into the Mutable NVMTable within the partition. (2) When the Mutable NVMTable is full, it will be converted into an Immutable NVMTable and appended to the tail of NTI. PartitionKV will adjust the priority of the NTI and then check if the partition needs to be split or merged. If this partition is required to be split or merged, PartitionKV will also adjust the priority of the NTO after completing the corresponding operation. (3) The available background compaction threads select NTIs or NTOs qualified for compaction simultaneously.

Read: The search priority, from high to low, is Mutable NVMTable, NTI, NTO, and SSTables from L_1 to L_n. Firstly, the system needs to find the partition to which the key of the read request belongs through the B+ tree, and then search from the Mutable NVMTable within the partition. If the KV is not found in the Mutable NVMTable, the ParitionKV will search from the tail to the head of NTI, then search from the tail to the head of NTO, and finally search from L_1 to L_n.

Concurrency control: Read and write concurrency in the partition layer is implemented using a lock. For read operations, the lock is first acquired to access the B+ tree and locate the corresponding partition. Pointers to the NVMTable, NTI, and NTO within the Partition Node are then recorded, along with the current database state (i.e., the Version object in RocksDB, which maintains metadata about all SSTables). The lock is then released, and the lookup proceeds using the recorded pointers and state information. For write operations, partition splitting or merging constitutes a step in the write process. The lock is acquired to locate the partition and to perform the splitting or merging if necessary, and is then released.

NVM space management: (1) PartitionKV creates a file on NVM according to the number and size of PM Logs and Meta Nodes, as shown in Figure 4. The file is divided into two regions—the Meta Node (MN) storage region and the NVM Log storage region—where the corresponding MNs and PM Logs are initialized. It is then mapped into the virtual address space, with pointers stored as

offsets relative to the virtual base address obtained after file mapping. This approach is not affected by the Address Space Layout Randomization (ASLR), which is used to enhance system security by randomizing the base addresses of stack, heap, and executable memory regions. (2) PartitionKV maintains two queues to manage MNs and NVM Logs, respectively. When a new partition is created or a partition requests more NVM space, an MN or PM Log is allocated from the corresponding queue. Similarly, when a partition is deleted or its NVM space is released, the corresponding MN and PM Log are returned to their respective queues. We adopt the Reference Counting mechanism [36] in RocksDB to ensure that PM Logs and Meta Nodes are not released while being accessed. Therefore, there is no NVM leak issue.

Consistency: NVM data structure must avoid inconsistencies caused by system failures [16, 17, 31, 34, 38]. In PartitionKV, NVM write/update operations are triggered by two types of events: KV writes and partition state updates. For KV writes, if the size of a KV exceeds the write granularity of NVM, a system crash may lead to partial persistence. To solve this issue, PartitionKV atomically updates the 8-byte Used Size in the NVM Log once the write operation is complete. If a system crash occurs before modifying the Used Size, the KV will be lost. For partition state updates, since multiple fields in MN need to be updated after the operation is completed, a system crash may prevent some of them from being successfully modified. PartitionKV records its status in the MANIFEST file before modifying the corresponding MN.

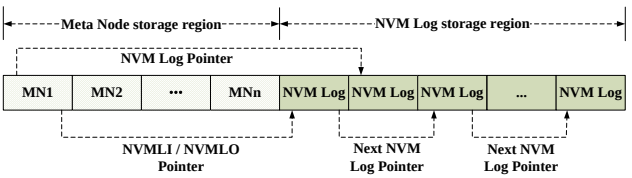


Fig. 4. Layout of NVM Storage Space.

5 Evaluation

5.1 Experiment Setup

All experiments are conducted on a Linux server with two Intel(R) Xeon(R) Gold 6240 2.60GHz 18-core processors and 379GB of memory. The kernel version is Linux 5.4.0-42-generic and the operating system is Ubuntu 20.04.6 LTS. The server uses a 1.84TB INTEL SSDPE2KX020T8 SSD with the EXT4 file system, and two 128GB Intel Optane Persist Memory Modules with the EXT4-DAX file system.

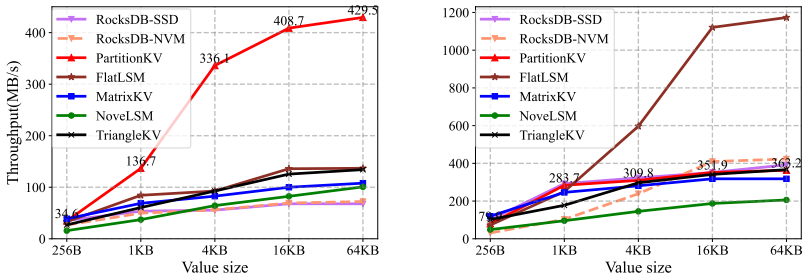
We compare the performance of PartitionKV with RocksDB [36], NoveLSM [30], MatrixKV [55], TriangleKV [19], and FlatLSM [25]. The experimental settings are briefly introduced as follows: (1) The version of RocksDB is 9.1.0. Its Memtable size is 64 MB, and sub-compaction is enabled with 5 threads. We use the terms RocksDB-NVM and RocksDB-SSD to refer to the system when its WAL is stored on NVM and SSD, respectively. (2) NoveLSM uses an 8 GB NVM Memtable and a 64 MB DRAM Memtable, which work alternately. (3) As for MatrixKV, the NVM size is 8 GB and the Memtable size is 64MB. (4) TriangleKV uses the same configuration as MatrixKV, with a single thread selecting the KVs for column compaction (i.e., right-angle side compaction). (5) In FlatLSM, the NVM space is 8GB, containing four 2GB PM Logs. The Flush operation is triggered when the number of Immutable PMTables reaches 2, and the number of compaction threads for handling Immutable PMTable compaction is set to 5. Note that we implement FlatLSM without using the key-value separation strategy. This is because after using this strategy, the data actually written to the LSM-tree in FlatLSM consists of the key and the addresses of value, resulting in a very small amount of data stored in the LSM-tree. Compared with other databases, when

writing the same amount of data, the amount of data stored in the LSM-tree in FlatLSM is too small and write stalls caused by L_0-L_1 compaction rarely occur. Therefore, it is unfair to directly compare FlatLSM with other databases to evaluate their performance in handling write stalls. (6) For PartitionKV, the NVM size is 8 GB, and the NVM Log size is 64 MB. Therefore, the number of NVM Logs is $8GB/64MB = 128$. The number of compaction threads between the partition layer and L_1 is set as 5, and threshold parameters are set as follows: $\delta_{min} = 20$, $\delta_{mid} = 40$, $\delta_{max} = 60$, $\beta_1^s = 2$, $\beta_2^s = 10$, $\beta_2^m = 4$, $\beta_3^s = 0.033$, $\beta_3^m = 0.008$, and $\theta = 3$. Each partition occupies at least one PM Log and at most $(1 + 2\theta)$ PM Logs without triggering a reduction in write speed. Assuming that the total number of PM Logs is m, we recommend that the following relation be satisfied: $\delta_{\max} \leq m;\ (1 + 2\theta)\,\delta_{\min} \geq m;\ \delta_{mid} = (\delta_{\max} + \delta_{\min})/2$.

The other configurations of these systems are set to default and can be found on the corresponding websites [36], [29], [54], [3], [4], and [1]. Additionally, in the experiments, when evaluating their write performances, we randomly write approximately 80GB of KVs to them, where the key size is 16B. If not otherwise specified, the value size is set to 4KB, and the accessed keys are generated using Uniform distributions.

5.2 Read and Write Performance Evaluation

5.2.1 Micro-Benchmark. In this section, we use db_bench to evaluate the random write, sequential write, random read, and sequential read performance of the above databases when the value sizes range from 256B to 64KB.



(a) Random Write (b) Sequential Write

Fig. 5. Write performance with different value sizes.

Write performance. The random write performance of these databases is shown in Figure 5(a). PartitionKV outperforms FlatLSM, followed by TriangleKV, MatrixKV, NoveLSM, and RocksDB. The reasons for PartitionKV achieving higher throughput are stated as follows: (1) FlatLSM introduces a multi-threaded compaction strategy similar to RocksDB's sub-compaction to accelerate the compaction of Immutable PMTables. However, it faces the same issue that SSTables in L_1 may participate in the compaction of other Immutable PMTables, leading to low compaction efficiency. In PartitionKV, since the same KVs are written to the same partition, the SSTables in L_1 usually do not participate in the compaction of other partitions, therefore PartitionKV achieves higher compaction efficiency. (2) MatrixKV does not reuse the WAL for storing KVs, resulting in additional transformation overhead compared to PartitionKV. Additionally, it uses single-threaded column compaction to migrate KVs from NVM to SSDs, leading to lower random write performance. (3) TriangleKV builds upon MatrixKV by including newly written files in NVM as candidates for compaction. This design enables newly written KVs to be compacted jointly with existing KVs within the same key range, thereby reducing the number of L_1 files involved in compaction. However, it also increases the overhead of selecting KVs for compaction. (4) NoveLSM maintains a

persistent skiplist on NVM, which significantly reduces efficiency, resulting in poor random write performance. (5) Due to the low efficiency of RocksDB's sub-compaction, the write stall issue caused by L_0-L_1 compaction is not effectively mitigated, resulting in low random write performance.

The sequential write performance of these databases is shown in Figure 5(b). PartitionKV's sequential write performance is comparable to RocksDB, MatrixKV and TriangleKV, lower than FlatLSM, and higher than NoveLSM. The reasons are as follows: (1) In sequential writes, since the keys of the written KVs are increasing, the SSTables in the LSM-tree have non-overlapping key ranges. As a result, L_i-L_{i+1} compaction does not merge SSTables, but simply moves SSTables in L_i to L_{i+1} by modifying their metadata, without rewriting the KVs. Compared to RocksDB, PartitionKV directly writes KVs to NVM, which corresponds to RocksDB's WAL writes and the Flush operation of Immutable MemTable. In this process PartitionKV is faster than RocksDB. However, during the subsequent L_0-L_1 compaction, RocksDB only modifies the metadata of SSTables, while in PartitionKV, data in NVM must be migrated to L_1 in SSDs. In this process RocksDB outperforms PartitionKV. These two processes ultimately result in similar sequential write performance between PartitionKV and RocksDB. (2) In the partition layer, PartitionKV actually has only one partition receiving KVs, so only a single thread handles compaction. Meanwhile, when the lengths of NTI and NTO reach the threshold θ, PartitionKV reduces the ingestion rate of KVs. Both of these factors lead to degraded write performance. In contrast, FlatLSM compacts a 2GB Immutable PMTable to L_1 using multiple threads, making it more efficient. (3) Since MatrixKV has additional transformation overhead compared to PartitionKV, its sequential write performance is slightly lower than PartitionKV. (4) In this case, the performance of TriangleKV is close to that of MatrixKV. (5) The throughput of NoveLSM is always lower compared to other databases. This is because maintaining a skiplist on NVM results in poor write performance.

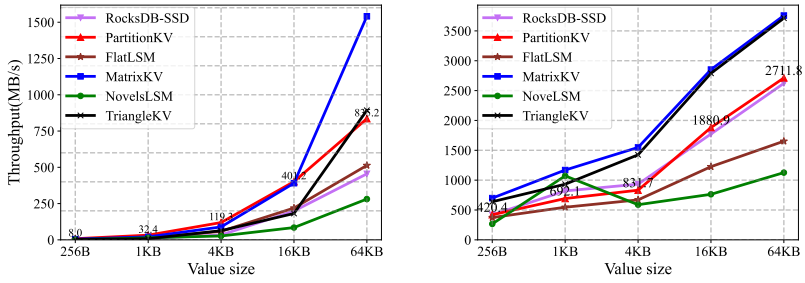
Table 1. Single threaded write performance

Database	Rand. (MB/s)	Seq. (MB/s)
PartitionKV	101.1	321.3
FlatLSM	55.4	584.8
MatrixKV	82.7	281.0
TriangleKV	92.7	296.8

As mentioned above, compared with sequential writes, random writes frequently trigger SSTable merges, leading to performance degradation. The performance gap between the two scenarios captures the impact of compaction, which actually reflects the differences in L_0-L_1 compaction efficiency among these systems, since the structures from L_1 to L_n are identical across these databases. To further examine this, we evaluated the single-threaded compaction performance for a more direct comparison. As shown in Table 1, PartitionKV achieves higher sequential and random write performance than MatrixKV and TriangleKV. Their performance gap between sequential and random writes is similar and much smaller than that of FlatLSM, indicating comparable compaction efficiency among them and superior to the FlatLSM. Moreover, even though MatrixKV and TriangleKV may be modified to use multiple threads for compaction, their efficiency will not exceed that of PartitionKV, because selecting KVs for compaction incurs significant overhead, whereas PartitionKV simply traverses two priority queues to select NTI/NTO, achieving higher efficiency.

Analysis of RocksDB: As shown in Figure 5(a), RocksDB-SSD and RocksDB-NVM have almost the same overall random write performance. This is mainly because their performance in this regard is dominated by write stalls resulting from compaction. As shown in Figure 5(b), RocksDB-SSD

performs better than RocksDB-NVM if value sizes is below 16KB, but worse otherwise. This is likely due to the hardware characteristics of NVM and its file system [46]. For clarity, we hereafter use RocksDB-SSD as the baseline.



(a) Random Read (b) Sequential Read

Fig. 6. Read performance with different value sizes.

Table 2. Statistics of lookup hits in NVM

Database	Value size	Total time (s)	Num.	Avg. (μs)	Total NTI Len.
MatrixKV	256B	2.094	69578	30.10	N/A
	1KB	2.132	74977	28.44	N/A
	4KB	1.754	77164	22.73	N/A
	16KB	1.666	76215	21.86	N/A
	64KB	2.209	74151	29.79	N/A
PartitionKV	256B	0.126	17990	7.00	0*53+1*1=1
	1KB	0.142	22162	6.41	0*48+1*3=3
	4KB	0.421	43297	9.72	0*10+1*27+2*19=65
	16KB	0.409	34809	11.75	0*6+1*30+2*19=68
	64KB	0.747	30516	24.48	0*11+1*26+2*20+3*1=69

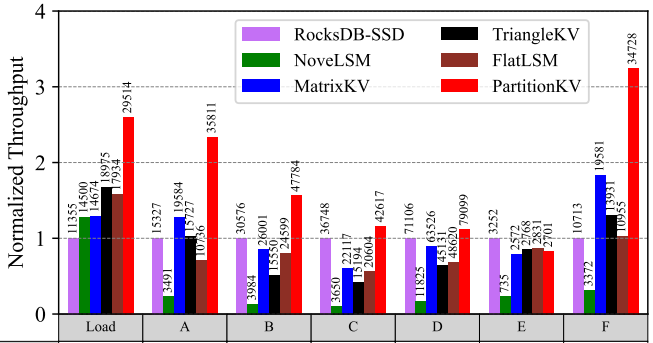
Note. "Total NTI Len." denotes the sum of NTI lengths over all partitions before the random read test begins. In "Total NTI Len.", the first number in each $a * b$ term denotes the NTI length, and the second denotes the number of partitions.

Read performance. To assess their read performances, we first write the same dataset to the databases and then measure random and sequential read performance by querying one million KVs. The random read performance of these databases is shown in Figure 6(a). PartitionKV performs similarly to MatrixKV, followed by TriangleKV, FlatLSM and RocksDB, which have comparable performance, with NoveLSM having the lowest performance. However, when the value size is 64KB, MatrixKV's random read performance exceeds that of PartitionKV. To figure out the reasons behind, we next further analyze the random read performance difference between PartitionKV and MatrixKV. Since the Memtable in MatrixKV is small and has negligible impact on random read, we can infer that the observed performance difference mainly comes from the 8GB NVM. Thus, we conducted experiments focusing on lookups hitting NVM. Note that in the experiments, since NTOs are prioritized for compaction, the length of NTOs in all partitions PartitionKV is zero. As shown in Table 2, that PartitionKV exhibits lower average lookup latency than MatrixKV, but significantly fewer hits on KVs in NVM. This is because no new KVs were written during the read tests, and KVs in NVM were continuously compacted to L_1 in SSD. MatrixKV's slower compaction results in a higher number of NVM hits. Hence, the reason why PartitionKV shows lower random read performance than MatrixKV under the value size of 64 KB is because their average lookup latencies are similar, but PartitionKV has far fewer NVM hits.

The sequential read performance of these databases is displayed in Figure 6(b). As shown, PartitionKV performs similarly to RocksDB, lower than MatrixKV and TriangleKV, but higher than FlatLSM and NoveLSM. This is likely due to the lower query performance of the skiplist in PartitionKV during sequential lookups.

From Table 2, we can observe that the average query time of PartitionKV increases with the value size. This is because a larger value size implies fewer queries hit the Mutable NVMTable, while most hits occur at the head of the NTI/NTO, resulting in longer query paths. The length of the NTI has a significant impact on query latency.

5.2.2 Macro-Benchmark. In this section, we use YCSB to evaluate the performance of these databases under different workloads (YCSB-A to YCSB-F), as shown in Figure 7. Under each workload, we first randomly write an 80GB dataset to these databases, with a key size of 16B and a value size of 4096B, then perform one million operations to measure their performance. The accessed keys are generated within the key space using a Uniform, Zipfian, or Latest distribution. Accordingly, at the partition layer, each partition's key range corresponds to a subrange of this key space.



	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%
Range query	0%	0%	0%	0%	0%	95%	0%
RMW	0%	0%	0%	0%	0%	0%	50%
Distribution	Uniform	Zipfian	Zipfian	Zipfian	Latest	Zipfian	Zipfian

Fig. 7. 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.

The experimental results show that in Load, YCSB-A, and YCSB-F, PartitionKV achieves higher throughput than FlatLSM, TriangleKV, MatrixKV, NoveLSM, and RocksDB. PartitionKV is 1.65×, 1.56×, 2.01×, 2.03×, and 2.60× faster than FlatLSM, TriangleKV, MatrixKV, NoveLSM, and RocksDB under Load(i.e., random write). This result is basically consistent with the result in Figure 5(a). In YCSB-B, YCSB-C, and YCSB-D, PartitionKV also achieves higher throughput than MatrixKV, FlatLSM, NoveLSM, and RocksDB. However, in YCSB-E, the throughput of PartitionKV is comparable to that of FlatLSM and MatrixKV, lower than that of RocksDB, and higher than that of NoveLSM. This is because only a few of the queries hit in NVM. These experimental results show that PartitionKV exhibits better performance except in the range query scenario.

5.2.3 Tail Latency. To evaluate the write latency of PartitionKV, we calculate the maximum, average, P90, P99, and P999 latencies under Load and YCSB-A in Table 3 and Table 4, respectively.

The random write latency is shown in Table 3. The average latencies of PartitionKV are the lowest, while the P90, P99, and P999 latencies of PartitionKV are higher than those of FlatLSM. The reasons stated are as follows: (1) PartitionKV implements fine-grained compaction similar to MatrixKV. Additionally, when the number of Immutable NVMTables in the NTO reaches the prespecified number(i.e., 3), it reduces KV ingestion speed, alleviating the pressure of partition layer and decreasing the frequency of slow compactions. As a result, its P99 and P999 latencies are higher than those of FlatLSM. (2) PartitionKV's multi-threaded compaction is more efficient compared to RocksDB and FlatLSM, resulting in the lowest average latency.

The update latency under YCSB-A is shown in Table 4, where each update involves reading a KV, modifying the value, and writing it back. PartitionKV has the lowest average, and P90 latency among these databases. However, its P99 and P999 latencies are higher than those of FlatLSM. These experimental results indicate that PartitionKV reduces write stalls under the adaptive partitioning strategy and multi-threaded compaction strategy, thereby achieving lower write latency.

Table 3. Random Write Latency (μs)

Database	Max.	Avg.	P90	P99	P999
RocksDB-SSD	257163.26	70.36	22.86	1093.63	2148.35
NoveLSM	33067892.73	54.41	26.06	40.99	5394.43
MatrixKV	101253.12	52.39	20.28	1087.15	1093.46
TriangleKV	729808.09	37.49	13.44	1084.41	1092.61
FlatLSM	51707379.71	43.88	**10.68**	**13.53**	**16.61**
PartitionKV	**56202.58**	**22.60**	12.71	582.66	588.63

Table 4. YCSB-A Update Latency (μs)

Database	Max.	Avg.	P90	P99	P999
RocksDB-SSD	159776.77	83.18	116.73	1167.36	1220.61
NoveLSM	31658606.59	342.16	228.22	617.47	32227.33
MatrixKV	78118.91	59.84	87.68	221.18	1212.41
TriangleKV	74776.57	118.77	136.57	1193.98	1232.89
FlatLSM	52546240.51	143.88	65.38	**76.93**	**86.97**
PartitionKV	**35389.44**	**32.09**	**41.70**	89.09	644.61

5.2.4 Analysis of Applicability. PartitionKV persists each KV as it is written, resulting in poor performance under small-granularity write workloads. Consequently, it is more suitable for large-granularity random writes. Moreover, its range query performance (i.e., sequential lookups within a certain key range) is relatively weak. Thus, it is less suitable for scenarios that emphasize continuous KV lookups. In contrast, MatrixKV and TriangleKV persist KVs from the Immutable MemTable to NVM in a batch manner, making them better suited for small-granularity random write workloads. In addition, they exhibit better sequential lookup performance, implying that they are more suitable for scenarios that emphasize continuous KV lookups. FlatLSM achieves the highest sequential write performance and is more suitable for scenarios where L_0–L_1 compaction involves few overlapping SSTables (e.g., nearly sequential write scenarios).

5.3 Extended Performance Evaluation

5.3.1 Analysis of Throughput. To observe how the throughput of these databases changes over time, we record their throughput every 10 seconds, as shown in the Figure 8. Since the write performance of these systems differs, their completion time of writing 80GB data also varies. We can see that the throughput of PartitionKV fluctuates over time, but its lowest throughput is still

much higher than that of the other databases, and it completes the same volume of writes with the fastest speed. Due to the compaction speed between the partition layer and L_1 being slower than the ingestion speed of KVs, PartitionKV will slow down the ingestion speed when the number of Immutable NVMTables in the NTO reaches the prespecified number(i.e., 3), causing a temporary drop in throughput. However, since the adaptive partitioning strategy enables the NTO to be quickly compacted into L_1, the throughput of PartitionKV can recover quickly.

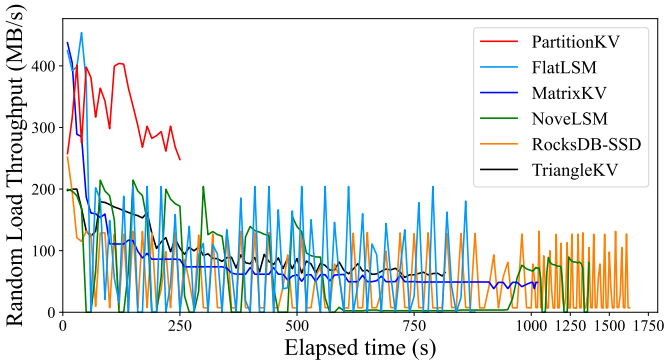


Fig. 8. Throughput fluctuation as a function of time.

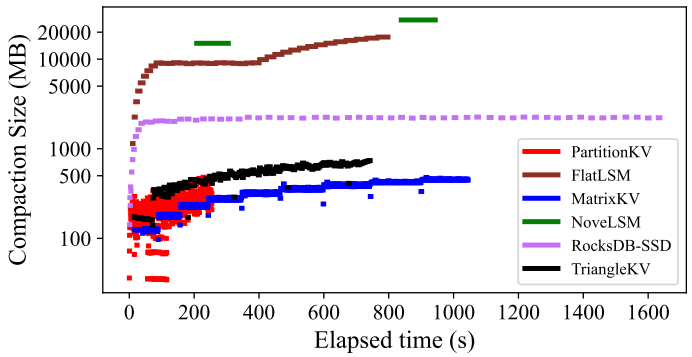


Fig. 9. The L_0-L_1 compaction. Each line segment indicates an L_0-L_1 compaction. The y-axis shows the amount of data involved in the compaction and the length along x-axis shows the duration of the compaction.

5.3.2 Analysis of L_0-L_1 Compaction. To observe the L_0-L_1 compaction behavior of these databases, we record the compaction time and data volume involved, as shown in Figure 9. We can see that the data involved in L_0-L_1 compaction of PartitionKV and MatrixKV ranges between 100MB and 500MB, which is significantly smaller than that of RocksDB, FlatLSM, and NoveLSM. This indicates that the adaptive partitioning strategy keeps the involved data within an appropriate range, achieving fine-grained compaction similar to MatrixKV.

5.3.3 Analysis of Write Amplification. We record their write amplification, as shown in Table 5. We can see that PartitionKV has the smallest write amplification. The actual written data volume in PartitionKV is 1.65×, 1.22×, 1.31×, 1.29×, 1.65× lower than that of FlatLSM, TriangleKV, MatrixKV, NoveLSM, and RocksDB, respectively. The reasons stated are as follows: (1) PartitionKV reuses WAL for storing KVs, eliminating the original Flush operation of Immutable Memtable. (2) KVs with

Table 5. Write Amplification

Database	WA	Write (GB)
RocksDB-SSD	5.14	411.2
NoveLSM	4.01	320.6
MatrixKV	4.08	326.8
TriangleKV	3.81	304.4
FlatLSM	5.15	411.7
PartitionKV	**3.12**	**249.6**

the same key are written to the same partition, and multiple immutable NVMTables in NTI/NTO are compacted together into L_1, reducing write amplification. (3) The overlapping SSTables in L_1 participate in compaction less frequently.

5.3.4 Analysis of Recovery Performance. In this subsection, we test the impact of different value sizes on recovery performance of PartitionKV and FlatLSM, with the value size ranging from 16B to 64KB. The experimental result is shown in Figure 10, which indicates that the recovery time of PartitionKV is lower than that of FlatLSM and their the recovery time decreases as the value size increases. During recovery in PartitionKV (or FlatLSM), NVMTables (or PMTables) require to be restored by rebuilding the skiplists in DRAM based on the NVM Logs (or PM Logs). Their difference is that PartitionKV needs to reconstruct multiple small skiplists because each NVM Log is 64MB, while FlatLSM needs to reconstruct at least two large skiplists because each PM Log is 2GB. However, during skiplist reconstruction, the insertion speed decreases as the number of index nodes increases, resulting in a significant difference in recovery time between PartitionKV and FlatLSM when storing a large number of KVs in NVM.

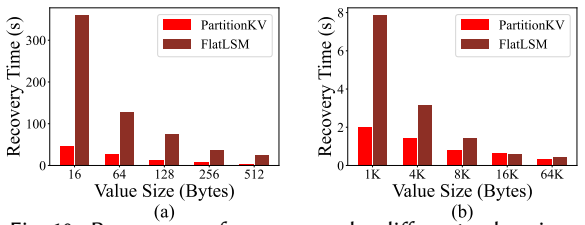


Fig. 10. Recovery performance under different value sizes.

5.4 Internal Performance Evaluation

5.4.1 Ablation Experiments regarding Partition. We use db_bench to conduct experiments for evaluating the performance of PartitionKV under both fixed and adaptive partitioning strategies in random write workloads with different key distributions, with L_0–L_1 compaction handled by a single thread.

As shown in Table 6, under uniform random writes, increasing the number of partitions helps improve write performance and reduce write amplification. This is because more partitions usually imply the following: (1) KVs can be distributed more evenly, leading to finer L_0–L_1 compaction granularity and higher overlap among KVs; and (2) reducing the likelihood of reaching the threshold θ that triggers a slowdown in ingestion speed. Regarding adaptive partitioning, it dynamically adjusts partitions based on predefined thresholds, requiring a gradual adjustment process. As a result, in this scenario, its performance is lower than that of a fixed 60-partition setup. Under Zipfian random writes, the fixed partition strategy exhibits significantly lower write throughput and higher write amplification, as hot key ranges generate many overlapping L_1 files. Adaptive partitioning

Table 6. Fixed vs. Adaptive Partitioning

Distribution	Partition num	Random write (MB/s)	Write amplification
Uniform	30	74.5	3.55
	40	96.0	3.06
	50	118.0	2.72
	60	135.3	2.58
	Ada.	101.5	3.00
Zipfian (0.50)	60	100.2	2.84
	Ada.	98.6	2.92
Zipfian (0.80)	60	51.2	4.03
	Ada.	105.1	2.50
Zipfian (0.99)	60	34.7	5.59
	Ada.	128.0	2.21

splits hot ranges into smaller, non-overlapping partitions, avoiding this issue. In addition, when writes are more skewed toward certain partitions, the higher overlap among KVs involved in compaction leads to higher write performance and lower write amplification.

Therefore, the adaptive partitioning strategy exhibits better general applicability compared to a fixed number of partitions. Moreover, since PartitionKV prioritizes the compaction of frequently written partitions, the data distribution within the partition layer tends to exhibit a smooth Zipfian distribution. Note that in our experiments, the NTI length of most partitions is less than 3.

5.4.2 The Relationship between Compaction Thread Number and Write Performance. To test the impact of different numbers of compaction threads on the random write throughput of PartitionKV, we varied the number of compaction threads from 1 to 5. The experimental result is shown in Figure 11(a), which indicates that throughput increases as the number of compaction threads rises. This is because increasing the number of compaction threads allows NTO to be compacted into L_1 more quickly, accelerating NVM space release for incoming KV writes.

5.4.3 The Relationship between Maximum Number of Partitions and Write Performance. To test the impact of different maximum number of partitions on the throughput of PartitionKV, we set the maximum number of partitions of PartitionKV to 20, 30, 40, 50, and 60, respectively. The experimental result is shown in Figure 11(b), which indicates that throughput increases as the maximum number of partitions rises. This is because the maximum number of partitions affects the granularity of compaction. Appropriately increasing the number of partitions can reduce the amount of data involved in NTO compaction. PartitionKV is not sensitive to δ_{min} and δ_{mid}, since they represent only a transitional phase in the early stage. In our experiments, the number of partitions remains roughly stable at around 56, and the impact on throughput under large-scale data ingestion is minimal, fluctuating around 12MB/s.

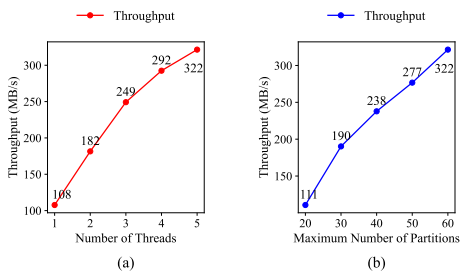


Fig. 11. Write performance under different number of threads or maximum number of partitions.

5.4.4 Breakdown of Time Overhead in Data Writing. We record the average time cost of each stage during data writing, with the value size being set as 128B and 4KB, respectively, as shown in Figure 12. NVM Data represents the time required to write KVs to NVM Log. DRAM Index represents the time required to add index nodes to the skiplist. Wait Compaction represents the time spent waiting for compaction during KVs writes. Split Merge represents the time required for partition splitting and merging. Others include the time costs of locks, B+ tree index time, and other factors. The results show that partition splitting and merging take negligible time, with most time spent adding index nodes to the skiplist, followed by writing KVs to NVM.

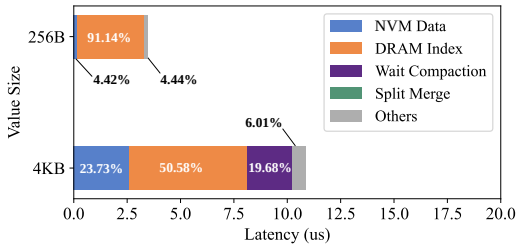


Fig. 12. The average time cost of each stage during the data writing process. Parts not shown in the figure account for nearly 0%.

6 Related Work

Optimizing LSM-tree based KV stores on SSD. To mitigate the write amplification of the leveling policy, many studies have proposed various compaction strategies to reduce unnecessary data rewriting. Spooky [18] partitions the two largest levels into separate key ranges to reduce the proportion of KVs that do not require updates during compaction. WiscKey [35] stores only keys and the addresses of values in SSTables, thereby avoiding rewriting values during compaction. BlockDB [47] reduces write amplification by changing the minimum compaction unit from the SSTable level to the block level, thereby reducing unnecessary merging of unmodified KVs. ALDC [12] reduces write amplification by accumulating more KVs within the same key range in L_i before performing the actual L_i-L_{i+1} compaction.

In addition, a few studies have also focused on mitigating the write stalls under the leveling policy. TRIAD [7] keeps frequently updated KVs in the MemTable, thereby reducing the number of overlapping files and minimizing subsequent compactions on these keys. SILK [8] introduces an I/O scheduler that prioritizes the L_0–L_1 compaction thread and the MemTable flush thread. CruiseDB [33] removes L_0 and merges MemTables directly with L_1 files, while adjusting I/O priorities and throttling user writes based on predicted system throughput, reducing write stalls and improving write stability.

Optimizing LSM-tree based KV stores on NVM. Some studies primarily leverage NVM to redesign L_0 in order to reduce the write stalls and write amplification caused by L_0-L_1 compaction. SLM-DB [28] reduces write amplification using a single-level structure and improves reads via a global B+ tree, but the high maintenance cost of the tree limits overall performance. NoveLSM [30] alternates between a DRAM MemTable and an NVM MemTable to ingest incoming KVs. However, it does not mitigate the excessive number of overlapping files in L_0–L_1 compactions. MatrixKV [55] replaces L_0 with ordered matrix containers to enable fine-grained column compaction. TriangleKV [19] builds upon MatrixKV by including newly written KVs in compaction, reducing L_1 file involvement, lowering write amplification, and improving write performance. FlatLSM [25] persists incoming KVs in NVM and maintains corresponding in-memory indexes, removing the

WAL. It also employs multithreading to accelerate L_0-L_1 compaction. BushStore [49] stores KVs from the Memtable, L_0, and L_1 in NVM, while maintaining in-memory B+ trees to replace the original indexes. During L_0-L_1 compaction, only the B+ trees and relevant NVM data are updated, reducing compaction overhead. ZigZagDB [48] introduces an NVM intermediate layer $L_{i+0.5}$ between L_i and L_{i+1}. When KVs in L_i need to be compacted to the next level, they are first moved to $L_{i+0.5}$ and partitioned by key. However, the large NVM space consumed by the underlying levels leads to lower write performance compared to MatrixKV.

7 Conclusion

In this paper, we propose PartitionKV, an LSM-tree-based KV store tailored for DRAM–NVM–SSD architectures. PartitionKV introduces an ordered partition layer spanning DRAM and NVM, which both persists KV writes and functions as the WAL. Together with adaptive partitioning and parallel compaction, this design significantly reduces write amplification and alleviates write stalls. Evaluation on Intel Optane PM demonstrates that PartitionKV achieves higher throughput, lower write latency, and markedly reduced write amplification compared with state-of-the-art systems under random-write workloads.

8 Acknowledgements

This work was supported by the Jing-Jin-Ji Regional Integrated Environmental Improvement–National Science and Technology Major Project of Ministry of Ecology and Environment of China under Grant 2025ZD1200600, the National Natural Science Foundation of China (NSFC) under Grants 62372360, 62372352, and 62332014, the Youth Elite Scientist Sponsorship Program by China Association for Science and Technology under Grant YESS20220610, and the Innovation Capability Support Program of Shaanxi Province under Grant 2024ZC-KJXX-021.

References

[1] 2025. PartitionKV. https://github.com/hxy114/PartitionKV/tree/test2.
[2] 2025. PMDK: Persistent Memory Development Kit. https://github.com/pmem/pmdk.
[3] 2025. Reproducing TriangleKV on MatrixKV. https://github.com/hahah114/Reproducing-TriangleKV-on-MatrixKV.
[4] 2025. Reproduction of FlatLSM. https://github.com/hxy114/flatlsm-r.
[5] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. 2018. Operating Systems: Three Easy Pieces. (2018).
[6] Anurag Awasthi, Avani Nandini, Arnab Bhattacharya, and Priya Sehgal. 2012. Hybrid HBase: Leveraging Flash SSDs to Improve Cost per Throughput of HBase. In *COMAD*. 68–79.
[7] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *2017 USENIX Annual Technical Conference*. 363–375.
[8] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference*. 753–766.
[9] Meenakshi Sundaram Bhaskaran, Jian Xu, and Steven Swanson. 2014. Bankshot: caching slow storage in fast non-volatile memory. (2014), 73–81.
[10] Edward Bortnikov, Anastasia Braginsky, Eshcar Hillel, Idit Keidar, and Gali Sheffi. 2018. Accordion: better memory organization for LSM key-value stores. *Proceedings of the VLDB Endowment* (2018), 1863–1875.
[11] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies*. 209–223.
[12] Yunpeng Chai, Yanfeng Chai, Xin Wang, Haocheng Wei, and Yangyang Wang. 2022. Adaptive Lower-Level Driven Compaction to Optimize LSM-Tree Key-Value Stores. *IEEE Transactions on Knowledge and Data Engineering* (2022), 2595–2609.
[13] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems* (2008), 1–26.

[14] Guoqiang Jerry Chen, Janet L Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. 2016. Realtime Data Processing at Facebook. In *Proceedings of the 2016 International Conference on Management of Data*. 1087–1098.

[15] Shimin Chen and Qin Jin. 2015. Persistent B+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment* (2015), 786–797.

[16] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGARCH Computer Architecture News* (2011), 105–118.

[17] Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. 2019. 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*. 441–454.

[18] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. 2022. Spooky: granulating LSM-tree compactions correctly. *Proceedings of the VLDB Endowment* (2022), 3071–3084.

[19] Chen Ding, Ting Yao, Hong Jiang, Qiu Cui, Liu Tang, Yiwen Zhang, Jiguang Wan, and Zhihu Tan. 2022. TriangleKV: Reducing Write Stalls and Write Amplification in LSM-Tree Based KV Stores With Triangle Container in NVM. *IEEE Transactions on Parallel and Distributed Systems* (2022), 4339–4352.

[20] Alexander Driskill-Smith. 2010. Latest advances and future prospects of STT-RAM. In *Non-Volatile Memories Workshop*. 11–13.

[21] Zhuohui Duan, Haikun Liu, Xiaofei Liao, Hai Jin, Wenbin Jiang, and Yu Zhang. 2019. HiNUMA: NUMA-Aware Data Placement and Migration in Hybrid Memory Systems. In *2019 IEEE 37th International Conference on Computer Design*. 367–375.

[22] Zhuohui Duan, Haodi Lu, Haikun Liu, Xiaofei Liao, Hai Jin, Yu Zhang, and Song Wu. 2021. Hardware-supported remote persistence for distributed persistent memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[23] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–16.

[24] Sanjay Ghemawat and Jeff Dean. 2016. LevelDB. https://github.com/google/leveldb.

[25] Kewen He, Yujie An, Yijing Luo, Xiaoguang Liu, and Gang Wang. 2023. FlatLSM: Write-Optimized LSM-Tree for PM-Based KV Stores. *ACM Transactions on Storage* (2023), 1–26.

[26] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies*. 187–200.

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

[28] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H Noh, and Young-ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *17th USENIX Conference on File and Storage Technologies*. 191–205.

[29] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. NoveLSM. https://github.com/sudarsunkannan/lsm_nvm.

[30] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *2018 USENIX Annual Technical Conference*. 993–1005.

[31] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. 399–411.

[32] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review* (2010), 35–40.

[33] Junkai Liang and Yunpeng Chai. 2021. CruiseDB: An LSM-Tree Key-Value Store with Both Better Tail Throughput and Tail Latency. In *2021 IEEE 37th International Conference on Data Engineering*. 1032–1043.

[34] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. *ACM SIGPLAN Notices* (2017), 329–343.

[35] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. WiscKey: Separating Keys from Values in SSD-Conscious Storage. *ACM Transactions On Storage* (2017), 1–28.

[36] Meta. 2025. RocksDB: A Persistent Key-Value Store for Flash and RAM Storage. http://rocksdb.org/.

[37] Meta. 2025. Subcompaction in RocksDB. https://github.com/facebook/rocksdb/wiki/Subcompaction.

[38] Iulian Moraru, David G Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. 2013. 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. 1–17.
[39] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th annual international symposium on Computer architecture*. 24–33.
[40] Andy Rudoff. 2017. Persistent memory programming. *Login: The Usenix Magazine* (2017), 34–40.
[41] Samsung. 2025. CXL Memory Module - Box (CMM-B). https://semiconductor.samsung.com/news-events/tech-blog/cxl-memory-module-box-cmm-b/.
[42] Samsung. 2025. Samsung CXL Solutions – CMM-D. https://semiconductor.samsung.cn/cxl-memory/cmm-d/.
[43] Samsung. 2025. Samsung CXL Solutions – CMM-H. https://semiconductor.samsung.com/news-events/tech-blog/samsung-cxl-solutions-cmm-h/.
[44] Mohammadreza Soltaniyeh, Gongjin Sun, Xuebin Yao, Amir Beygi, Ramdas Kachare, Dongwan Zhao, Hingkwan Huen, Andrew Chang, Senthil Murugesapandian, and Caroline Kahn. 2025. Revisiting Memory Hierarchies with CMM-H: Use Device-side Caching to Integrate DRAM and SSD for a Hybrid CXL Memory. In *Proceedings of the 17th ACM Workshop on Hot Topics in Storage and File Systems*. 45–51.
[45] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. 2008. The missing memristor found. *Nature* (2008), 80–83.
[46] Guoyu Wang, Xilong Che, Haoyang Wei, Shuo Chen, Puyi He, and Juncheng Hu. 2025. Boosting File Systems Elegantly: A Transparent NVM Write-ahead Log for Disk File Systems. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. 19–34.
[47] Xiaoliang Wang, Peiquan Jin, Bei Hua, Hai Long, and Wei Huang. 2022. Reducing write amplification of LSM-tree with block-grained compaction. In *2022 IEEE 38th International Conference on Data Engineering*. 3119–3131.
[48] Yi Wang, Jiajian He, Kaoyi Sun, Yunhao Dong, Jiaxian Chen, Chenlin Ma, Amelie Chi Zhou, and Rui Mao. 2024. Boosting Write Performance of KV Stores: An NVM - Enabled Storage Collaboration Approach. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 2082–2095.
[49] Zhenghao Wang, Lidan Shou, Ke Chen, and Xuan Zhou. 2024. BushStore: Efficient B+Tree Group Indexing for LSM-Tree in Non-Volatile Memory. In *2024 IEEE 40th International Conference on Data Engineering*. 4127–4139.
[50] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. 2015. Overcoming the challenges of crossbar resistive memory architectures. In *2015 IEEE 21st international symposium on high performance computer architecture*. 476–488.
[51] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. 2019. 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*. 427–439.
[52] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies*. 169–182.
[53] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. 2020. Leaper: a learned prefetcher for cache invalidation in LSM-tree based storage engines. *Proceedings of the VLDB Endowment* (2020), 1976–1989.
[54] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV. https://github.com/PDS-Lab/MatrixKV.
[55] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *2020 USENIX Annual Technical Conference*. 17–31.

Received 15 July 2025; revised November 2025; accepted November 2025