An LSM Tree Augmented with B$^+$ Tree on Nonvolatile Memory

DONGUK KIM and JONGSUNG LEE, Seoul National University and Samsung Electronics, Korea
KEUN SOO LIM, JUN HEO, TAE JUN HAM, and JAE W. LEE, Seoul National University, Korea

Modern log-structured merge (LSM) tree-based key-value stores are widely used to process update-heavy workloads effectively as the LSM tree sequentializes write requests to a storage device to maximize storage performance. However, this append-only approach leaves many outdated copies of frequently updated key-value pairs, which need to be routinely cleaned up through the operation called *compaction*. When the system load is modest, compaction happens in background. However, at a high system load, it can quickly become the major performance bottleneck. To address this compaction bottleneck and further improve the write throughput of LSM tree-based key-value stores, we propose LAB-DB, which augments the existing LSM tree with a pair of B$^+$ trees on byte-addressable nonvolatile memory (NVM). The auxiliary B$^+$ trees on NVM reduce both compaction frequency and compaction time, hence leading to lower compaction overhead for writes and fewer storage accesses for reads. According to our evaluation of LAB-DB on RocksDB with YCSB benchmarks, LAB-DB achieves 94% and 67% speedups on two write-intensive workloads (Workload A and F), and also a 43% geomean speedup on read-intensive YCSB Workload B, C, D, and E. This performance gain comes with a low cost of NVM whose size is just 0.6% of the entire dataset to demonstrate the scalability of LAB-DB with an ever increasing volume of future datasets.

CCS Concepts: • **Information systems** → **Data management systems**; **Storage class memory;**

Additional Key Words and Phrases: Nonvolatile memory, key-value store, LSM tree, compaction

ACM Reference format:
Donguk Kim, Jongsung Lee, Keun Soo Lim, Jun Heo, Tae Jun Ham, and Jae W. Lee. 2024. An LSM Tree Augmented with B$^+$ Tree on Nonvolatile Memory. *ACM Trans. Storage* 20, 1, Article 4 (January 2024), 24 pages.
https://doi.org/10.1145/3633475

1 INTRODUCTION

Key-value stores [1, 22–24, 35, 39, 42, 52] are widely used for many large-scale applications on cloud such as messaging [7, 22], online shopping [20], search indexing [10], and advertising [10, 22]. Unlike the traditional RDBMS, which offers more functionalities at the expense of

We thank the anonymous reviewers for their valuable comments. We also thank Jung Ho Ahn from Seoul National University for providing a server with Intel Optane DCPMM for our evaluation. This work was supported in part by an Institute of Information & Communications Technology Planning & Evaluation (IITP) grant (2021-0-00853, Developing Software Platform for PIM Programming) funded by the Korea Government (MSIT)..
Authors' addresses: D. Kim and J. Lee, Seoul National University and Samsung Electronics, 1, Gwanak-ro, Gwanak-gu, Seoul 08826, Korea; e-mails: {dongukim12, leitia}@snu.ac.kr; K. S. Lim, J. Heo, T. J. Ham, and J. W. Lee (Corresponding author), Seoul National University, 1, Gwanak-ro, Gwanak-gu, Seoul 08826, Korea; e-mails: {me9907, j.heo, taejunham, jaewlee}@snu.ac.kr.

1553-3077/2024/01-ART4
https://doi.org/10.1145/3633475

complexity, key-value stores provide a scalable way to handle a large amount of data using a very simple interface. Among various key-value stores, the most popular type is persistent key-value stores based on LSM trees [1, 2, 10, 22, 24]. Such key-value stores are designed to sustain high write throughput as the LSM tree always accesses a storage device sequentially. However, this append-only approach of LSM tree requires additional background operations called *compaction*, which frees the old, outdated key-value pairs if there exists a more up-to-date copy for the same key. When there are many incoming writes, frequent compaction operations often become the main performance bottleneck of the key-value store.

Unfortunately, it is challenging to eliminate this compaction bottleneck. One possible way to address this challenge is to perform as many in-place updates as possible in DRAM to filter multiple updates on the same key. However, scaling DRAM size is not a cost-effective solution. Another way is to relax the append-only characteristics of the LSM tree to a certain degree, but it inevitably results in an increase of random writes on the storage device, eventually degrading the write throughput.

The recent introduction of byte-addressable **nonvolatile memory** (**NVM**) devices [29, 30] provides new opportunities to address this challenging problem. On one hand, NVM is close to the storage device in that it provides persistency. On the other hand, it also has similar characteristics to DRAM in that it provides a byte-addressable interface and has a low access latency more comparable to DRAM than a storage device like NAND flash. While there are existing proposals to leverage NVM in key-value stores, they reduce either only compaction frequency [33] or only compaction time [55], but not both. While SLM-DB [32] takes a more holistic approach to address the compaction overhead, its scalability to larger datasets is limited as the B^+ tree index for the entire dataset must be maintained in NVM and garbage collection must be performed for all leaf nodes.

Thus, we propose LAB-DB, a novel LSM tree-based key-value store augmented with B^+ tree on byte-addressable NVM, addressing both compaction frequency and compaction time in a cost-effective way. By introducing the auxiliary B^+ tree, LAB-DB can substantially reduce the cost of compaction operations for writes as well as the frequency of storage accesses for reads. We implement LAB-DB by extending RocksDB and evaluate it on both micro-benchmarks released by RocksDB and real-world YCSB macro-benchmarks to demonstrate significant speedups for both write- and read-intensive workloads. This performance gain comes with a low cost of NVM whose size is just a tiny fraction of the total dataset size.

In summary, we make the following contributions:

- We carefully analyze and compare the design of recently proposed NVM-augmented key-value stores [21, 32, 33, 55] to identify opportunities for improving the space efficiency of NVM.
- We design and implement LAB-DB, which augments an LSM tree with a pair of B^+ trees on NVM to address the compaction bottleneck. LAB-DB substantially reduces the compaction overhead by effectively utilizing the B^+ trees on NVM for in-place updates and fine-grained compaction, thus managing a large number of key-value pairs in a space-efficient manner.
- We demonstrate the effectiveness of LAB-DB by evaluating it on a system with real NVM devices (Intel Optane DC Persistent Memory Modules [29]). Compared to the vanilla RocksDB running YCSB workloads, LAB-DB achieves 94% and 67% higher throughput on two write-intensive workloads (Workload A and F) and a 43% geomean speedup on four read-intensive workloads (Workload B, C, D, and E).

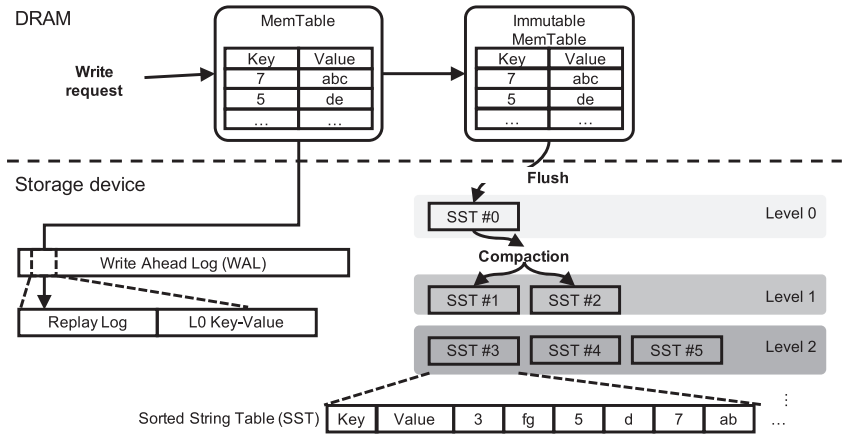

Fig. 1. RocksDB overview.

2 BACKGROUND AND MOTIVATION

2.1 LSM Tree-Based Key-Value Store

Overview. Log-Structured Merge (LSM) tree is a popular data structure for key-value stores requiring high write throughput. Many modern key-value stores such as Cassandra [1], HBase [2], BigTable [10], LevelDB [24], and RocksDB [22] utilize this data structure. Figure 1 shows the basic structure of RocksDB [22], a popular LSM tree-based key-value store. As shown in the figure, the LSM tree-based key-value store maintains multiple data structures. Specifically, it maintains a set of data structures (i.e., MemTable and Immutable MemTable) in DRAM and the rest of data structures (i.e., **Write-Ahead Log (WAL)** and a set of Sorted String Table (SSTable) files) in a storage device. Here, SSTables are structured in a multi-level tree (i.e., LSM tree), where the top-level (called L_0) has the fewest SSTable files, and the following levels (L_1, L_2, L_3...) have an increasingly larger number of SSTable files. Below, we briefly introduce the basic operations of the LSM tree-based key-value store.

Write (Put) Operation. When a user inserts a key-value pair, it is first logged into a storage-resident WAL. Then, the system checks whether the same key is already resident in the in-memory data structure MemTable. If so, the value for the key is overwritten in place. If not, the key-value pair is newly added to the MemTable. With repeated write operations with different keys, the MemTable will eventually fill up. In such a case, the MemTable is immediately declared immutable, and a new, empty MemTable will store any upcoming key-value pairs.

Flush Operation. A *flush* is a background operation that converts an immutable MemTable in memory to a single SSTable file in storage. Specifically, this newly created SSTable is stored at the very first level of the LSM tree. This operation is scheduled when an immutable MemTable exists, or if the pre-allocated space for WAL becomes full. Once a flush happens, the data become persistent, and thus WALs for those updates recorded in the flushed MemTable are cleared.

Compaction Operation. Figure 2 shows the RocksDB compaction flow. A flush operation adds an SSTable file to the top level (L_0) of the LSM tree. If the top-level of the LSM tree was already full, the L_0 compaction operation is triggered. During this operation, all existing SSTable files in L_0 are first read and merged. If there are multiple SSTable files having the same keys, only the most recent one is kept, and the rest are discarded. Then, the merged SSTable file is again merged with L_1 SSTable files whose key range overlaps with the merged L_0 SSTable file. Then, this merged SSTable is divided into several files of a similar size and then stored in L_1 of the LSM tree. The repeated

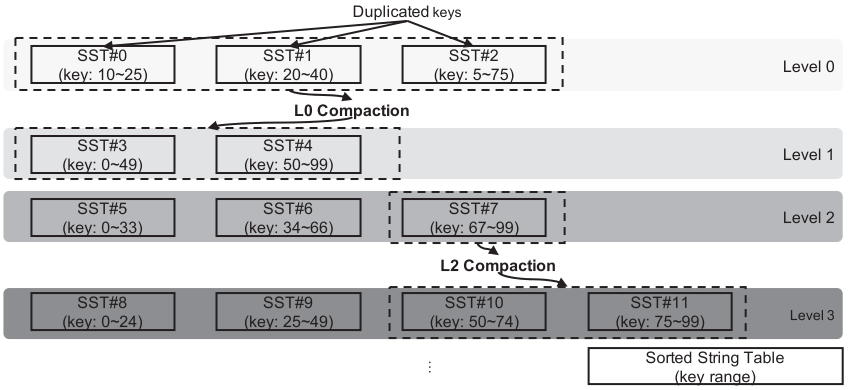

Fig. 2. RocksDB compaction flow.

compaction will eventually fill up the L_1 of the LSM tree, which triggers the L_1 compaction. One of the SSTable files in L_1 is selected, and then overlapping SSTable files from L_2 are merged with the selected L_1 file and then stored back in L_2. Compaction at a lower level is similar to the L_1 compaction.

Read (Get) Operation. Read operation first searches for the key in MemTable. If found, the value is retrieved from memory. If not, the immutable MemTable is searched. If the value is not in the immutable MemTable, the value should be searched in L_0 of the LSM tree, from the most-recently stored SSTable file to the oldest SSTable file. To speed up this process, a Bloom filter is utilized to check if an SSTable file can possibly contain the target key or not. If a filter indicates that an SSTable file may contain the target key, the SSTable index file is inspected to (i) check the existence of the target key in this file, and (ii) find out the exact location of the value for the target key. This information is then utilized to retrieve the final value. If the value is not found in any of the SSTable files in L_0, the L_1 SSTable file covering the target key range is inspected. Similarly, if it is not in the corresponding L_1 file, the SSTable files in the following levels are inspected.

2.2 Compaction Bottleneck in LSM Trees

Compaction Bottleneck. The primary bottleneck of the LSM tree-based key-value store is compaction operations. Ideally, those operations should happen in background, and its latency should be completely hidden. However, when keys are added to the key-value store at a very high rate, the compaction threads cannot secure some free space at upper levels of the LSM tree with a sufficiently high rate, and eventually need to stall the write operations. Figure 3 shows the throughput of RocksDB over time when random key-value pairs (8 B key, 1024 B value) are inserted repeatedly on a 90 GB dataset. As shown in the figure, the system's peak throughput is around 40 K operations/s. However, it often fails to maintain this level of throughput, which often goes down very low (e.g., 1.6 K operations/s). As a result, the average throughput (i.e., 5 K operations/s) is well below its peak throughput. The sudden degradation of the throughput during the execution is the result of the compaction bottleneck. Blue and grey horizontal lines at the top of the figure represent the L_0 and L_1 compaction activities, respectively. As expected, throughput degradation often is highly correlated with compaction activities in time. The compaction threads fail to secure free space for the upper levels of the LSM tree, which blocks flush operations. Thus, MemTable remains full, and no more write can happen until the compaction thread finishes its operations and secures some free space in the LSM tree.

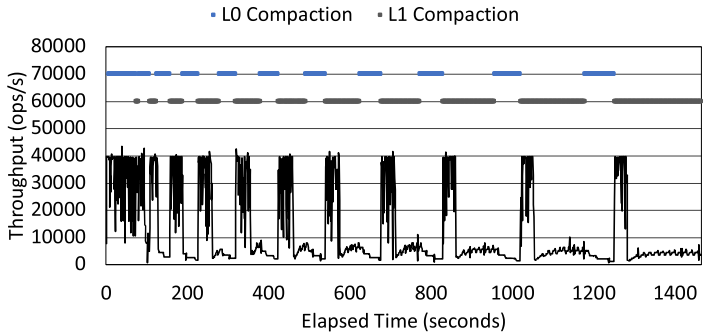

Fig. 3. RocksDB throughput over time for random writes.

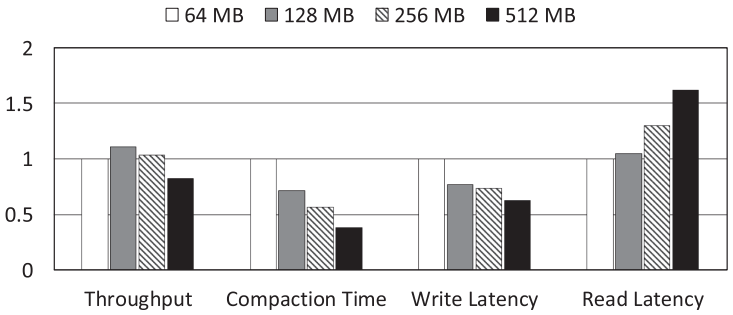

Fig. 4. Throughput and total compaction time of RocksDB running YCSB Workload A with varying MemTable size. The total dataset size is 90 GB and the growth factor between adjacent layers is set to 10.

Challenges for Addressing Compaction Bottleneck. One way to mitigate the compaction bottleneck is to minimize the amount of data that needs to be compacted. This can be achieved by increasing the size of the MemTable, as shown in Figure 4. We use the YCSB Workload A with a Zipfian constant of 0.99 for the key distribution. To configure the size of each level of the LSM tree, we use a growth factor of 10, which is the default setting of RocksDB. Our experiments demonstrate that increasing the MemTable size from 64 MB to 512 MB results in an increase in the rate of in-place updates for the existing key from 34% to 44%. This eliminates duplicate keys in SSTable files, which would otherwise be flushed separately to the storage device. More duplicate key updates reduce the number of keys that propagate to the storage device for writes, and thus the number of keys that need to be compacted is getting closer to that of unique keys. Thus, the time spent on compaction keeps decreasing as the MemTable size increases.

However, this approach has several significant issues. First, it does not necessarily lead to an increase in performance as a larger MemTable results in an increase in MemTable search time to slow down reads. Furthermore, it is not cost-effective to increase the MemTable size since the capacity of DRAM is not very scalable. Moreover, if the value size is much larger than the key size, scaling a MemTable would lead to much smaller performance gains. For example, if the value size were 1 MB instead of 1 KB, the effective MemTable size (i.e., the number of keys that are buffered in the MemTable) would decrease by a factor of 1,000×. Thus, scaling MemTable may be a one-time solution that can mildly alleviate the compaction bottleneck, but does not eliminate its root cause.

Leveraging NVM for Compaction Bottleneck. NVM provides unique opportunities for addressing the compaction bottleneck. It is known to have a 2x–4x higher density than the

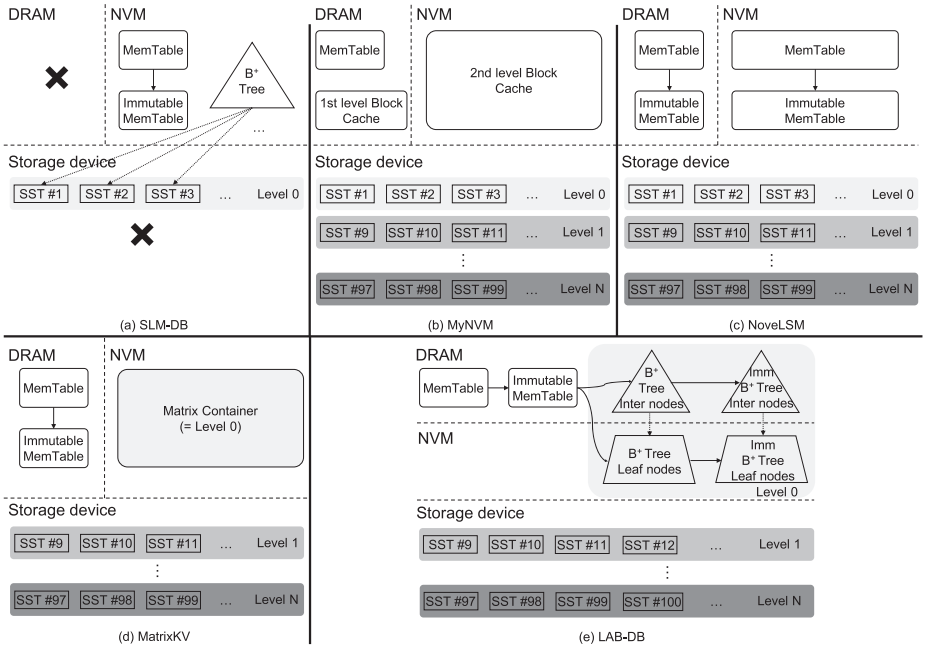

Fig. 5. Comparison of existing NVM-augmented LSM trees (a)–(d) and our work (e).

conventional DRAM [29] and provides persistency. Figure 5(a) through (d) compare the prior art of NVM-augmented key-value stores based on LSM trees, which attempt to mitigate the compaction bottleneck. SLM-DB [32] (Figure 5(a)) maintains a B^+ tree index on NVM, which tracks the location of values in the storage device for all keys in the dataset. Indeed, this effectively obviates the need for compaction, but this is not very scalable since even the dense NVM memory may not be large enough to house *all* keys in the dataset. On the other hand, MyNVM [21] (Figure 5(b)) utilizes NVM as a giant block cache for the storage device. This can almost transparently reduce the access to the storage devices, and thus can partially alleviate the compaction bottleneck; however, this does not really reduce the number of compaction operations. Rather, it simply handles some of the compaction operations at the NVM cache level. NoveLSM [33] (Figure 5(c)) utilizes NVM as an extra space to put a MemTable and an immutable MemTable. This is effectively increasing the size of the MemTable, albeit with NVM rather than DRAM. Finally, MatrixKV [55] (Figure 5(d)) utilizes a unique data structure called MatrixContainer on NVM to replace the L_0 level of the LSM tree. This structure makes the compaction process faster; however, it does not reduce the number of compactions or the size of the data that need to be compacted, and performance degradation due to other levels of compaction cannot be prevented. Note that all of these proposals require the NVM space of a significant fraction (at least 10%) of the dataset size to realize their potential benefits. This space requirement can potentially become a scalability bottleneck for much larger datasets of the future.

Our Work. For various reasons the prior work fails to provide an effective solution to mitigate the compaction bottleneck. Previous studies have not adequately addressed the issue of L_0 accumulating all duplicate writes to the same key, which is the root cause of compaction overhead. This characteristic of L_0 leads to a rapid increase in L_0 size, which in turn causes a significant increase in storage traffic due to L_0 compaction and subsequent compaction at lower levels. Simply utilizing NVM as an extension to DRAM, or a cache or faster storage does not really

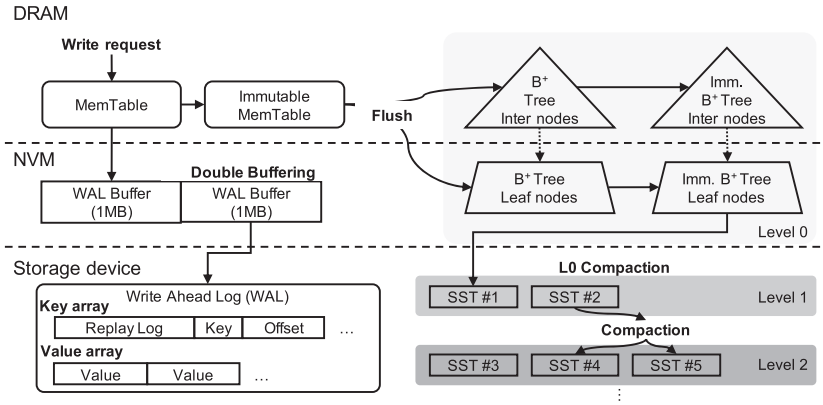

Fig. 6. LAB-DB overview.

address the root cause of the compaction bottleneck. Our premise is that *we should reduce the total volume of data to compact through the efficient use of NVM*. To achieve this goal, we propose LAB-DB (Figure 5(e)), which augments the L_0 of an LSM tree with a B$^+$ tree placed in NVM. Our design minimizes L_0 compaction triggers, which in turn decreases the frequency of compaction at lower levels by providing more opportunities for flushed key-value pairs to be updated in place within the B$^+$ tree. Additionally, to provide a cost-effective design, we have included key-value separation and log management functions.

3 LAB-DB DESIGN

3.1 Overview

Figure 6 shows the overall architecture of LAB-DB, which builds on RocksDB. The key architecture change is that LAB-DB replaces L_0 of the LSM tree with a B$^+$ tree and then places it on NVM instead of the storage device. Since L_0 is at the highest level, reducing the compaction frequency can significantly reduce not only the L_0 compaction overhead but also the cascading lower-level compaction overhead from L_0 compaction.

LSM Tree Augmented with B$^+$ Tree (LAB). A B$^+$ tree is fundamentally different from SSTable files in the LSM tree in that it supports in-place updates. Rather than simply appending new put requests to the SSTable files, and then later merging them, B$^+$ tree instead immediately checks if the tree already has an entry for the target key. If so, its value is updated in place. If not, a node for the new key-value pair is inserted into the tree. The primary advantage of B$^+$ tree is that it maintains a concise representation for the current set of key-value pairs. On the other hand, SSTable files potentially contain multiple value instances for the same key, and thus those files need to be merged later. The in-place update of B$^+$ tree greatly reduces the compaction overhead for two reasons. First, the in-place update utilizes the space more efficiently, and L_0 compaction is invoked less frequently. Second, even if compaction happens, its overhead is greatly reduced. LAB-DB does not need to merge multiple SSTable files in L_0 since B$^+$ tree only stores a single value for each key.

Despite these advantages, replacing the L_0 of the LSM tree with a B$^+$ tree does not work well on a conventional system. This is because both read and write to a B$^+$ tree generate random data accesses, which can result in significant performance degradation on block storage devices. To avoid this potential performance degradation, LAB-DB places the B$^+$ tree on byte-addressable NVM. By doing so, it is possible to exploit all the advantages of B$^+$ tree-based L_0, while avoiding the performance degradation that the conventional block storage would suffer.

3.2 LAB-DB Operations

Write (Put) Operation. First, as in the conventional RocksDB, a key-value pair is recorded in the WAL. Originally, this log is directly written to the storage device to guarantee persistency. In LAB-DB, this is instead first stored in 2 MB WAL buffer (double-buffered) located in the NVM. Then, once the buffer becomes full, the WAL in the buffer is flushed to the storage device as usual. This design choice can substantially reduce the write latency since persistency is guaranteed as soon as the data is written on the NVM device. It also improves the SSD bandwidth utilization as our design issues more coarse-grained write requests (e.g., 1 MB) to the storage device. Storage devices that use a block-interface, such as HDD or SSD, are more efficient at handling large I/O requests than smaller ones. RocksDB's group logging mechanism attempts to consolidate small I/O requests for logging purposes, but it is not always successful, resulting in some small write operations being logged individually. This can lower the utilization of I/O bandwidth. To address this, we have implemented a method that utilizes a small amount of NVM (e.g., 2 MB) to eliminate small-sized logging.

Moreover, LAB-DB optimizes the WAL structure. Specifically, the original WAL stores the replay log and key-value pairs in the contiguous region. Instead, we separate the region for values (*Value Array* in Figure 6) from the one for the replay log and key (*Key Array* in Figure 6). After write-ahead logging, the key-value pair needs to be written to the MemTable. Here, LAB-DB extends the MemTable and also records the logical address of the value in the storage device. This design choice is made to efficiently utilize values in the storage device as data records for the B^+ tree (more details in the following paragraph). When the MemTable is full, it becomes an immutable one, and a flush operation is scheduled.

Flush Operation. Originally, a flush operation is a simple operation that copies the list of key-value pairs in an Immutable MemTable to the storage device and stores it as an SSTable file. With LAB-DB, the top-level (L_0) of the LSM tree is replaced with a B^+ tree resident on NVM, and thus the flush operation now needs to update a B^+ tree by taking a list of key-value pairs passed from the MemTable. The B^+ tree of LAB-DB consists of internal nodes and leaf nodes. LAB-DB proposes a cost-effective structure by utilizing the property of the B^+ tree that allows the entire database to be recovered if only the data of leaf nodes is recoverable. For this reason, internal nodes are stored on DRAM, and leaf nodes are stored on NVM. An internal node points to another internal node or a leaf node, and each leaf node has a key and a pointer to the corresponding value in the value array resident in the storage device. During a flush operation, when a key that already exists in the B^+ tree is inserted, an in-place update operation is performed. The old value offset in the B^+ tree is updated to the newly inserted value offset.

Figure 7 illustrates the flush operation in LAB-DB step by step. Before the update of B^+ tree, all data in the WAL Buffer is first flushed to the storage device ❶. Then, a list of key-value pairs are iteratively inserted (potentially in parallel) into the B^+ tree ❷. During the insertion if there already exists a node with the same key, its pointer to the value is updated ❸. Once the B^+ tree update completes, the key array portion of the WAL is no longer necessary and thus cleared ❹. Note that LAB-DB only utilizes NVM to store the nodes of the B^+ tree, not the data records (values) themselves. This effectively allows the B^+ tree to track a larger number of key-value pairs in a space-efficient manner. To guarantee the correctness of the scan semantic, which requires to read the values at a specific point in time, even with in-place L_0 updates, LAB-DB schedules flushes in a way that scan and flush operations are mutually exclusive. This mutex incurs negligible performance overhead because a flush operation consumes very little time by leveraging fast NVM compared to much slower scan operations, as demonstrated by our evaluation with a scan workload (YCSB-E) in Section 4.2.

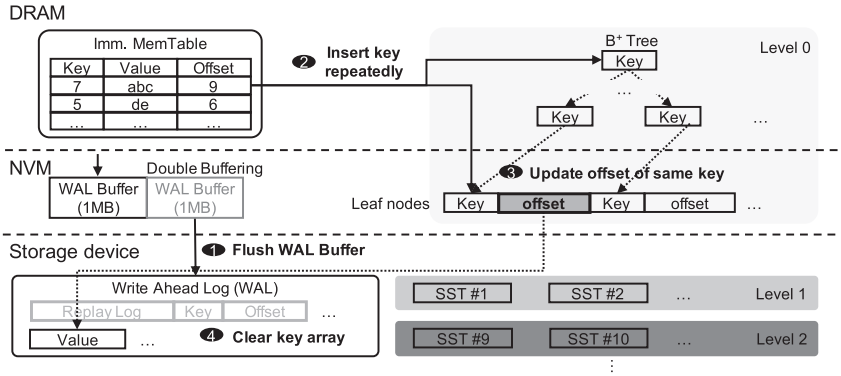

Fig. 7. Illustration of flush operation in LAB-DB.

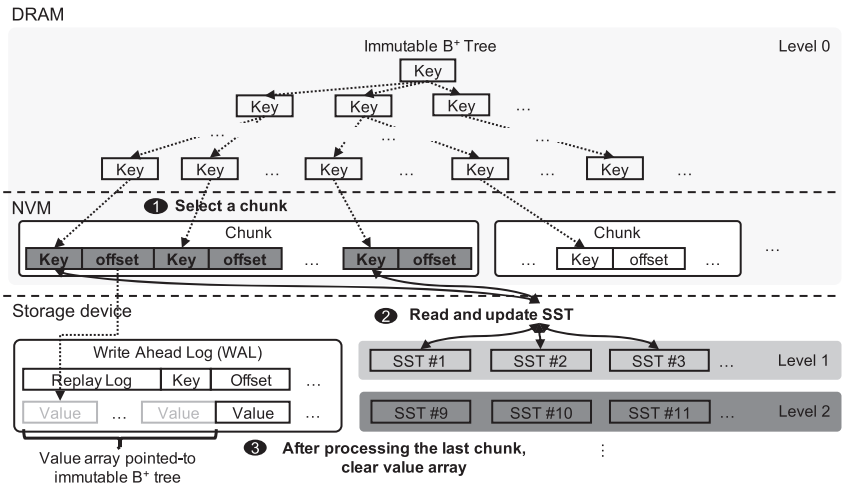

Fig. 8. Illustration of compaction operation in LAB-DB.

Compaction Operation. The above flush operation updates the B$^+$ tree by inserting key-value pairs passed from the MemTable. This leads to an increase in B$^+$ tree size. Once the total space utilized for B$^+$ tree leaf nodes exceeds a specific threshold (e.g., 75% of the pre-allocated NVM size for L_0), the L_0 compaction happens. Specifically, if there is no existing immutable B$^+$ tree, the current B$^+$ tree is declared immutable, and the following flush operation will construct a new mutable B$^+$ tree. Then, Figure 8 illustrates the rest of the steps for a compaction operation in LAB-DB. The compaction operation for a portion of immutable B$^+$ tree is scheduled ❶. To select this portion, the leaf nodes of the immutable B$^+$ tree is logically divided into multiple similar-sized chunks in a way that each chunk is responsible for a contiguous range of keys. Then, a chunk that is responsible for the leftmost range from the remaining chunks is scheduled for compaction. During this compaction, values for the target chunk are read from the WAL stored in a storage device and they are used to update the SSTable files in L_1 whose range overlaps with this chunk ❷. Then, the leaf nodes responsible for the chunks are removed from NVM, and relevant internal nodes are updated or deleted as well. Finally, if a particular L_0 compaction processes the last chunk for the immutable B$^+$ tree, the whole value array region responsible for the immutable B$^+$ tree is cleared ❸.

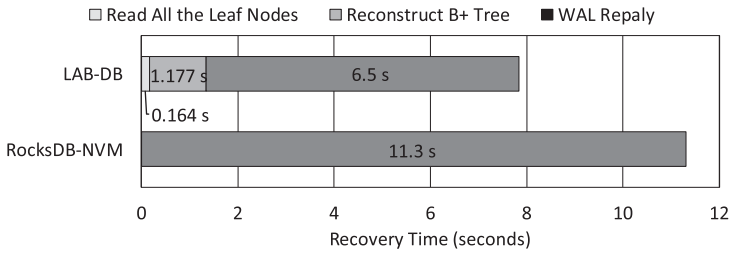

Fig. 9. Breakdown of time spent for recovery.

Note that LAB-DB does not perform compaction on the whole B^+ tree at once. Rather, LAB-DB makes it immutable and incrementally compacts a portion of it with L_1. This approach has two advantages. First, it helps maintaining a high occupancy of the L_0, so that the L_0 can effectively be utilized as a cache for read operations that miss at the MemTable. Second, the finer-grained compaction leads to a tighter tail latency. LAB-DB does not affect the L_1 or lower-level compaction mechanisms.

Read Operation. Overall, the LAB-DB read operation is similar to that of the RocksDB baseline. It first checks the MemTable, and if it does not find the key in the MemTable, it searches the immutable MemTable. If it misses again, then it inspects L_0 of the LSM tree, which is now managed with B^+ tree structures. First, it checks the mutable B^+ tree. Starting from the root and internal nodes stored in DRAM, it traverses the tree and reaches the leaf node if the tree has a matching key. Then, it utilizes the pointer at the leaf node to retrieve the value in a storage device. If it does not find the key in a mutable B^+ tree, it searches for the immutable B^+ tree and retrieves the value in a similar manner. If the key is not found in L_0, it continues its search on lower levels as in the conventional LSM tree.

Recovery. The recovery process for a failure is mostly similar to RocksDB, but LAB-DB recovery requires slightly more work. Specifically, it needs to reconstruct the internal nodes for B^+ tree that was stored in DRAM. The recovery process of LAB-DB consists of two additional steps: (i) reading the key and offset information of the leaf node; (ii) replaying that information to reconstruct the B^+ tree. Once these additional steps are completed, data recovery will be performed through the WAL replay operation, which is the recovery process of RocksDB. Figure 9 shows the recovery time breakdown for LAB-DB using a 512 MB B^+ tree and RocksDB-NVM, which is RocksDB with an 8 GB NVM L_0, when using a 90 GB dataset. LAB-DB spends 0.164 seconds reading all leaf nodes and 1.177 seconds reconstructing the B^+ tree, resulting in a total of 1.341 seconds for B^+ tree recovery. Notably, the reconstruction process does not require much time because the B^+ tree is composed solely of DRAM and NVM, so there is no need to access storage devices. The WAL replay time for LAB-DB and RocksDB-NVM is 6.5 seconds and 11.3 seconds, respectively. The reason for the shorter WAL replay time of LAB-DB is that it can process requests faster than RocksDB-NVM through B^+ tree. The total recovery time of LAB-DB is 3.5 seconds shorter than the baseline RocksDB-NVM, indicating that LAB-DB has no negative impact on the recovery time.

3.3 Log File Management

LAB-DB efficiently manages the L_0 by applying the in-place update method of the B^+ tree. However, as an in-place update occurs, the log size continues to grow until compaction happens. To address this, we propose two techniques to reduce log size.

First, we introduce two methods of constructing chunks, one of which is selected adaptively according to the phase of compaction. The first method scans consecutive leaf nodes (*leaf scan*)

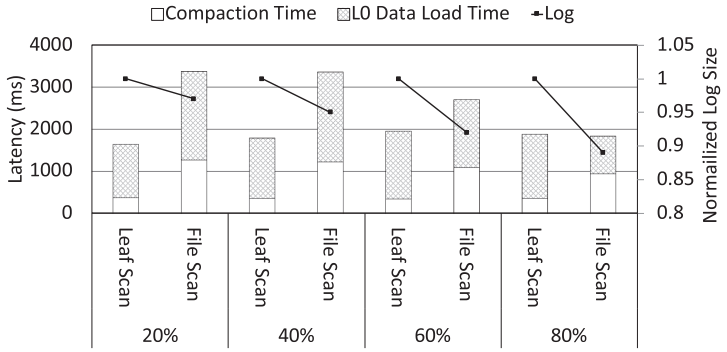

Fig. 10. L_0 compaction time and normalized log size for two methods of constructing chunks when performing YCSB Workload A on a 90 GB dataset.

mentioned in the previous section, and the second method collects the leaf nodes with the key belonging to the oldest log file (*file scan*) to remove the log file as quickly as possible. Figure 10 shows the latency of L_0 compaction and the change in the log size for the two methods with the varying degree of compaction progress. As shown in the figure, we observe that, as the compaction ratio increases, the file scan method shows similar latency to the leaf scan method, but decreases log size by 11%. As the compaction process makes a progress, both the key range and the number of leaf nodes that need to be read through the file scan are reduced, thereby reducing the compaction time and the L_0 data load time, respectively. Based on this observation, the following process is added to LAB-DB's L_0 compaction operation. When the B+ tree is scheduled to be immutable, the utilization of compacted leaf nodes is calculated and the leaf node scan method is applied until 80% is reached, and after that, the file scan method is applied to delete the log.

Second, LAB-DB utilizes NVM to write the log for hot keys. As in-place updates occur more frequently, the performance of LAB-DB increases, but the log size increases accordingly. Buffering frequently accessed keys in the NVM reduces the log size significantly by leveraging DRAM-like in-place writes. By adding a part to manage access count for each key in B+ tree, for keys exceeding a specific threshold value, the log is recorded in NVM. Before write-ahead logging, LAB-DB is modified to check the location of the log through B+ tree search.

3.4 Implementation

We implement LAB-DB by extending RocksDB [22] from Facebook (Version 5.7). We utilize *memkind* library [8] to allocate and access the byte-addressable NVM (i.e., Intel Optane DCPMM) with load/store instructions. Overall, we modify about 5,000 lines of code in RocksDB for write, read, flush, and L_0 compaction operations as follows.

Put Operation. We modify the `write_batch.cc` module. We extend its WAL mechanism to implement the buffered I/O as described in Section 3.2. We allocate 2 MB in NVM for this buffering and flush the buffer when the buffered data hit 1 MB as it is double-buffered. In addition, we also increase the size of the storage region allocated for WAL buffering to 20 GB. We also extend the MemTable to store the logical address of the current value in a storage device. Since we do not increase the MemTable size, the number of MemTable entries slightly decreases due to this overhead.

Flush. We heavily modify the `flush_job.cc` module so that the flush operation works as described in Section 3.2. For high-performance implementation of B+ tree, we utilize FAST-FAIR [26, 43], an open-source B+ tree. For the B+ tree leaf nodes, we allocate 256 MB space. We do

not pre-allocate the DRAM region for the internal nodes. Note that the size of the DRAM region for internal nodes is effectively bounded by the space allocated for the leaf nodes (e.g., 256 MB).

Compaction. We modify `version_set.cc` and `db_impl_compaction_flush.cc` to implement the modified L_0 compaction scheme. For efficient compaction of a chunk from B^+ tree and SSTable files in L_1, LAB-DB first fetches a set of values for the keys belonging to the target chunk and stores them in memory. Then, the merge is performed in memory and the outcome is stored in the storage device. We set the total size of the values in a single chunk to be similar to a single MemTable size (64 MB). Finally, we extend the compaction scheduler of RocksDB such that it takes into account the occupancy of the L_0 area in NVM assigned to determine the priority for compaction operations; that is, the new scheduler prioritizes L_0 compaction when the sizes of B^+ tree in L_0 becomes close to the pre-allocated limit (e.g., 75%).

3.5 Summary

LAB-DB proposes a solution to mitigate the compaction overhead based on the fact that unnecessary compaction frequently occurs in L_0, where duplicate keys can exist, triggering compaction in other levels. First, LAB-DB replaces L_0 with a B^+ tree structure to enable in-place updates and ensure that only unique keys exist in L_0. In-place updates greatly reduce not only L_0 compaction but also lower-level compaction overhead that may occur as a result of L_0 compaction. Second, In order to use NVM cost-effectively, LAB-DB maintains the internal nodes of the B^+ tree in DRAM and records only key and value offset information in the leaf nodes of the B^+ tree. This design component reduces the required NVM capacity by not storing values in NVM and constructing internal nodes in DRAM. Finally, LAB-DB proposes methods for selecting leaf nodes during L_0 compaction and caching hot key-value pairs to limit the growth of the WAL file while maintaining throughput.

4 EVALUATION

4.1 Experimental Setup

System Configuration. Our evaluation platform has two Intel Xeon Platinum 8260 2.4GHz 48-core processors with a heterogeneous main memory comprised of both DDR4 DRAM, Intel Optane DCPMM. We use a Samsung 840 EVO 1 TB SSD to store SSTable files and the WAL. Following the conventions in the previous works [27, 32, 33], we scale down the dataset to 90 GB to keep the running time in a manageable range. Proportionally, we restrict the DRAM capacity to 16 GB (through the use of `cgroup`) to prevent a scenario of the entire database fitting in DRAM. The operating system is Ubuntu 18.04.4, with kernel version 5.4.0. Our LAB-DB prototype extends RocksDB 5.7 and we compare it with the RocksDB of the same version.

In our experiments, we use RocksDB-NVM, a modified version of RocksDB that uses 8 GB NVM as L_0, as well as NoveLSM [33], SLM-DB [32], and MatrixKV [55] as baselines. Throughput is normalized to that of RocksDB-NVM. We measure the performance using eight client threads, two compaction threads and one flush thread. We configure the MemTable size in DRAM to be 64 MB for RocksDB-NVM, MatrixKV and LAB-DB, and NoveLSM utilizes two 2 GB NVM MemTables. MatrixKV shows the effect of fine-grained compaction only when a sufficient NVM space is secured, so we adopt 8 GB NVM following the original MatrixKV article [55]. The size of L1 is set to 256 MB, and the growth factor is set to 10, both of which are default values of RocksDB. MatrixKV and RocksDB-NVM use 8 GB NVM as L_0, and NoveLSM uses 4 GB excluding the NVM MemTable. Throughout the experiments, the configuration of NoveLSM's NVM MemTable (i.e., 2 GB) and L_0 (i.e., 4 GB), which shows the best performance of YCSB Workload A, is set. LAB-DB is configured to utilize up to 514 MB NVM space for its B^+ tree (up to 256 MB) as well as WAL

Table 1. YCSB Core Workloads

Workload	Characteristics	Read / Write Ratio
A	Write-intensive	50% / 50%
B	Read-intensive	95% / 5%
C	Read-only	100% / 0%
D	Read-latest	95% / 5%
E	Short ranges	95% / 5%
F	Read-modify-writes	50% / 50%

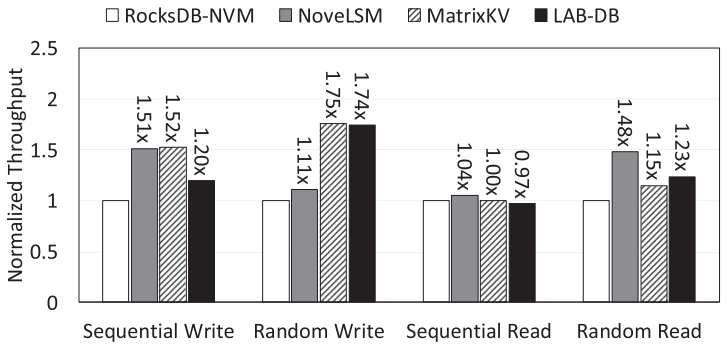

Fig. 11. Micro-benchmarks throughput.

buffer (2 MB) and hot buffer (256 MB). We set immutable B+ tree threshold parameter to 75%. We disable the compression option to control its impact on performance.

Workloads. To evaluate the performance of LAB-DB, we conduct performance evaluation by using the micro-benchmarks released with RocksDB and the **Yahoo! Cloud Serving Benchmark** (**YCSB**) [13], a widely used macro-benchmark suite. YCSB benchmark consists of a total of six workloads, each with different characteristics as listed in Table 1. The key distribution of Workload A, B, C, E, and F follows Zipfian distribution. In Workload D, the chance of accessing the recently written keys follows the Zipfian distribution (i.e., the most recently written key has a very high chance of being read soon). The key size is configured to be 8 B, and the value size is configured to be 1024 B. We run YCSB following the instructions of its official documentation. Specifically, 90 M key-value pairs are first inserted into the database, and Workload A, B, C, D, and F are run in series. Finally, Workload E is run independently.

4.2 Performance Results

Micro-benchmark Performance. Figure 11 shows the database throughput for four micro-benchmarks: sequential write, random write, sequential read, and random read. All of the results are normalized by the out-of-the-box RocksDB-NVM throughput. For all experiments, the database is pre-populated with 90 GB worth of key-value items, and the size of the key-value item is set to 1 KB. MatrixKV outperforms RocksDB-NVM with 1.52× and 1.75× speedups on sequential write and random write benchmarks. This is because MatrixKV utilizes its column compaction mechanism. However, when the NVM buffer is reduced to 4 GB, the effectiveness of column compaction is drastically reduced, and the random write performance increases only by 19% compared to the baseline. LAB-DB shows superior performance especially for random write and random read benchmarks. This is because LAB-DB reduces L_0 compaction overhead by B+ tree compaction like column compaction of MatrixKV, and also reduces L_0 search latency due to search mechanism using B+ tree.

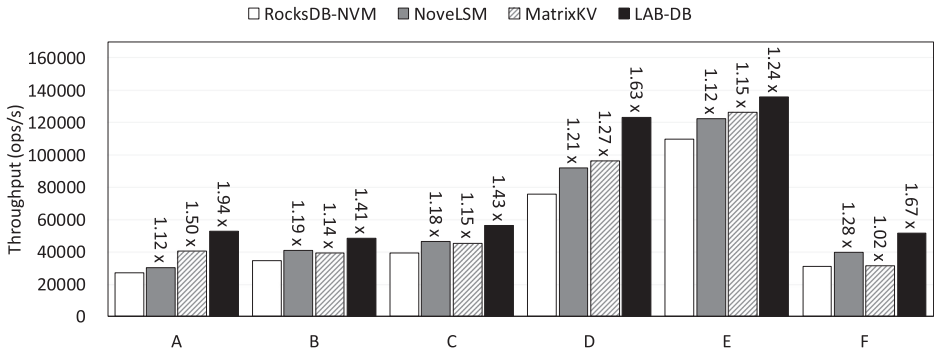

Fig. 12. YCSB throughput.

LAB-DB can cover a wide key range by efficiently using NVM, so it achieves high performance improvements in random write and random read. For the sequential read benchmark, NoveLSM, MatrixKV, and LAB-DB show comparable performance to RocksDB-NVM with <4% difference. This is expected because these databases are not optimized for sequential read benchmarks, while they are more optimized towards access patterns with a locality, which reflects real-world use cases better.

YCSB Performance. Figure 12 compares LAB-DB performance with the baseline RocksDB-NVM on YCSB workloads. We evaluate Workload A-F for an hour. As shown in the figure, LAB-DB achieves better performance than the baseline RocksDB-NVM for all YCSB workloads. Specifically, LAB-DB achieves substantially better performance than RocksDB-NVM on Workload A, D, and F. Overall, LAB-DB outperforms the baseline in write-intensive workloads (e.g., YCSB-A) with its ability to perform in-place L_0 update, which significantly reduces the need for compaction operations. LAB-DB achieves better performance than the baseline in read-intensive workloads through the efficient use of NVM spaces. Although LAB-DB only utilizes 0.25 GB NVM space for B^+ tree, it can track a large number of keys (e.g., over 16 M keys since each leaf node requires 16 B space). The values cover by those keys amount 16 $M \cdot 1\ KB = 16\ GB$. As a result, write and read requests for a substantial portion of the total dataset can be handled in L_0 without the need to access L_1 or L_2. For Workload E, LAB-DB achieves more than 24% performance improvement over RocksDB-NVM, which is a result of reduced storage access due to many accesses to keys buffered in NVM. LAB-DB has a low L_0 hit rate due to the characteristics of range scan, so it is difficult to achieve high performance improvement for Workload E, but it provides the same or better performance than the baseline.

In contrast, NoveLSM and MatrixKV report much lower throughput gains. NoveLSM achieves an overall throughput improvement of 18% compared to the baseline. NoveLSM's large NVM MemTable improves the overall performance by increasing the MemTable hit rate, but it does not bring a big performance improvement due to the high skewness of YCSB benchmark. Furthermore, we find that simply scaling the MemTable size in NoveLSM does not lead to a speedup since it dramatically degrades the read performance by slowing down the MemTable search. MatrixKV achieves a particularly large speedup on write-intensive workloads (i.e., Workload A). Since write-intensive workloads tend to benefit more from the its column compaction mechanism. On the other hand, its cross-row hint search does not lead to a huge speed up on the read-intensive Workload B and C.

Note that LAB-DB achieves much higher performance while requiring a *much smaller NVM space*. LAB-DB only uses 0.51 GB NVM, which is about 0.6% of the dataset size (90 GB). The other prior works utilize an order of magnitude larger NVM space (i.e., 8 GB) for the same dataset.

Table 2. 90–99.99% Tail Latency for YCSB Workload A

Latency (us)	Avg	90%	99%	99.9%	99.99%
Baseline	726	1,536	3,311	64,647	228,490
NoveLSM	618	1,445	2,645	53,521	209,102
MatrixKV	479	1,221	1,981	31,539	188,635
LAB-DB	406	1,198	1,559	9,404	45,332

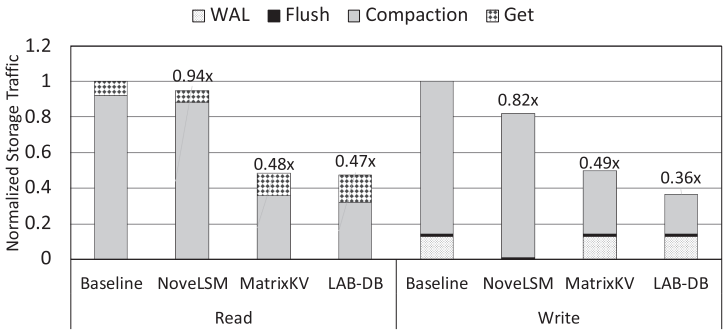

Fig. 13. Volume of storage traffic on YCSB Workload A.

Tail latency. Table 2 shows the 90–99.99% tail latency values. As shown in the table, the average tail latency is significantly smaller in LAB-DB than other works. This is because (i) the overall time spent on compaction is smaller in LAB-DB than in the other databases, and (ii) LAB-DB avoids long compactions. LAB-DB demonstrates a much greater reduction in tail latency at an extreme tail (e.g., 99.99%). On a highly skewed benchmark like YCSB, NoveLSM's large MemTable hit rate does not increase significantly compared to the baseline, and the search latency of MemTable and L_0 is quite increased. MatrixKV greatly reduces L_0 read latency due to the cross-row hint search method and a large (8 GB) NVM buffer. However, even if it uses a large L_0, the hit rate is 64% lower than LAB-DB due to the existence of duplicate keys in L_0, which reduces the effective capacity. This leads to more frequent accesses to the lower levels in the LSM tree, and hence longer tail latency.

4.3 Analysis

Read/Write Traffic Analysis. Figure 13 shows the amount of read and write accesses to the storage device for YCSB workload A. The figure shows that RocksDB-NVM (baseline) and NoveLSM access the storage device much more than LAB-DB and MatrixKV. This is mainly because (i) LAB-DB and MatrixKV incur much less compaction compared to RocksDB-NVM, and (ii) LAB-DB performs many in-place updates to L_0 residing on NVM, which reduces the volume of data that needs to be compacted. Unlike LAB-DB, however, MatrixKV triggers lower-level compaction more frequently as it cannot remove duplicate keys in L_0. This reduction in write traffic to the storage device not only leads to speedups but also improves the lifetime of flash-based storage devices. The results also show that the compaction is also a major source of read traffic to the storage device. In fact, the actual amount of data read from the storage device for a Get operation is relatively small, since a significant portion of data can be retrieved from MemTable. However, the compaction process needs to read a large amount of data from the storage device to perform a merge of multiple SSTable files in the storage device.

Flush/Compaction Time Analysis. Figure 14 compares the RocksDB-NVM (baseline), NoveLSM, MatrixKV, and LAB-DB's total time spent on compaction. The figure shows that LAB-DB

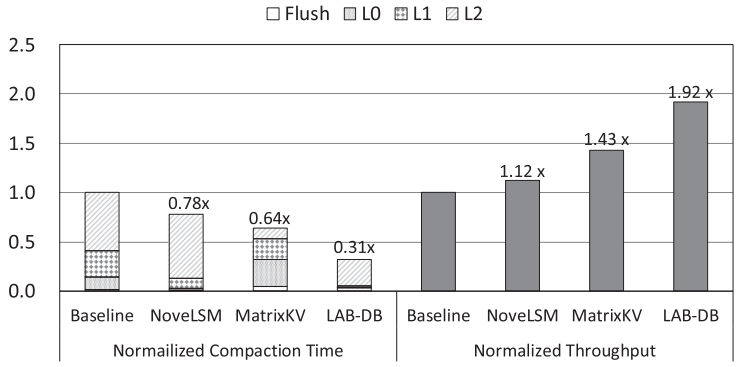

Fig. 14. Flush/compaction time on YCSB Workload A.

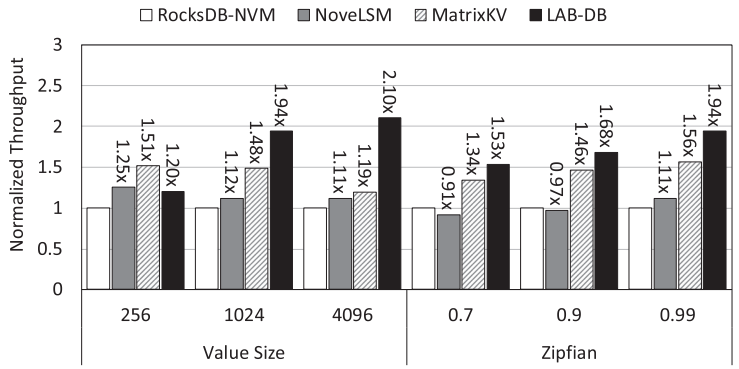

Fig. 15. Normalized throughput for Workload A varying value size and Zipfian constants.

indeed spends 69% less time on flush/compaction operations. The time spent on L_0, L_1, L_2 compaction is much smaller on LAB-DB than RocksDB-NVM, NoveLSM, and MatrixKV. MatrixKV also spends 36% smaller compaction time than RocksDB-NVM, but the compaction overhead is still relatively higher than LAB-DB. NoveLSM uses 4 GB NVM L_0 to prevent performance degradation caused by a flush operation, which increased the L_0 compaction overhead. LAB-DB increases the chance for the in-place update due to the wide L_0 key range, and also takes a small key range of the L_0 compaction, thus dramatically reducing the compaction overhead. The flush operation of LAB-DB incurs additional overhead due to B^+ tree insert. However, the increased flush time does not affect the performance because this proportion in the total compaction time is 0.5%.

Sensitivity to Value Size and Zipfian Constant. Figure 15 shows the normalized throughput on YCSB Workload A with varying value sizes and Zipfian constants. LAB-DB shows the superior performance improvement in all Zipfian constants. NoveLSM, MatrixKV, and LAB-DB all shows higher performance as the Zipfian constant was larger, because these databases are more optimized for skewed patterns. When the Zipfian constant is 0.7, MatrixKV and LAB-DB have only 1% and 16% NVM hit rates, but the normalized throughput achieve 34% and 53% of the baseline, respectively. This is because the compaction overhead is reduced compared to the baseline by the column compaction of MatrixKV and the B^+ tree compaction of LAB-DB. In addition, LAB-DB maintains a relatively high L_0 hit rate because there are no duplicate keys, so that its read latency can always be kept low. To evaluate the impact of value size, we keep the number of key-value pairs same at 90 M, only the value size was changed to 256 B, 1024 B, and 4096 B. LAB-DB shows

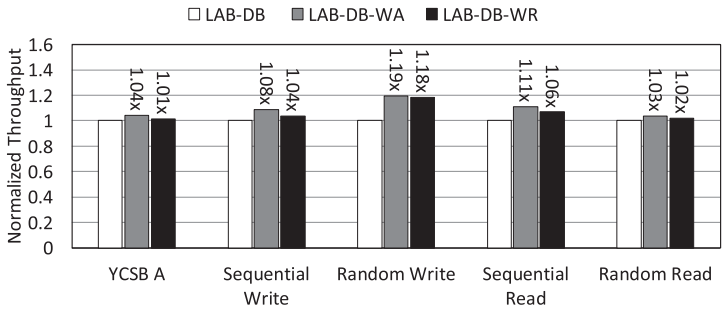

Fig. 16. Normalized throughput for micro-benchmarks and YCSB Workload A when store the WAL in the NVM.

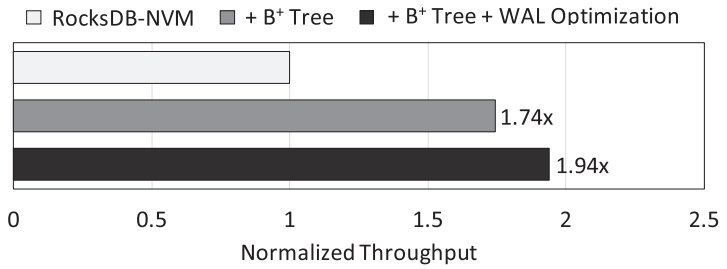

Fig. 17. Contributions to performance from different components.

the best performance when the value size is 1024 B and 4096 B, but MatrixKV shows the best performance at 256 B. This is because MatrixKV can buffer more key-value pairs in NVM as the value size decreases.

Impact of WAL Storage. Figure 16 presents the normalized throughput for the micro-benchmarks and YCSB A when the WAL is stored in NVM instead of the SSD. LAB-DB-WR allocates the remaining NVM space out of 8 GB to the WAL, and if the WAL exceeds the remaining NVM, it will be stored on the SSD. LAB-DB-WA records all of the WAL in NVM. LAB-DB-WA and LAB-DB-WR achieve geomean speedups of 9.4% and 6.6%, respectively, due to NVM's fast access time. Random Write has a significant L_0 compaction overhead, which incurs high I/O traffic, so LAB-DB-WR improves throughput 18.6% than LAB-DB. LAB-DB-WR outperforms LAB-DB in the uniform distribution benchmarks, but this gain decreases in realistic cases (e.g., YCSB-A). Thus, we use LAB-DB with the highest space efficiency by default.

Impact of Design Component. Figure 17 quantifies the performance gains coming from two different design components: B+ tree and WAL optimization. To evaluate the effectiveness of each component, we conduct experiments using YCSB Workload A. When we replaced L_0 with the B+ tree in the RocksDB-NVM baseline, throughput is improved by 74%. This was due to the reduced storage traffic required for compaction due to the in-place update of the B+ tree, and the faster L_0 search time that improves read performance. Adding WAL optimization techniques to the B+ tree can improve performance by 94% compared to the baseline. Although WAL optimization techniques are designed to prevent excessive growth of the WAL, the use of a hot buffer allows data to be read/written to fast NVM instead of slow SSD, resulting in additional performance improvements.

Impact of NVM Capacity. Figure 18 shows the impact of NVM capacity on throughput and log size. When the NVM size is set to 512 MB, the performance is improved by 72% compared

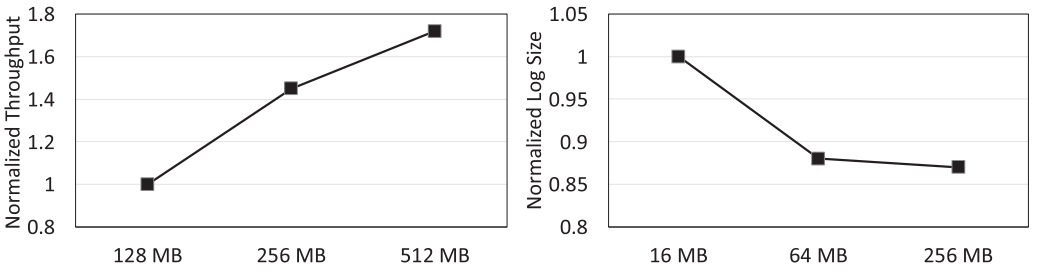

Fig. 18. Sensitivity to NVM capacity.

Table 3. Performance Comparison of Micro-benchmarks between LAB-DB and SLM-DB (ops/s)

Workload	SLM-DB	LAB-DB	Improvement
Sequential Write	28,072	30,885	10.01%
Random Write	6,179	24,241	292.26%
Sequential Read	22,816	44,315	94.21%
Random Read	2,068	1,681	-18.74%

to the case of 128 MB. As the B^+ tree grows, performance increases due to the reduction of L_0 compaction and efficient L_0 search. The figure demonstrates that the hot buffer (NVM) does not significantly reduce the log size, even if it exceeds 64 MB. The reason why the size of the hot buffer does not significantly affect the log size is that the traffic is extremely skewed with a small number of keys.

Performance Comparison with SLM-DB. Table 3 compares the throughput of SLM-DB and LAB-DB using the micro-benchmarks. Since SLM-DB supports only a single thread, LAB-DB also uses a single thread, and the live-key ratio is set to 0.7. For Random Write, LAB-DB shows higher performance than SLM-DB by 292.26%. Selective compaction of SLM-DB degrades throughput for large datasets because there are too many compaction candidates. In Sequential Read, LAB-DB outperforms SLM-DB by more than 94%. This is because, unlike SLM-DB, LAB-DB maintains an SSTable file structure (L_1, L_2, L_3...), which stores key-value pairs sequentially. For Random Read, SLM-DB provides 18.74% higher throughput than LAB-DB as it stores the index of the entire dataset in the B^+ tree.

Throughput over Time. Figure 19 shows how the throughput of LAB-DB and the control groups changes over time for YCSB Workload A. RocksDB-NVM and NoveLSM show comparable throughput patterns with a steady-state throughput at a lower level. MatrixKV maintains high performance when the NVM buffer is sufficiently large but maintains similar performance to the baseline when it is not. In contrast, LAB-DB maintains consistently high throughput over time. The only significant throughput drop during the interval between 45 and 60 seconds is attributed to the compaction operation. LAB-DB has a higher throughput bound than the other designs as the frequency of multiple-level compaction is substantially lower due to in-place updates.

5 RELATED WORK

LSM Design. Frequent compaction is a major performance bottleneck in LSM tree-based key-value stores such as RocksDB. Many prior studies have attempted to address it. Monkey [15] and Dostoevsky [16] optimize tuning points to improve performance by applying techniques such as

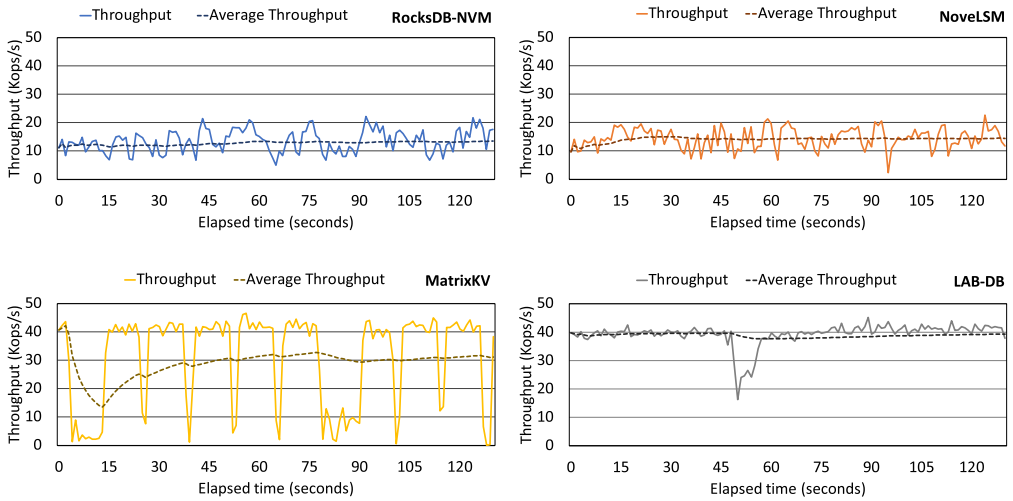

Fig. 19. RocksDB-NVM, NoveLSM, MatrixKV, and LAB-DB throughput over time on YCSB workload A.

control-capable merge and level-specific filtering policies. Other proposals aim at reducing the amount of data needed for compaction by isolating keys from values. WiscKey [40] saves values in vLog separately and performs compaction only with keys, and Bourbon [14] accelerates the search by implementing a learned index in WiscKey. HashKV [9] and Parallax [53] propose designs that reduce the overhead of managing the vLog via group-based garbage collection and hybrid KV placement. These methods reduce the compaction overhead by excluding values from compaction, but at the cost of managing values separately. Unlike the conventional key-value separation mechanism that manages the entire dataset, LAB-DB manages only L_0. This minimizes the overhead of value management while preserving RocksDB's characteristic of being advantageous for sequential reads.

TRIAD [4] keeps frequently accessed (hot) keys in MemTable and values in the WAL such that compaction does not occur for hot keys in heavily skewed workloads. This can greatly reduce the compaction overhead. However, the maximum number of hot keys is limited by the size of MemTable, and this number further goes down as the value size increases. Unlike TRIAD, which stores the hot key in the MemTable, LAB-DB uses NVM to set up a new L_0 region to provide a more scalable solution to the dataset size. SILK [5] enforces a scheduling policy to control the disk bandwidth used for background operations such as compaction and flushing and prioritize those operations at a low system load. Hailstorm [6] disaggregates compute and storage to allow load balancing and improve overall resource utilization. bLSM [48] proposes a merge scheduler that bounds write latency to ensure stable write throughput. While they bound write throughput, unlike LAB-DB, they do not reduce the compaction overhead itself.

dCompaction [44] reduces write amplification by batching compaction and hence reducing compaction frequency. PebblesDB [45] introduces a Fragmented LSM tree to save the fragmented key-value pairs in a small file. These works effectively reduce I/O traffic through virtual compaction or a fragmented LSM-tree. However, unlike LAB-DB, they could not reduce the frequency of compaction propagated to lower levels due to duplicate keys in L_0. HyperLevelDB [47], LWC-tree [54], SplinterDB [12], and PapyrusKV [34] also improve the performance of LSM tree-based key-value stores by optimizing the compaction policy. Zhang et al. [58] and Sun et al. [49] leverage FPGAs to accelerate compaction operations. Unlike these proposals, which only reduces the compaction latency, LAB-DB reduces the write traffic through in-place updates as well.

NVM-augmented LSM. NoveLSM [33] and SLM-DB [32] propose a new design to use NVM for LSM tree-based key-value stores. NoveLSM introduces a persistent MemTable to allow the MemTable entries to be moved directly to the presistent MemTable. NoveLSM demonstrates superior performance when the working set fits in the persistent MemTable. Instead, LAB-DB proposes a method to reduce compaction overhead by utilizing NVM and delivers high performance even for random write workloads. SLM-DB structures the entire database in B^+ trees and all values are handled in an append-only manner, so it requires all indexes of the database to be allocated in the persistent memory region. As a result, the NVM capacity becomes the primary limiting factor of the database size. Like SLM-DB, LAB-DB also stores only the key and offset in the NVM, but uses much smaller NVM space than SLM-DB to provide a more scalable solution. MatrixKV provides the cross-row hint search for read optimization and makes efforts to reduce compaction latency through fine-grained L_0 compaction. However, MatrixKV still suffers from the issue of duplicate keys in L_0, and in order to improve performance, MatrixKV requires a large amount of NVM. In contrast, LAB-DB leverages a smaller amount of NVM compared to MatrixKV to remove the duplicate keys, reducing the number of unnecessary compaction triggers and greatly improving performance. Bullet [25], Pmemenv [37], myNVM [21], HiLSM [38], NVLSM [57], and pmemkv [28] also propose different designs exploiting NVM technology. Also, others have proposed persistent data structures to be used in NVM, such as CDDS [50] and PFHT [17]. LAB-DB stands out from other techniques because it does not replace a portion of RocksDB storage with NVM. Instead, it reduces L_0 compaction overhead significantly by changing L_0 to a B^+ tree structure.

2B-SSD [3] takes a different approach to minimize the logging overhead by directly accessing data on the storage via **memory-mapped I/O** (**MMIO**) on SSDs supporting **Persistent Memory Region** (**PMR**), which is included in the NVMe V1.3 specifications to provide both byte-I/O and block-I/O paths. SpanDB [11] utilizes the SPDK to improve the WAL write speed and enables asynchronous requests. KVell [36] proposes a design that can maximally utilize the throughput of NVMe SSDs while retaining the index structure in DRAM. Masstree [41] goes even further, keeping all data in memory to improve throughput while ensuring data to be persistent. Unlike these proposals, LAB-DB utilizes WAL as part of L_0 and imposes no constraints on DRAM or NVM size for index maintenance.

Key-Value SSDs. PinK [27] utilizes key-value SSDs to implement an LSM tree-based key-value store. It removes Bloom filters and pins down in DRAM prefixes for search as well as top-k levels to improve the performance. KAML [31] introduces a key-value SSD with a key-value interface, which provides fine-grained transactions by utilizing multiple independent namespaces instead of coarse-grained locks used by earlier key-value SSDs. Check-In [56] suggests a way to reduce tail latency by offloading checkpointing workloads from the host's storage engine to SSDs, minimizing unnecessary data transfers. SILT [39], SkimpyStash [19], FlashStore [18], SlimDB [46], and LOCS [51] also aim at enhancing the performance of key-value stores leveraging SSDs. Unlike these works, which do not address the compaction overhead, LAB-DB proposes a way to reduce this overhead.

6 CONCLUSION

This article presents LAB-DB, a scalable solution to address the compaction bottleneck in the LSM tree-based key-value store. While the LSM tree-based key-value store handles user-provided write and read operations in a simple, performant way, it leaves the difficult work of compaction to the background threads. As a result, the compaction operation eventually becomes the bottleneck and limits the system's sustained throughput. To prevent this from happening, it is important to minimize the storage traffic caused by compaction. Unfortunately, DRAM is not large enough to filter all write requests with duplicate keys. Furthermore, simply increasing the MemTable size

turned out to be not very effective for its increased search time. To address this issue, LAB-DB utilizes NVM, which has byte-addressable and persistency characteristics. LAB-DB augments the existing LSM tree with a pair of B^+ trees resident in NVM. These B^+ trees not only mitigate the compaction bottleneck for writes but also serve read requests more efficiently. LAB-DB can be readily integrated into existing LSM trees to eliminate the compaction bottleneck in a cost-effective manner.

REFERENCES

[1] Apache. 2012. Cassandra. Retrieved December 16, 2023 from http://https://cassandra.apache.org/

[2] Apache. 2012. HBase. Retrieved December 16, 2023 from http://hbase.apache.org/

[3] Duck-Ho Bae, Insoon Jo, Youra Adel Choi, Joo-Young Hwang, Sangyeun Cho, Dong-Gi Lee, and Jaeheon Jeong. 2018. 2B-SSD: The case for dual, byte-and block-addressable solid-state drives. In *Proceedings of the 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*. IEEE, 425–438. DOI: https://doi.org/10.1109/ISCA.2018.00043

[4] 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 *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'17)*. USENIX Association, 363–375. Retrieved from https://www.usenix.org/conference/atc17/technical-sessions/presentation/balmau

[5] 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 *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'19)*. USENIX Association, Renton, WA, 753–766. Retrieved from https://www.usenix.org/conference/atc19/presentation/balmau

[6] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. 2020. Hailstorm: Disaggregated compute and storage for distributed LSM-based databases. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. Association for Computing Machinery, 301–316. DOI: https://doi.org/10.1145/3373376.3378504

[7] C. Băsescu, C. Cachin, I. Eyal, R. Haas, A. Sorniotti, M. Vukolić, and I. Zachevsky. 2012. Robust data sharing with key-value stores. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'2012)*. 1–12. DOI: https://doi.org/10.1145/1993806.1993843

[8] C. Cantalupo, V. Venkatesan, J. Hammond, K. Czurlyo, and S. Hammond. 2015. Memkind Library. Retrieved December 16, 2023 from https://github.com/memkind/memkind

[9] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. 2018. HashKV: Enabling efficient updates in KV storage via hashing. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC'18)*. USENIX Association, Boston, MA, 1007–1019. https://doi.org/10.48550/arXiv.1811.10000

[10] 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* 26, 2, Article 4 (2008), 26 pages. DOI: https://doi.org/10.1145/1365815.1365816

[11] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. SpanDB: A fast, cost-effective LSM-tree based KV store on hybrid storage. In *Proceedings of the19th USENIX Conference on File and Storage Technologies (FAST'21)*. USENIX Association, 17–32. Retrieved from https://www.usenix.org/conference/fast21/presentation/chen-hao

[12] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: Closing the bandwidth gap for NVMe key-value stores. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC'20)*. USENIX Association, 49–63. Retrieved from https://www.usenix.org/conference/atc20/presentation/conway

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*. Association for Computing Machinery, 143–154. DOI: https://doi.org/10.1145/1807128.1807152

[14] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A learned index for log-structured merge trees. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 155–171. Retrieved from https://www.usenix.org/conference/osdi20/presentation/dai

[15] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 79–94. DOI: https://doi.org/10.1145/3035918.3064054

[16] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better space-time tradeoffs for LSM-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*. 505–520. DOI: https://doi.org/10.1145/3183713.3196927

[17] Biplob Debnath, Alireza Haghdoost, Asim Kadav, Mohammed G Khatib, and Cristian Ungureanu. 2016. Revisiting hash table design for phase change memory. *ACM SIGOPS Operating Systems Review* 49, 2 (2016), 18–26. DOI: https://doi.org/10.1145/2819001.2819002

[18] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. FlashStore: High throughput persistent key-value store. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 1414–1425. DOI: https://doi.org/10.14778/1920841.1921015

[19] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2011. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. 25–36. DOI: https://doi.org/10.1145/1989323.1989327

[20] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*. Association for Computing Machinery, 205–220. DOI: https://doi.org/10.1145/1294261.1294281

[21] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 2018. Reducing DRAM footprint with NVM in Facebook. In *Proceedings of the 13th EuroSys Conference*. 1–13. DOI: https://doi.org/10.1145/3190508.3190524

[22] Facebook. 2017. RocksDB. Retrieved December 16, 2023 from https://rocksdb.org/

[23] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. 2015. Scaling concurrent log-structured data stores. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)*. Association for Computing Machinery, 14 pages. DOI: https://doi.org/10.1145/2741948.2741973

[24] Google. 2014. LevelDB. Retrieved December 16, 2023 from https://github.com/google/leveldb

[25] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. 2018. Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC'18)*. 967–979. Retrieved from https://www.usenix.org/conference/atc18/presentation/huang

[26] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable transient inconsistency in byte-addressable persistent B+-Tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*. USENIX Association, Oakland, CA, 187–200. Retrieved from https://www.usenix.org/conference/fast18/presentation/hwang

[27] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. 2020. PinK: High-speed in-storage key-value store with bounded tails. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC'20)*. 173–187. Retrieved from https://www.usenix.org/conference/atc20/presentation/im

[28] Intel. 2019. pmemkv. Retrieved December 16, 2023 from https://github.com/pmem/pmemkv

[29] Intel. 2019. Technology Brief: Intel Optane Technology. Retrieved October 12, 2020 from https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/what-is-optane-technology-brief.pdf

[30] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. *Basic Performance Measurements of the Intel Optane DC Persistent Memory Module*. Technical Report. University of California, San Diego. Retrieved from http://arxiv.org/abs/1903.05714

[31] Y. Jin, H. Tseng, Y. Papakonstantinou, and S. Swanson. 2017. KAML: A flexible, high-performance key-value SSD. In *Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*. 373–384. https://doi.org/10.1109/HPCA.2017.15

[32] 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 *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. USENIX Association, Boston, MA, 191–205. Retrieved from https://www.usenix.org/conference/fast19/presentation/kaiyrakhmet

[33] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for nonvolatile memory with NoveLSM. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'18)*. USENIX Association, 993–1005.

[34] Jungwon Kim, Seyong Lee, and Jeffrey S Vetter. 2017. PapyrusKV: A high-performance parallel key-value store for distributed NVM architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14. DOI: https://doi.org/10.1145/3126908.3126943

[35] Redis Labs. 2009. Redis. Retrieved December 16, 2023 from https://github.com/redis/redis

[36] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: The design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 447–461. DOI: https://doi.org/10.1145/3341301.3359628

[37] Lucas Lersch, Ismail Oukid, Ivan Schreter, and Wolfgang Lehner. 2017. Rethinking DRAM caching for LSMs in an NVRAM environment. In *Proceedings of the European Conference on Advances in Databases and Information Systems*. Springer, 326–340. DOI: https://doi.org/10.1007/978-3-319-66917-5_22

[38] Wenjie Li, Dejun Jiang, Jin Xiong, and Yungang Bao. 2020. HiLSM: An LSM-based key-value store for hybrid NVM-SSD storage systems. In *Proceedings of the 17th ACM International Conference on Computing Frontiers*. 208–216. DOI: https://doi.org/10.1145/3387902.3392621

[39] Hyeontaek Lim, Bin Fan, David Andersen, and Michael Kaminsky. 2011. SILT: A memory-efficient, high-performance key-value store. In *SOSP'11 - Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. 1–13. DOI: https://doi.org/10.1145/2043556.2043558

[40] 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* 13, 1, Article 5 (2017), 28 pages. DOI: https://doi.org/10.1145/3033273

[41] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*. 183–196. DOI: https://doi.org/10.1145/2168836.2168855

[42] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. 2015. NVMKV: A scalable, lightweight, FTL-aware key-value store. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC'15)*. 207–219. Retrieved from https://www.usenix.org/conference/atc15/technical-session/presentation/marmol

[43] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A hybrid SCM-DRAM persistent and concurrent B-Tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16)*. Association for Computing Machinery, 371–386. DOI: https://doi.org/10.1145/2882903.2915251

[44] Fengfeng Pan, Yinliang Yue, and Jin Xiong. 2017. dCompaction: Delayed compaction for the LSM-tree. *International Journal of Parallel Programming* 45, 6 (2017), 1310–1325. DOI: https://doi.org/10.1007/s10766-016-0472-z

[45] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. Association for Computing Machinery, 497–514. DOI: https://doi.org/10.1145/3132747.3132765

[46] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A space-efficient key-value storage engine for semi-sorted data. *Proceedings of the VLDB Endowment* 10, 13 (2017), 2037–2048. DOI: https://doi.org/10.14778/3151106.3151108

[47] Jeff Dean Sanjay Ghemawat. 2019. HyperLevelDB. Retrieved from https://github.com/rescrv/HyperLevelDB

[48] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 217–228. DOI: https://doi.org/10.1145/2213836.2213862

[49] Xuan Sun, Jinghuan Yu, Zimeng Zhou, and Chun Jason Xue. 2020. FPGA-based compaction engine for accelerating LSM-tree key-value stores. In *Proceedings of the 2020 IEEE 36th International Conference on Data Engineering (ICDE'20)*. IEEE, 1261–1272. Retrieved from https://www.usenix.org/conference/fast20/presentation/zhang-teng

[50] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the FAST*. 61–75. Retrieved from https://www.usenix.org/conference/fast11/consistent-and-durable-data-structures-non-volatile-byte-addressable-memory

[51] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the 9th European Conference on Computer Systems*. 1–14.

[52] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based ultra-large key-value store for small data items. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC'15)*. USENIX Association, Santa Clara, CA, 71–82. Retrieved from https://www.usenix.org/conference/atc15/technical-session/presentation/wu

[53] Giorgos Xanthakis, Giorgos Saloustros, Nikos Batsaras, Anastasios Papagiannis, and Angelos Bilas. 2021. Parallax: Hybrid key-value placement in LSM-based key-value stores. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'21)*. Association for Computing Machinery, 305–318. DOI: https://doi.org/10.1145/3472883.3487012

[54] Ting Yao, Jiguang Wan, Ping Huang, Xubin He, Fei Wu, and Changsheng Xie. 2017. Building efficient key-value stores via a lightweight compaction tree. *ACM Transactions on Storage* 13, 4 (2017), 1–28.

[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 *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC'20)*. USENIX Association, 17–31. Retrieved from https://www.usenix.org/conference/atc20/presentation/yao

[56] Joohyeong Yoon, Won Seob Jeong, and Won Woo Ro. 2020. Check-In: In-storage checkpointing for key-value store system leveraging flash-based SSDs. In *Proceedings of the 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA'20)*. IEEE, 693–706. DOI: https://doi.org/10.1109/ISCA45697.2020.00063

[57] Baoquan Zhang and David H. C. Du. 2021. NVLSM: A persistent memory key-value store using log-structured merge tree with accumulative compaction. *ACM Transactions on Storage* 17, 3, Article 23 (2021), 26 pages. DOI: https://doi.org/10.1145/3453300

[58] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, Zhongdong Huang, and Jianling Sun. 2020. FPGA-accelerated compactions for LSM-based key-value store. In *Proceedings of the18th USENIX Conference on File and Storage Technologies (FAST'20)*. 225–237. Retrieved from https://www.usenix.org/conference/fast20/presentation/zhang-teng

Received 5 November 2022; revised 3 May 2023; accepted 28 August 2023