Received July 29, 2020, accepted August 9, 2020, date of publication August 18, 2020, date of current version September 2, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3017651

RangeKV: An Efficient Key-Value Store Based on Hybrid DRAM-NVM-SSD Storage Structure

LING ZHAN[1], KAI LU[2], ZHILONG CHENG[2], AND JIGUANG WAN[2]
[1]Division of Information Science and Technology, Wenhua University, Wuhan 430074, China
[2]Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430074, China

Corresponding author: Jiguang Wan (jgwan@mail.hust.edu.cn)

This work was supported by Natural Science Foundation of Hubei Province of China (No: 2018CFB774), the Scientific Research Project of the Education Department of Hubei Province (No: B2018335).

ABSTRACT Persistent key-value (KV) stores are an integral part of storage infrastructure in data centers. Emerging non-volatile memory (NVM) technologies are potential alternatives for future memory architecture design. In this study, we use NVM to optimize the KV store and propose RangeKV, an LSM-tree based persistent KV store built on a heterogeneous storage architecture. RangeKV uses RangeTab in NVM to manage L0 data and increases L0 capacity to reduce the number of LSM tree levels and system compactions. RangeKV pre-constructs the hash index of RangeTab data to reduce NVM access times and adopts a double-buffer structure to reduce LSM-tree write amplification due to compactions. We implement RangeKV based on RocksDB and conduct a comparative test and performance evaluation with RocksDB and NoveLSM. The test results show that the overall random write throughput is improved by 4.5× to 5.7× compared to RocksDB. In addition, RangeKV has a significant performance advantage over NoveLSM.

INDEX TERMS Key-value store, log-structured merge-tree, hybrid storage, non-volatile memory, hash index.

I. INTRODUCTION

Persistent key-value (KV) stores are an integral part of storage infrastructure in data centers and have been used in many applications including cloud storage [1], online games [2], advertising [3], [4], e-commerce [5], web indexing [3], [6], and social networks [4], [7], [8]. KV stores can be divided into three categories depending on the index structure used: hash index-based design [9], [10], B-tree-based design [11], and log-structured merge (LSM)-tree-based design [12]. Among them, the LSM-tree-based KV stores, such as BigTable [3], Cassandra [13], LevelDB [6], [14], RocksDB [4], and Hbase [15], are state-of-the-art persistent KV stores for write-intensive workloads.

These widely used KV stores are mainly deployed on a DRAM-SSD hybrid architecture. Because of the limitations of DRAM, it is difficult to further improve the system performance by increasing the DRAM size [16]. On the other hand, emerging non-volatile memory (NVM) technologies, such as phase-change RAM (PCRAM) [17], memristors [18], and spin-torque transfer RAM (STT-RAM) [19], are potential alternatives for future memory architecture design owing to their non-volatility, byte addressability, high density, good scalability, and low leakage power.

Therefore, enhancing storage systems (e.g., KV stores) with NVM is seen as a cost-efficient choice. However, it would be unwise to directly replace DRAM with NVM, because NVM is 10−15× slower than DRAM in terms of the write performance and 3−5× slower in terms of the read performance [20]. Moreover, a direct replacement of SSDs is not cost-effective, because merely adapting NVM as an alternative persistent storage device would bring only a satisfactory performance, compared to its high device performance [21]. Therefore, incorporating NVM into the traditional DRAM-SSD storage architecture and fully utilizing NVM with memory access is a recognized solution in academia and industry.

To improve KV storage with NVM, an approach called NoveLSM [21] based on heterogeneous storage architectures (DRAM, NVM, and SSD) has been proposed; it uses an

The associate editor coordinating the review of this manuscript and approving it for publication was Cristian Zambelli.

NVM to store memtables (files in the memory of LSM-tree) of larger size, such as 8 GB, and make them mutable to reduce compaction. However, our test results demonstrate that NoveLSM has significant performance fluctuations and delay overhead because of design flaws. Specific test results and analysis are presented in Section 2.

To further improve the system performance, overcome the drawbacks of NoveLSM, and reduce the write amplification of LSM-tree, in this paper, we propose RangeKV, an LSM-tree-based persistent KV store built on a DRAM-NVM-SSD storage architecture. RangeKV has four key features:

A. SHORTER COMPACTION DELAYS

RangeKV uses multiple RangeTab structures in the NVM to organize the L0 level in the LSM tree, to shorten the compaction delay. RangeTab stores flushed data from DRAMs in the form of chunks as the basic unit.

B. REDUCING THE NUMBER OF LSM-TREE LEVELS

RangeKV increases the capacity of L0, which helps reduce the numbers of LSM-tree levels and system compactions.

C. OPTIMIZING READ PERFORMANCE

To improve the lookup performance, RangeKV pre-constructs the hash index of the data by the RangeTabs to reduce the number of times the storage medium is accessed.

D. IMPROVING COMPACTION EFFICIENCY

RangeKV adopts a double-buffer structure in the compaction process of the LSM-tree to reduce the delay in system write blocking and write amplification. In addition, RangeKV preferentially selects RangeTabs with the lowest write amplification ratio to participate in the compaction to improve the compaction efficiency.

We implement RangeKV based on RocksDB and conduct a comparative test and performance evaluation with RocksDB and NoveLSM. The test results show that the random write performance of RangeKV is 4.5−5.7× that of RocksDB, the number of compactions is more than 50% lower than that of RocksDB, and the average merged data volume is reduced by approximately 40%. The write amplification of the system is only approximately 25% of that of RocksDB. In addition, RangeKV has a significant performance advantage over the NoveLSM solution.

The rest of this paper is organized as follows. Section 2 describes the background and motivation behind our work. We introduce the design and implementation details of RangeKV in Sections 3 and 4. Section 5 presents our comprehensive experiments and evaluation result analysis. Related work is discussed in Section 6. Finally, Section 7 concludes the paper.

II. BACKGROUND AND MOTIVATION

A. LOG-STRUCTURED-MERGE-TREES IN RocksDB

RocksDB is a popular LSM-tree based KV store. It effectively converts small random writes into sequential writes, thus improving system write performance. In RocksDB, first, an LSM-tree batches the writes in a fixed-size in-memory buffer called *memtable*. When the memtable is full, it will be converted to an *immutable memtable*, and the batched data are then flushed to the storage as *SSTables*. The SSTables in the storage devices are sorted and stored in multiple levels and merged from lower levels to higher levels (e.g. L0, L1..., Ln level). The level size increases exponentially by an amplification factor (e.g., AF=10). The process of merging and cleaning SSTables is called *compaction*. Compaction is conducted throughout the lifetime of an LSM-tree to clean invalid/stale KV items and keep the data sorted on each level for efficient reads [12], [22].

However, compaction inevitably incurs repeated reads and writes, resulting in higher *read and write amplifications* [23], [24], which adversely affects the system performance. Therefore, when designing an LSM-tree-based KV store, the optimization of the read and write amplifications must be considered.

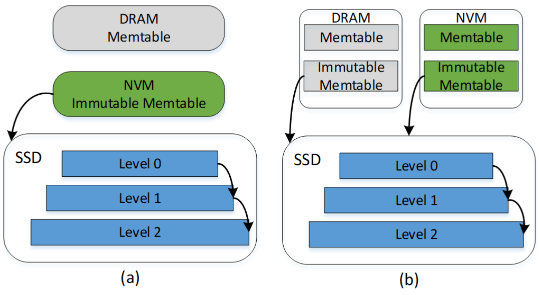


FIGURE 1. The architecture of NoveLSM.

B. NVM-BASED KEY-VALUE STORES

1) A RELATED WORK: NoveLSM

System researchers have considered improving KV stores via exploiting NVM in a heterogeneous storage architecture. However, the performance improvement of a KV system using NVM directly as a data storage medium is not evident, mainly because it fails to take full advantage of the NVM features such as byte addressability. To fully utilize the characteristics of the NVM and high I/O bandwidth to improve the random write performance of LSM-tree-based KV stores, NoveLSM was developed with two schemes using NVM and SSD as the hybrid storage structures: in Scheme 1, the NVM stores the immutable data, as shown in Figure 1 (a); when the memtable data in the DRAM is almost full, the data are written to the NVM memtable, as shown in Figure 1 (b). To better solve the problem wherein the system may block while processing write requests when memtable is written to NVM, NoveLSM finally chooses scheme 2.

2) DRAWBACKS OF NoveLSM

This study evaluates NoveLSM by randomly writing the same 80 GB database. NoveLSM uses an 8 GB NVM. By analyzing the test results shown in Figure 2, we find the following problems:

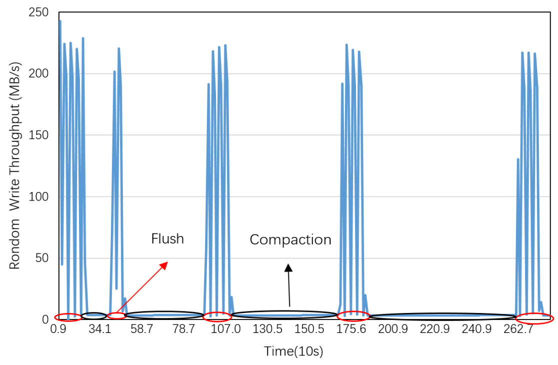


FIGURE 2. Random write performance of NoveLSM measured in every 10 seconds.

(1) The NVM memTable in NoveLSM has a higher capacity. When it is full, the data need to be flushed to the SSD. However, the amount of data (4 GB) is high, and the system performance decreases significantly. In the time interval corresponding to the red elliptical coverage area in Figure 2, frequent fluctuations can be observed in the system performance.

(2) When the number of L0-level SSTables is too high, compaction is triggered. Because of the high amount of merged data, the flush thread blocking wait time is extended, which corresponds to the time interval corresponding to the curve covered by the black ellipse in Figure 2.

(3) The greater the amount of data the system writes, the higher the number of compactions and the amount of data between the LSM-tree levels. The longer the compaction thread takes, the longer the wait time for the flush thread to block, but the longer the write performance of the system is at the lowest level. Therefore, the time range indicated by the black elliptical coverage area in Figure 2 is also gradually extended.

Unfortunately, NoveLSM does not include specific optimization measures for the compaction operation of the LSM-tree. The main bottleneck in the system remains the performance of the data writing to the SSD.

Based on the detailed analysis of the NoveLSM scheme, this paper proposes a performance optimization KV store, **RangeKV**, based on a hybrid storage structure. By reducing the number of LSM-tree levels and the compaction time, RangeKV reduces the system write amplification ratio, increases the bandwidth of processing request, and achieves the purpose of improving the random write performance.

III. RangeKV

A. DESIGN OF RangeKV

We present RangeKV, an LSM-tree based persistent KV store built on a three-level heterogeneous storage system. Figure 3 shows the architecture of RangeKV. RangeKV is divided into three levels in terms of the storage structure: DRAMs, NVMs, and SSDs. In this system: 1) The DRAM layer design is the same as that in RocksDB, which batches write requests with memtable and immutable memtable. 2) The NVM layer

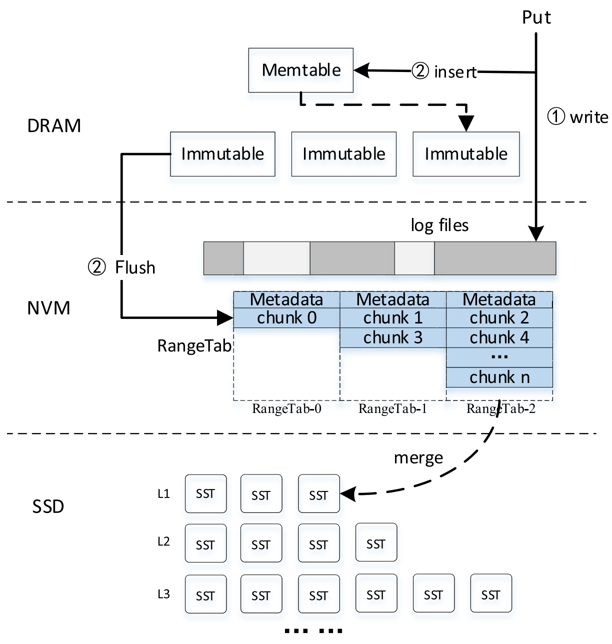


FIGURE 3. System architecture of RangeKV.

contains log and RangeTabs. The log keeps the consistency and completeness of the data in the volatile DRAM. RangeTab manages the unsorted and overlapped L0 that moves from the SSDs. Data are merged into L0 through a new compaction strategy. 3) The other levels of the LSM-tree except L0 are stored in the SSDs.

Design Objectives: Compared with RocksDB and NoveLSM, RangeKV has the following three design objectives:

1) SHORTER COMPACTION DELAYS

In RocksDB, the key ranges of SSTables in L0 are unordered and may overlap. Under compaction, the wider key ranges are easy to read and write more data, and the compaction delay is increased. RangeKV uses the RangeTab structure to reorganize KV pairs in L0 to shorten the compaction delay.

RangeKV maps multiple RangeTabs into different and non-overlapping key ranges. The memtable data are flushed to the corresponding RangeTab depending on the key range. Only one or a few adjacent RangeTabs participate in the compaction at a time. This helps shorten the actual key ranges involved in the compaction and avoid the participation of excessive data in the lower level, thus effectively improving the compaction efficiency and reducing the delay in system data write blocking.

2) REDUCING THE NUMBER OF LSM-TREE LEVELS

The more the LSM-tree levels, the higher the number of times data are merged when the system writes the same amount of data, which increases the write amplification. In particular, when the amount of data written to the system is high, the influence of the number of levels on the write performance of the system is more evident. RangeKV increases the data capacity of all RangeTabs by appropriately increasing the number of RangeTabs. Under the condition that the ratio of adjacent level capacity remains unchanged, RangeKV can accommodate more data per level while reducing the numbers of LSM-tree levels and compactions.

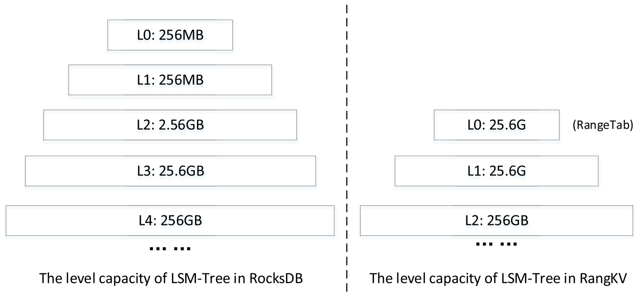


FIGURE 4. Reducing levels in RangeKV.

As shown in Figure 4, it is assumed that RangeKV has a total of 100 RangeTabs, each having a size of 256 MB. Under the premise that the other factors remain unchanged, the data capacity of the entire L0 level is expanded from 256 MB to 25.6 GB. When the system writes the same amount of data, RangeKV can reduce the actual number of LSM-tree levels, thereby reducing the number of data merges, to alleviate the impact of compaction operation on the system write performance.

3) OPTIMIZING READ PERFORMANCE

RocksDB uses in-memory bloom filters [22] to avoid unnecessary searching in some levels. However, the bloom filters have an error rate problem. There may be overlaps in the key ranges when searching for data in the L0 level. When using dichotomy to find data in each SSTable file, we need to access the storage medium multiple times, which significantly affects the efficiency of the system in finding data in the L0 level.

RangeKV pre-constructs the hash index of the data by the RangeTab stored in the NVM to reduce the number of times the storage medium is accessed. The data stored in the SSD are still organized in the SSTable file, and the search method remains unchanged. The hash indexing method of RangeTab can significantly reduce the number of accesses to NVM, accelerate the search, and indirectly improve the read performance of the RangeKV system.

B. RangeTab

RangeKV moves L0 from SSD to NVM and reorganizes data in L0 with RangeTabs. As shown in Figure 5, the RangeTab structure consists of metadata, index, and data. The metadata area is used to record the metadata of the RangeTab, such as the size of RangeTab in the NVM, the size of the used physical space, and the starting physical offset address. The data area stores data, and the index area consisting of cur and seq records some index information of the data. Cur is used to

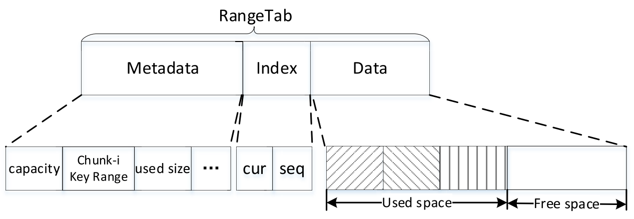


FIGURE 5. RangeTab of RangeKV.

obtain the offset address of the data to be found, and the initial value of seq is 0; when data are written in RangeTab, its value is automatically incremented by 1. When data are merged in the next level, its value is reset to 0. When the system is restarted, we can compare whether the seq value is equal to 0 to determine whether the system needs to reconstruct the RangeTab structure.

- **Chunk**. The data in the data area are stored in the form of chunks as the basic unit, and the chunk structure is designed as shown in Figure 6. The data area is composed of multiple chunks whose physical addresses are consecutive, and each chunk records data and metadata. All the keys stored in the chunk data are ordered. The relative offset address of each key is recorded in turn, and the number of keys in the chunk is recorded last.

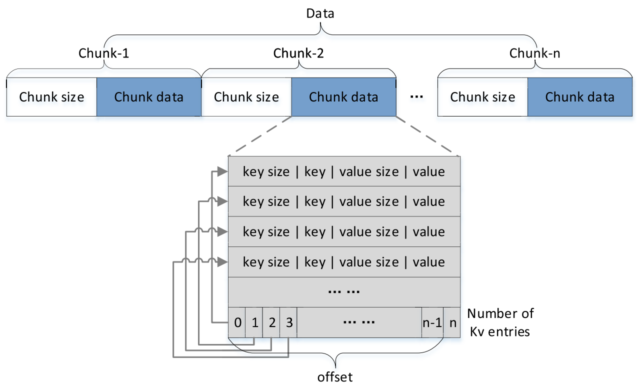


FIGURE 6. Chunk structure of RangeTab.

- **Hash Index**. In chunks, we can use the binary search method to find keys. However, this method requires accessing the NVM each time a key is compared. Repeatedly accessing the NVM reduces the overall search efficiency of RangeTabs. To improve the efficiency of finding keys in the chunk, RangeKV constructs a hash index in each group of several chunks in the RangeTabs. When looking up a key in a chunk, we need to first obtain an address after the key is calculated by the hash function. We can then obtain the offset address of the key in the hash address space through this address. Finally, we can find the data in the chunk from the offset address. Compared with the binary search method, using the hash index to find the offset address of the data can

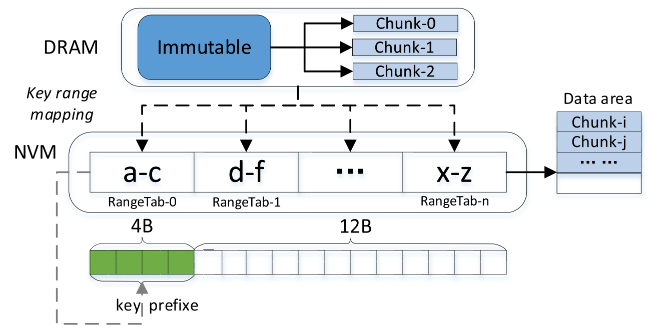


FIGURE 7. Key range mapping of RangeTab.

significantly reduce the number of NVM accesses and improve the search efficiency of the RangeTab data.

- **Key Range Mapping**. When the immutable memtable is written to each RangeTab in the NVM, RangeKV divides the key ranges based on the key prefix for RangeTab to distribute the adjacent keys to the adjacent physical address, thus ensuring that the keys of each chunk in the same RangeTab have the same prefix. As shown in Figure 7, suppose that the size of the key is 16 bytes and the prefix takes the first 4 bytes, the range of 4 bytes the prefix determines the key range of RangeTab. Before the immutable memtable is written to the NVM, multiple keys with the same key prefixes are organized in the chunks and mapped to RangeTabs with the same prefixes based on the prefixes of the keys. The data are written to the data area of RangeTabs.

C. COMPACTION IN RangeTab

1) DOUBLE-BUFFER STRUCTURE

When a RangeTab is selected to participate in the compaction process, a flush thread may want to write chunk data to the RangeTab. Generally, this can be solved in one of two ways: blocking the flush thread until the compaction process of all data in the RangeTab ends or temporarily allocating an additional physical space in the NVM, temporarily writing the data to the physical space, and waiting for the RangeTab data compaction to complete. Finally, the data of the physical space are moved back to the RangeTab.

For the former, blocking the flush thread to write data can be easily implemented with code. However, the efficiency of the system thread processing request is considerably reduced because the flush thread is blocked and cannot complete the data flush process. The latter scheme will increase unnecessary data migration overhead and reduce the efficiency of the flush thread flushing data to the RangeTab.

To solve this problem, RangeKV adopts a double-buffer structure as shown in Figure 8. Each RangeTab pre-allocates the original double-sized physical space termed buffer-1 and buffer-2 when it is initialized to allocate the data area space. When RangeTab is initialized, neither buffer-1 nor buffer-2 stores any chunk data, and its state is set to *init_state*.

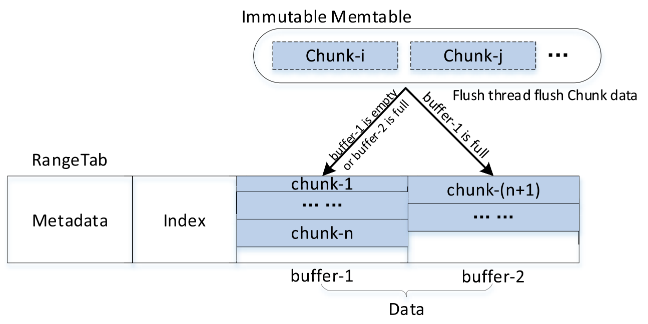


FIGURE 8. Double-buffer structure of RangeTab.

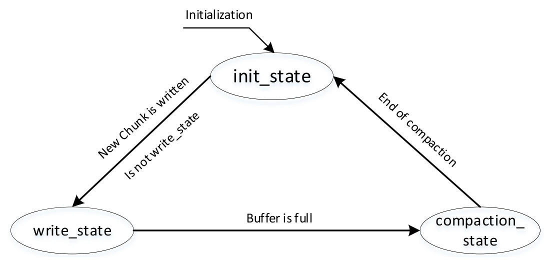


FIGURE 9. Buffer state transition.

Figure 9 shows the transition between the states. When the flush thread is ready to write chunk data to the RangeTab, since the buffer-1 and buffer-2 states are *init_state*, we need to first modify the buffer-1 state to *write_state*, indicating that its current state can store chunk data. The chunk data are then written to buffer-1. When the remaining physical space of buffer-1 is almost exhausted, its state will be converted from *write_state* state to *compaction_state*, indicating that the data of the block physical space will be or is being merged. If there are new chunk data to be flushed into RangeTab, we cannot continue to write to buffer-1. At this time, if the buffer-2 state is *init_state*, it is converted to *write_state*. Only when the state of the buffer is *write_state*, the chunk data can be written. Otherwise, the write process will be blocked until the state of a buffer is changed to *write_state*.

2) COMPACTION TRIGGER STRATEGY

If the amount of data for a compaction is too high, the blocking delay of the flush thread may be too long, and the system write performance will fluctuate considerably. If the amount of data is too low, the number of compactions will increase. There are two disadvantages: first, the read and write operations for a small amount of data cannot fully utilize the high I/O bandwidth of the NVM; second, the SSTable is selected as the unit for compaction in the lower level of the SSD. Less data to be merged in the RangeTab does not shorten the key range of the actual merged data, but may increase the proportion of underlying SSTables, which will increase the write amplification ratio of the compaction operation.

Therefore, the method whereby the data size is chosen to participate in the compaction determines the efficiency and indirectly affects the write performance of the system.

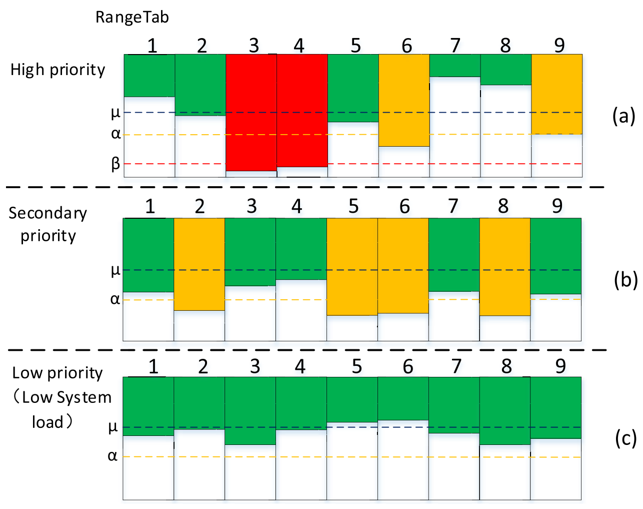


FIGURE 10. **RangeTab compaction trigger threshold.**

As shown in Figure 10, the red-filled rectangle indicates that the physical space of a block of RangeTab is running out. The yellow one indicates that the buffer stores more data and that there is less physical space. The green-filled rectangle indicates that there is sufficient available physical space. RangeKV sets three thresholds for the proportion of the physical space occupied by each RangeTab data: μ, α and β, and $\beta > \alpha > \mu$.

RangeTab merge trigger threshold analysis: In Figure 10(a), RangeTabs numbered 3 and 4 are filled with red, the ratio of physical space used exceeds β, and the available space is almost exhausted. These data need to be merged into the lower level as soon as possible; otherwise, they will block the flush thread. They need to be given the highest priority. When the average ratio of all RangeTab data volumes is greater than α, there must be some RangeTab data ratios exceeding α, such as RangeTabs numbered 2, 5, 6, and 8 in Figure 10(b). To minimize the probability of subsequent flush thread blocking, these RangeTabs should also be merged into the lower level, but need not start processing immediately, and the corresponding thread can have a lower priority.

To reduce the number of unnecessary compactions, only when the system CPU and I/O load are low, and the average of all RangeTab data ratios reaches μ, only those RangeTabs whose data amount ratio is in the range of (μ, α) are considered, as shown in Figure 10(c). Multiple RangeTabs selected for compaction with a low write magnification can increase. To prevent the key range from being too wide, it is necessary to ensure that the range of RangeTab keys selected is adjacent and that the number of RangeTabs and the total amount of data participating in the compaction is not excessive. In the case of a low system load, appropriately increasing the compaction write amplification does not significantly affect the write performance of the system. In summary, meeting any of the following three conditions can trigger RangeTab data compaction in RangeKV:

1) The physical space available for a RangeTab is running out, i.e., the amount of data exceeds β;

2) The average of all RangeTab data volume ratios (or the ratio of total data to total capacity) reaches a higher value $\alpha (\alpha < \beta)$;

3) When the system load is low, and the average of all RangeTab data volumes to capacity ratios (or the ratio of total data to total capacity) reaches a higher value $\mu (\mu < \alpha)$.

3) RangeTab SELECTION ALGORITHM

The steps involved in the RangeTab selection algorithm are as follows:

(1) If the ratio of the amount of RangeTab data to the capacity exceeds the threshold β ($\beta > \alpha$), i.e., 90%, these RangeTabs are ranked from high to low by the ratios of the amount of data to the total capacity, and are merged to the lower level from the highest ratio in turn. This is because these RangeTabs are more likely to block the flush threads, which require the highest priority.

(2) Second, for the RangeTabs whose data volume-to-capacity ratios reach the threshold α, we calculate the ratio of the data volume of each RangeTab to the number of SSTables overlapped by the adjacent lower level, and select the RangeTab with the highest ratio to participate in the compaction. This is done to reduce the actual write amplification of the compaction as much as possible.

(3) Finally, when the system load is low, and the average of all RangeTab data-to-capacity ratios is greater than μ, only the RangeTabs (there is no RangeTab with a ratio greater than α at this time) with data volume-to-capacity ratios less than α but greater than μ (eg $\mu \geq 60\%$) are considered. We randomly select multiple RangeTabs with adjacent key ranges and as much data as possible to participate in the compaction. The number of RangeTabs cannot be too high, in order to ensure that the combined key ranges are always maintained in a narrow range.

Moreover, the ratio of the total RangeTabs involved in the compaction to the upper limit of a single RangeTab capacity cannot exceed the threshold γ, i.e., 2, in order to avoid excessive data volume.

(4) If the first three conditions are not met, the RangeTab will not trigger the compaction. The RangeTabs selected by the RangeTab selection algorithm and the SSTables with overlapping key ranges from the lower level will participate in the compaction together.

By shortening the compaction key ranges and reducing the amount of data actually involved in the compaction, the selection algorithm maximizes the ratio of the RangeTab data volume to the lower SSTables, reduces the write amplification ratio of the LSM-tree compaction, and improves the compaction efficiency and system write performance.

IV. IMPLEMENTATION

We implement RangeKV based on RocksDB and the persistent memory development kit (PMDK) [25] from Intel to access NVM devices. PMDK is a library based on direct access feature (DAX), which allows applications to load/store persistent memory by memory mapping files on a persistent-aware file system. The RangeKV system adopts a leveled modular implementation, which facilitates the code implementation and function expansion of the system. The optimization for the key-value store is mainly concentrated in the RangeKV level.

The RangeKV level provides a variety of API call interfaces for user-level applications, implementing common basic functions of KV store such as Get, Put, Delete, and Seek. The RangeKV level mainly includes four modules: flush process, RangeTab management, compaction process, and PMDK interface. The flush process is responsible for writing the immutable data to the RangeTab physical space in the NVM. The compaction process manages the compaction policy and the compaction operation of all the LSM-tree levels including the process of merging RangeTab data into the lower level. The PMDK interface is responsible for encapsulating some data structures and function interfaces provided by calling the PMDK library file.

The optimization scheme proposed in this paper focuses on the RangeTab structure, which belongs to the L0 level in the original LSM-tree structure. Therefore, RangeTab management is the core part of the optimization scheme.

The main tasks include hashing the key and writing the corresponding KV to its mapped RangeTab based on its hash value, and appending it to the data area of the RangeTab with a chunk structure. When looking up the data in RangeTab, we use a pre-constructed chunk hash index and find the key offset address through the hash value to improve the search efficiency. Moreover, we use the compaction trigger strategy to start the compaction thread to merge RangeTab data into the lower level of the LSM-tree based on the RangeTab selection algorithm.

V. EVALUATION

In this section, we present the evaluation results, which demonstrate the advantages of RangeKV. Specifically, we present the results of extensive experiments conducted to answer the following questions:

1) What are the advantages of RangeKV in terms of the performance? (Section 5.2);

2) What are the effects of different values of RangeKV parameters on the system performance? (Section 5.3). Answering these questions will help optimize RangeKV.

A. EXPERIMENTAL SETUP

All the experiments are run on a test machine from Intel with two Genuine Intel(R) 2.20 GHz processors and 32 GB of memory. The kernel version of the test machine is 64-bit Linux 4.13.9, and the operating system in use is Fedora 27. Two storage devices are used in the experiments: an 800 GB Intel SSDSC2BB800G7 SSD and two 128 GB Apache Pass (i.e., the 3D X-Point NVM device from Intel). The basic read and write performances of the test SSD are 356 MB/s (sequential read), 290 MB/s (random read), 273 MB/s (sequential write), and 245 MB/s (random write). The test NVM is approximately 10× faster than the test SSD. We refrain from providing detailed information of the test NVM since it has not been publicly released.

We compare RangeKV with RocksDB-SSD, RocksDB-NVM, and NoveLSM. NoveLSM is based on LevelDB, whereas the other three are based on RocksDB. The detailed parameters of each system are as follows:

- RocksDB-SSD is based on the DRAM-SDD structure, where memtables are stored in the DRAM, and the log and SSTable files are stored in the SSD;
- RocksDB-NVM is based on the DRAM-NVM structure, where memtables are stored in the DRAM, and the log log and SSTable file are stored in the NVM;
- NoveLSM uses an 8 GB NVM as an NVM memtable (4 GB) and an NVM immutable memtable (4 GB), and all the SSTables are stored in the SSD;
- RangeKV uses an 8 GB NVM as the RangeTab structure to reorganize L0 and stores all SSTables in SSD.

To make a fair comparison, all the four databases take one thread for compaction and one thread for flushes. The size of memtables, immutable memtables, and SSTable is 64 MB, as the default configuration in RocksDB.

B. PERFORMANCE EVALUATION

We perform evaluations on the benchmark performance, compaction efficiency, and write amplification of the four KV stores.

1) BENCHMARK PERFORMANCE

We evaluate the read and write performances of the four KV stores using the db_bench released with RocksDB. The overall inserted data volume is 80 GB. The size of the keys is 16 bytes, and the size of the values vary from 1 to 64 KB.

Figures 11 and 12 show the sequential write and random write performances of the four KV stores. Comparing the test results, we can draw the following conclusions:

a: RANDOM WRITE

RangeKV can significantly improve the random write performance. From Figure 11, we can conclude that RocksDB-SSD has a performance of 8.7–10.4 MB/s in the value range of 1–64 KB. The random write of RangeKV is already 4.5× to 5.7× faster than that of RocksDB-SSD, which is NoveLSM 1.6× to 1.8×. This is because RangeTab and chunk structure make RangeKV require only appending the write operation when flushing immutables to the NVM, thus avoiding the process of writing the additional metadata when the data constitute SSTable files. The compaction strategy of RangeKV can shorten the key ranges, reduce the amount of data, and improve the compaction efficiency. In addition, because the RangeTab level can accommodate more data, RangeKV reduces the actual number of LSM-tree levels, the number of compactions, and the amount of data written to

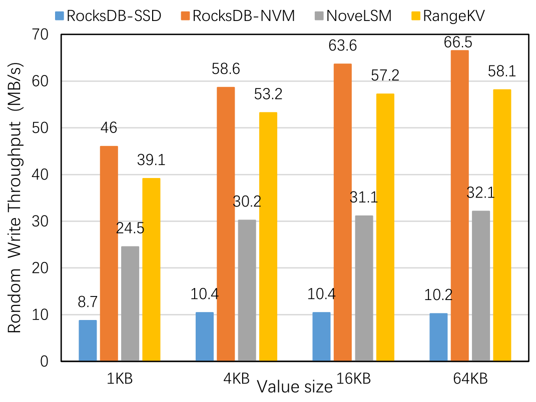


FIGURE 11. **Random write throughout.**

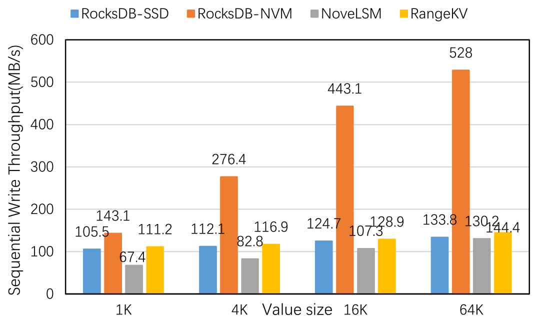


FIGURE 12. **Sequential write throughout.**

the storage medium, thereby improving the random write performance of the system.

b: SEQUENTIAL WRITE

The sequential write performance of RangeKV is better than those of RocksDB-SSD and NoveLSM, as shown in Figure 13.The sequential write operation of RocksDB-SSD, RocksDB-NVM, and NoveLSM does not trigger a compaction operation. The main factor determining their sequential write performance is the performance of the storage medium storing the memtables and SSTable files. The NoveLSM solution is based on the LevelDB KV store, which uses DRAM and large-capacity NVM alternate storage systems to write memtables, making its sequential write performance slightly lower than that of RocksDB-SSD, which only uses DRAM storage to write memtables. In the sequential write operation, RangeKV has two processes: writing DRAM data to NVM, and reading from NVM and writing to SSD. However, the two processes are asynchronous, and the capacity of the NVM is greater than that of the DRAM and significantly lower than that of the SSD. NVM plays the role of a temporary buffer for writing DRAM data into the SSD, so its sequential write performance is better than the process whereby DRAM data are directly written to the SSD. However, it is worse than the RocksDB-NVM scheme, which writes data directly from DRAM to NVM. Therefore, the sequential write performance of RangeKV is better than that of RocksDB-SSD, and the sequential write performance of RocksDB-SSD is better than that of NoveLSM.

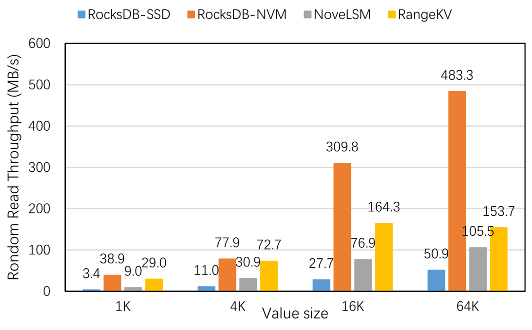


FIGURE 13. **Random read throughout.**

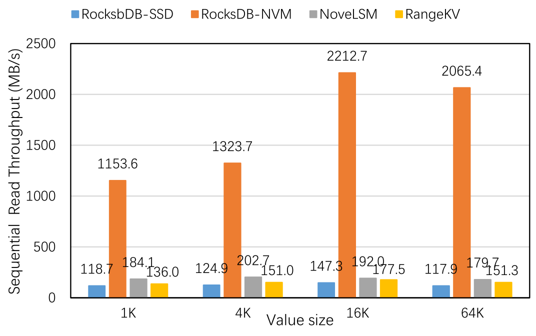


FIGURE 14. **Sequential read throughout.**

c: RANDOM AND SEQUENTIAL READ

Figures 13 and 14 show the sequential and random read performances of the four KV stores. Comparing the test results, we can draw the following conclusions:

(1) The random read and sequential read performances of RangeKV are significantly higher than those of RocksDB-SSD, but its sequential read performance is slightly lower than that of NoveLSM. The excellent read performance of RangeKV is mainly attributed to the design of its RangeTab structure. The division and mapping of RangeTab reduces the search ranges of the data, and the fewer LSM-tree levels help reduce the number of SSTable files in the SSD. The hash indexing accelerates the search efficiency, thus significantly reducing the number of times RangeTabs need to access the NVM. The advantage of the search efficiency makes up for the performance overhead of constructing the hash index; therefore, the random and sequential read performances are better than those of RocksDB-SSD.

NoveLSM uses two threads to read data from the memtables, immutables, and LSM-tree structures stored in the SSD to improve the read performance. However, when the amount of data is large enough, such as 80 GB, the parallel search efficiency is slightly low, and its random read performance is lower than that of RangeKV. For sequential read performance, the data may only be in the DRAM memtables (or in the NVM 8 GB memtables and immutables) or in an exact

SSTable in the SSD. At this time, the performance advantage of the dual-thread parallelization is maximized, and the test result is better than that of RangeKV.

(2) RocksDB-NVM has the highest random read and sequential read performances. This is because, although we have made some optimizations for the read operation performance of the RangeKV solution, the huge gap between the SSD and NVM performances cannot be completely offset by the RangeTab structural design.

From the above test results on the benchmark performance, it is concluded that although RangeKV has a lower sequential write performance than NoveLSM, it outperforms RockDB-SSD and NoveLSM in other aspects, indicating that it has a good overall performance advantage.

2) EVALUATION OF COMPACTION

The compaction operation has a significant influence on the write performance of KV stores. To quantify the compaction operation of different schemes, we record the detailed compaction processes in the four systems during the random writing process. Figure 15 shows the data volume of each compaction, and Figure 16 shows the average data volume and number of compactions.

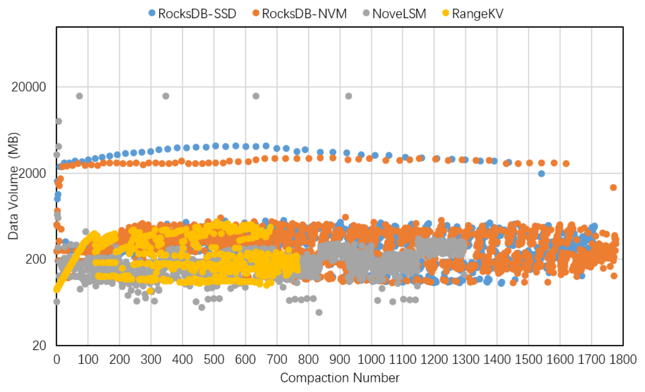


FIGURE 15. Compaction comparison.

From the test results, we can make four observations. First, RangeKV has the least compaction numbers among the four systems. The reduced compaction number attributes to fewer levels in RangeKV. Second, NoveLSM has fewer compaction numbers than RocksDB. This is because mutable NVM memtable serves more write requests in NVM and absorbs part of the update requests. Third, the average amount of RangeKV compaction data is the lowest. As shown in Figure 16, the average data volume of RangeKV is only 225.2 MB, whereas that of RocksDB is approximately 370 MB, a reduction of approximately 40%. This is mainly because the key ranges and compaction policy of RangeTab limit the compaction key ranges, which reduces the amount of data involved in the compaction.

3) EVALUATION OF WRITE AMPLIFICATION

Write amplification (WA) is another factor influencing the performance of KV stores. We record the valid data volume

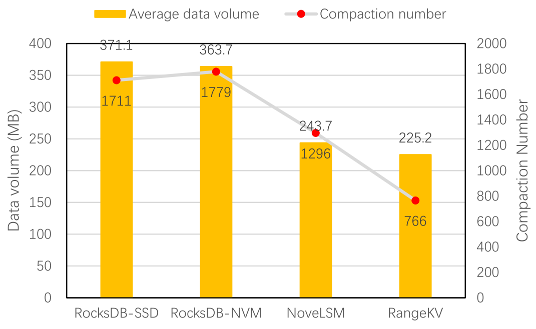


FIGURE 16. Average data volume in compaction.

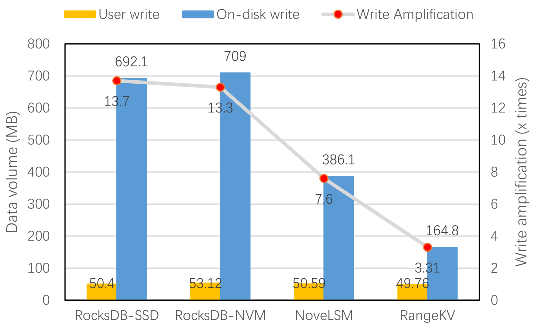


FIGURE 17. Write amplification.

from users and the overall disk-write volume for the four KV stores. We then calculate the WA by dividing the overall disk-write volume by the total volume of the user data written. Figure 17 shows the WA of the four systems. RocksDB-SSD and RocksDB-NVM have the same WA, which demonstrates that the efficiency of RocksDB-NVM merely comes from the fast-storage NVMs. The WA of RangeKV is only 3.3, which is the lowest among the four stores, 75% lower than RocksDB and 57% lower than NoveLSM. This is because RangeKV reduces the levels, and the reduced compaction data volume and compaction numbers, as shown in Figure 15, also contribute to mitigating on-disk writes.

C. PARAMETER SENSITIVITY EVALUATION

During the test, different parameter values of RangeKV may have had different effects on the test results, and the more significant one is the single RangeTab capacity. In this section, we study how different parameter values affect the system performance and further analyze the specific causes of the impact. We additionally test how the RangeKV reduction level affects the system performance, test the performance of the RangeKV system on all NVM storage media, and analyze the rationality and scope of application of RangeKV.

1) EVALUATION ON SINGLE RangeTab CAPACITY

The product of the single RangeTab capacity and the number of RangeTabs is a fixed value of 8 GB. We test RangeKV with varying single RangeTab capacity to explore how the single RangeTab capacity affects the random write performance of

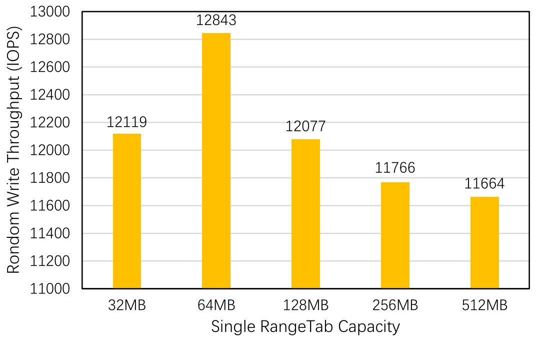


FIGURE 18. **Sensitivity study on the value of single RangeTab capacity.**

RangeKV. Figure18 shows the random write performance with single RangeTab capacities varying from 32 to 512 MB. The test results show that RangeKV exhibits the best random write performance when the capacity is 64 MB. This is because a higher single RangeTab results in larger compaction data volume and higher compaction latency. However, a smaller single RangeTab leads to more frequent compactions. They reach a balance at the threshold of 64 MB.

TABLE 1. **Data distribution on LSM-trees.**

(GB)	RocksDB L1=256MB	RocksDB L1=8GB	RangeKV L1=256MB	RangeKV L1=8GB
L0	0.12	0	0.3	0.4
L1	0.2	7.89	0.25	7.6
L2	2.45	44.06	2.19	41.76
L3	25.32	0	25	0
L4	22.66	0	19.83	0

2) EVALUATION OF REDUCING LEVELS

To evaluate the design choice of reducing levels, the size of L0 and L1 in RangeKV is set to 8 GB based on the 80 GB test database, which ensures a two-level structure on the SSDs (i.e., L1 and L2). In this section, we evaluate the impact of reducing LSM-tree levels by increasing the L1 size. RocksDB-SSD and RangeKV are evaluated with the original L1 size of 256 MB and an increased L1 size of 8 GB. Table 1 lists the data distribution on different levels for each of them after randomly loading 80 GB datasets. The test results demonstrate that both RocksDB and RangeKV reduce the level numbers when increasing the L1 size.

Figure 19 shows the performances of RocksDB and RangeKV with different L1 sizes. The test results show that increasing the L1 size has negative effects on RocksDB, whereas it significantly increases the performance of RangeKV. For RocksDB, the overhead of compacting L0 to L1 increases with the L1 size. However, the compaction overhead of RangeKV is independent of the level sizes owing to the fine granularity compaction. The contrasting performances of RocksDB and RangeKV confirm our analysis in Section 2.

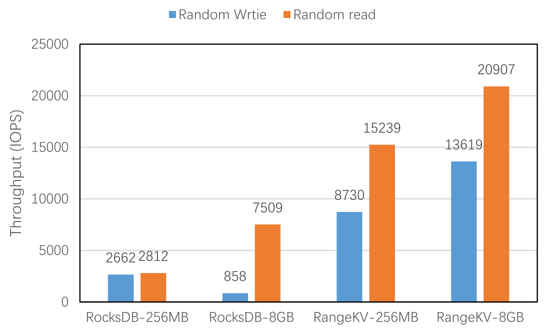


FIGURE 19. **Reducing levels.**

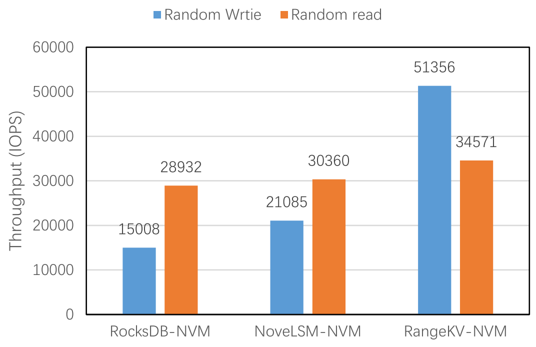


FIGURE 20. **System performance on DRAM-NVM.**

3) SYSTEM PERFORMANCE ON NVMs

We evaluate the availability of three KV stores on NVMs in the DRAMNVM hierarchy by randomly loading 80 GB dataset and seeking one million KV items. The three KV stores are (1) RocksDB-NVM: All SSTables are stored in NVM; (2) NoveLSM-NVM: Using 8 GB NVM to store NVM memtable and the remaining NVM space to store SSTables; (3) RangeKVNVM: Using 8 GB NVM to store matrix cache and the remaining NVM space to store SSTables. Figure 20 shows the test results. RangeKV achieves the best random write throughput among the three systems. The excellent performance of RangeKV demonstrates that RangeKV utilizes NVM in a better way than NoveLSM. NoveLSM outperforms RocksDB on the DRAM-NVM storage system. The results of NoveLSM are in accordance with the test results reported in [21].

VI. RELATED WORK

A. LSM-TREE-BASED KV STORES

Many previous works modified the LSM-tree design for improved compaction performance. PebblesDB [26] reduces write amplification by relaxing the restriction of keeping disjoint key ranges in each level and using guards to maintain partially sorted levels. bLSM [22] first uses bloom filters to accelerate lookup and proposes a new merge scheduler to avoid blocking writes for compaction. Lwc-tree [27] achieves lightweight compaction by merging and sorting only the metadata in SSTables. LSM-trie [28] reduces write amplification by maintaining data in a trie structure, which organizes KV pairs by hash-based buckets within each SSTable. VTtree [29] uses an additional layer of indirection to allow lightweight compaction overhead at the cost of fragmentation. TRIAD [30] reduces write amplification by leveraging the skewed data popularity and delayed compaction strategy. WiscKey [23] and HashKV [24] separate keys from values and simply merge the keys to reduce the compaction overhead.

B. OPTIMIZING LSM-TREES FOR NVM

Previous works redesigned LSM-trees to exploit the high performance of NVM. NoveLSM [21] is described in Section 2. We analyze its problem, and show that RangeKV has a better performance. NVMRocks [31] makes RocksDB NVM-aware using a persistent mutable memtable in the NVM. Unlike our use of NVM as a persistent memory, MyNVM [32] utilizes NVM block devices to reduce DRAM usage in SSD-based KV stores.

C. OPTIMIZING LSM-TREES FOR DIFFERENT STORAGE DEVICES

Many works have optimized LSM-trees for other storage devices such as SSDs, hard-disk drives, and HM-SMR drives. Currently, an increasing number of KV stores use SSDs as the persistent memory instead of hard-disk drives, such as WiscKey [23], SkimpyStash [2], FAWN [33], and HashKV [24]. GearDB [34] achieves both good performance and space efficiency by fully utilizing HM-SMR drives.

VII. CONCLUSION

In this study, we developed RangeKV, an LSM-tree-based persistent KV store built on a DRAM-NVM-SSD storage architecture. RangeKV exhibited a high read and write throughput by moving L0 from SSDs to NVMs using RangeTab, thus managing L0. By reducing the LSM-tree levels, optimizing the compactions with a double-buffer structure, and pre-constructing the hash index, RangeKV could reduce the write and read amplification. We implemented RangeKV based on RocksDB write real NVM devices. The evaluation results showed that RangeKV has a high and stable performance.

REFERENCES

[1] C. Lai, S. Jiang, L. Yang, S. Lin, G. Sun, Z. Hou, C. Cui, and J. Cong, "Atlas: Baidu's key-value storage system for cloud data," in *Proc. 31st Symp. Mass Storage Syst. Technol. (MSST)*, May 2015, pp. 1–14.

[2] B. Debnath, S. Sengupta, and J. Li, "SkimpyStash: RAM space skimpy key-value store on flash-based storage," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, 2011, pp. 25–36.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 1–26, Jun. 2008.

[4] (Jul. 2020). *RocksDB*. [Online]. Available: https://github.com/facebook/rocksdb

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proc. 21st ACM Symp. Operating Syst. Princ.*, Stevenson, WA, USA, Oct. 2007, pp. 205–220.

[6] (Jul. 2020). *LevelDB*. [Online]. Available: https://github.com/google/leveldb

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

[8] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah, "Serving large-scale batch computed data with project Voldemort," in *Proc. FAST*, vol. 12, 2012, p. 18.

[9] J. Zawodny, "Redis: Lightweight key/value store that goes the extra mile," *Linux Mag.*, vol. 79, no. 8, pp. 1–10, 2009.

[10] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, and H. Lee, "Scaling memcache at Facebook," in *Proc. USENIX Conf. Netw. Syst. Design Implement.*, 2013, pp. 385–398.

[11] L. Stanescu, M. Brezovan, and D. D. Burdescu, "Automatic mapping of MySQL databases to NoSQL MongoDB," in *Proc. Federated Conf. Comput. Sci. Inf. Syst.*, Sep. 2016, pp. 837–840.

[12] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Inf.*, vol. 33, no. 4, pp. 351–385, Jun. 1996.

[13] (Jul. 2020). *RocksDB*. [Online]. Available: http://cassandra.apache.org/

[14] A. S. Dent, *Getting Started With LevelDB*. 2013.

[15] M. N. Vora, "Hadoop-HBase for large-scale data," in *Proc. Int. Conf. Comput. Sci. Netw. Technol.*, vol. 1, Dec. 2011, pp. 601–605.

[16] J. Arulraj and A. Pavlo, "How to build a non-volatile memory database management system," in *Proc. ACM Int. Conf. Manage. Data (SIGMOD)*, 2017, pp. 1753–1758.

[17] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, and C. H. Lam, "Phase-change random access memory: A scalable technology," *IBM J. Res. Develop.*, vol. 52, nos. 4–5, pp. 465–479, 2008.

[18] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature*, vol. 453, no. 7191, pp. 80–83, May 2008.

[19] A. Driskill-Smith, "Latest advances and future prospects of STT-RAM," in *Proc. Non-Volatile Memories Workshop*, 2010, pp. 11–13.

[20] J. Yang, Q. Wei, C. Chen, C. Wang, B. He, and K. L. Yong, "NV-tree: Reducing consistency cost for NVM-based single level systems," in *Proc. 13th USENIX Conf. File Storage Technol. (FAST)*, 2015, pp. 167–181.

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

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

[23] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "WiscKey: Separating keys from values in SSD-conscious storage," *ACM Trans. Storage*, vol. 13, no. 1, pp. 1–28, Mar. 2017.

[24] H. H. Chan, C. J. Liang, Y. Li, W. He, P. P. Lee, L. Zhu, Y. Dong, Y. Xu, Y. Xu, J. Jiang, R. Arpaci-Dusseau, "Hashkv: Enabling efficient updates in *KV* storage via hashing," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2018, pp. 1007–1019.

[25] S. Scargall, "Introducing the persistent memory development kit," in *Programming Persistent Memory*. Springer, 2020, pp. 63–72.

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

[27] T. Yao, J. Wan, P. Huang, X. He, Q. Gui, F. Wu, and C. Xie, "A light-weight compaction tree to reduce I/O amplification toward efficient key-value stores," in *Proc. 33rd Int. Conf. Massive Storage Syst. Technol. (MSST)*, 2017, pp. 1–13.

[28] X. Wu, Y. Xu, Z. Shao, and S. Jiang, "LSM-trie: An LSM-tree-based ultra-large key-value store for small data items," in *Proc. USENIX Annu. Tech. Conf. (USENIXATC)*, 2015, pp. 71–82.

[29] P. Shetty, R. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok, "Building workload-independent storage with VT-trees," in *Proc. USENIX Conf. File Storage Technol.*, 2013, pp. 17–30.

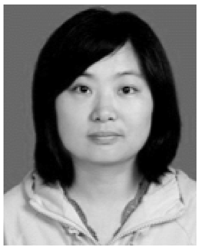
[30] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka, "TRIAD: Creating synergies between memory, disk and log in log structured key-value stores," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2017, pp. 363–375.

[31] A. Eisenman, D. Gardner, I. AbdelRahman, J. Axboe, S. Dong, K. Hazelwood, C. Petersen, A. Cidon, and S. Katti, "Reducing DRAM footprint with NVM in Facebook," in *Proc. 13th EuroSys Conf.*, Apr. 2018, pp. 1–13.

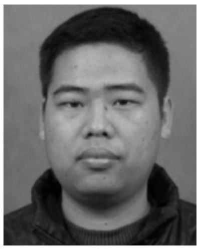
[32] J. Li, A. Pavlo, and S. Dong, "NVMRocks: RocksDB on non-volatile memory systems," Tech. Rep., 2017.

[33] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, and V. Vasudevan, "FAWN: A fast array of wimpy nodes," in *Proc. ACM Symp. Operating Syst. Princ.*, 2009, pp. 1–14.

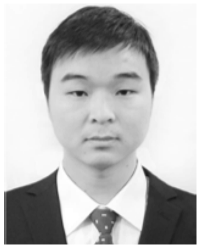
[34] T. Yao, J. Wan, P. Huang, Y. Zhang, Z. Liu, C. Xie, and X. He, "GearDB: A GC-free key-value store on HM-SMR drives with gear compaction," in *Proc. 17th USENIX Conf. File Storage Technologies (FAST)*, 2019, pp. 159–171.



LING ZHAN received the bachelor's degree in computer application from Henan University, China, in 1996, and the M.S. and Ph.D. degrees in computer science from the Huazhong University of Science and Technology (HUST), China, in 2005 and 2010, respectively. Her research interests include computer architecture and key-value systems.



KAI LU received the bachelor's degree in computer science from the Huazhong University of Science and Technology (HUST), China, in 2018, where he is currently pursuing the Ph.D. degree with the Computer Science Department. His research interests include computer architecture and key-value systems.



ZHILONG CHENG received the bachelor's and M.S. degrees in computer science from the Huazhong University of Science and Technology (HUST), China, in 2016 and 2019, respectively. His research interests include computer architecture and key-value systems.

JIGUANG WAN received the bachelor's degree in computer science from Zhengzhou University, China, in 1996, and the M.S. and Ph.D. degrees in computer science from the Huazhong University of Science and Technology (HUST), China, in 2003 and 2007, respectively. His research interests include computer architecture and key-value systems.