NVLSM: A Persistent Memory Key-Value Store Using Log-Structured Merge Tree with Accumulative Compaction

BAOQUAN ZHANG and DAVID H. C. DU, University of Minnesota–Twin Cities

Computer systems utilizing byte-addressable **Non-Volatile Memory** (**NVM**) as memory/storage can provide low-latency data persistence. The widely used key-value stores using **Log-Structured Merge Tree** (**LSM-Tree**) are still beneficial for NVM systems in aspects of the space and write efficiency. However, the significant write amplification introduced by the leveled compaction of LSM-Tree degrades the write performance of the key-value store and shortens the lifetime of the NVM devices. The existing studies propose new compaction methods to reduce write amplification. Unfortunately, they result in a relatively large read amplification. In this article, we propose NVLSM, a key-value store for NVM systems using LSM-Tree with new accumulative compaction. By fully utilizing the byte-addressability of NVM, accumulative compaction uses pointers to accumulate data into multiple floors in a logically sorted run to reduce the number of compactions required. We have also proposed a cascading searching scheme for reads among the multiple floors to reduce read amplification. Therefore, NVLSM reduces write amplification with small increases in read amplification. We compare NVLSM with key-value stores using LSM-Tree with two other compaction methods: leveled compaction and fragmented compaction. Our evaluations show that NVLSM reduces write amplification by up to 67% compared with LSM-Tree using leveled compaction without significantly increasing the read amplification. In write-intensive workloads, NVLSM reduces the average latency by 15.73%–41.2% compared to other key-value stores.

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

Additional Key Words and Phrases: Non-volatile memory, key-value store, log-structured merge tree

ACM Reference format:
Baoquan Zhang and David H. C. Du. 2021. NVLSM: A Persistent Memory Key-Value Store Using Log-Structured Merge Tree with Accumulative Compaction. *ACM Trans. Storage* 17, 3, Article 23 (August 2021), 26 pages.
https://doi.org/10.1145/3453300

1 INTRODUCTION

Persistent memory systems use DRAM for peripheral data and **Non-Volatile Memory** (**NVM**) for data persistence. However, building systems using NVM is challenging since data consistency can be a problem when the system crashed. The introduced performance overhead to guarantee data consistency for writes may counteract the performance benefits of NVM [7, 33, 39, 45]. In addition,

This work was partially supported by NSF I/UCRC Center Research in Intelligent Storage and NSF awards 1439622, 1525617, and 1812537.
Authors' addresses: B. Zhang and D. H. C. Du, University of Minnesota–Twin Cities, 319-15th Avenue S.E., Minneapolis, MN 55455; emails: zhan4281@umn.edu, du@umn.edu.

1553-3077/2021/08-ART23 $15.00
https://doi.org/10.1145/3453300

certain types of NVM devices suffer from short endurance. Applications running on NVM devices have to minimize the number of write operations to increase the lifetime of NVM devices [50].

In a modern storage infrastructure, key-value stores, like LevelDB [22] and RocksDB [20] using **Log-Structured Merge Tree** (**LSM-Tree**) [56], have attracted more attention for write-intensive workloads [19, 26, 66]. Existing studies have pointed out that maintaining batch updates using LSM-Tree is still beneficial for NVM, given that the speed of NVM is multiple times slower than DRAM [35]. In addition, the capacity of NVM is limited compared to traditional storage devices. LSM-Tree is space-efficient since it does not reserve free space for future updates like B+Tree. Therefore, deploying LSM-Tree on persistent memory systems to achieve great performance improvements and reduce space overheads is critical to many upper-layer applications.

An LSM-Tree organizes key-value pairs into multiple components with increasing data capacity. New key-value pairs will firstly be inserted into the smallest component. A process called compaction migrates key-value pairs to the next component if the size of the current component has reached a threshold. However, the most widely used leveled compaction migrates data with read-merge-writes leading to large write amplification. That is, the size of data written to the NVM device is multiple times larger than that from applications. In persistent memory systems, the large write amplification considerably degrades the insert performance due to the data consistency overhead. Moreover, it also potentially shortens the lifetime of NVM devices. Therefore, it is crucial to reduce the write amplification for the LSM-Tree key-value store deployed on NVM.

Some other LSM-Tree key-value stores use tiered compaction to achieve a small write amplification, but they significantly increase read amplification. In addition, tiered compaction requires a larger transient space for a single compaction [25, 71]. Monkey [15] and Dostoevsky [16] have discussed the tradeoffs among overheads of write, read, and space. PebblesDB [58] employs a fragmented compaction to reduce write amplification. Fragmented compaction can achieve a similar write amplification as tiered compaction while limiting the required transient space for single compaction. However, it still results in a relatively large read amplification. In this article, we intend to answer the question: *Can we reduce write amplification in LSM-Tree using NVM without significantly increasing read amplification with similar space overhead as in fragmented compaction?*

To address this question, we propose **NVLSM:** a design of LSM-Tree key-value store for persistent memory systems using a new compaction scheme called Accumulative Compaction. Accumulative compaction fully leverages the byte-addressable data accesses of NVM by including a pointer with each key-value pair. A sorted run of key-value pairs is migrated from one component to the next larger component by building a new floor using these pointers instead of using read-merge-writes to rewrite data. A merging operation will only be carried out when the number of floors reaches a threshold. It thereby achieves a small write amplification with similar space amplification as fragmented compaction. In addition, NVLSM constructs fine-grained links among key-value pairs between floors to enable a cascading search to limit/reduce read amplification.

With accumulative compaction, NVLSM achieves small write and read amplifications simultaneously. The threshold of the maximum number of floors can be used to obtain a tradeoff between write and read amplifications. A larger threshold reduces write amplification more significantly. However, the read amplification increases. We compare NVLSM with key-value stores using LSM-Tree with leveled compaction and fragmented compaction. The results indicate that NVLSM reduces the write amplification by up to 67% compared with LSM-Trees using leveled compaction. It achieves a small write amplification close to that of LSM-Trees using fragmented compaction and a smaller read amplification comparable with that of LSM-Trees using leveled compaction. In write-intensive workloads, NVLSM reduces the average latency by 15.73%–41.2% compared to other key-value stores.

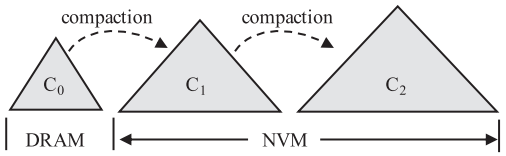


Fig. 1. LSM Tree.

The rest of the article is organized as follows. Section 2 introduces persistent memory systems and discusses the LSM-Trees on persistent memory systems. Section 3 discusses the benefits of LSM-Tree on NVM. Section 4 presents the design and implementation of NVLSM. Section 5 discusses some related work. Section 6 evaluates the performance of NVLSM. Section 7 offers conclusions and future work.

2 BACKGROUND

2.1 Persistent Memory Systems

The emerging NVM technologies, like **Spin-Transfer Torque Memory** (**STT-RAM**) [63], **Phase Change Memory** (**PCM**) [59], 3D XPoint [24], and so on, are changing the architecture of storage systems. However, current NVM technologies cannot replace DRAM [67, 76]. In this article, we target persistent memory systems with a hybrid main memory architecture including a large but a bit slower NVM sitting on the memory bus together with a small and fast DRAM. Compared to the systems using traditional storage devices, e.g., hard disks or flash drives, persistent memory systems provide low-latency and byte-addressable data accesses.

Building systems for NVM is challenging since NVM suffers from an asymmetric performance for reading and writing. An unexpected shutdown may result in data inconsistency due to the small atomic update granularity, and the instruction reorders [5, 11, 46, 49, 51]. Writes to NVM are typically protected by some mechanisms, e.g., write-ahead log, copy-on-write, log-structured updates combining with *mfence* [43], and *clflush* [42] instructions [2, 3, 9, 32, 48, 60]. The performance overhead of ensuring data consistency during updates should be minimized to utilize the high performance of NVM systems fully. In addition, certain types of PCM-based devices have shorter endurance. With a large number of write operations, the lifetime of an NVM device can be significantly shortened [21, 50]. Therefore, applications running on NVM systems have to minimize their write operations to NVM.

2.2 LSM-Tree

LSM-Tree is a write-optimized data structure widely used in many key-value stores [18, 19, 22, 26]. Figure 1 shows one possible structure of LSM-Tree on persistent memory systems. LSM-Tree organizes key-value pairs into multiple components (or levels in LevelDB). The sizes of components are increased by a configurable size ratio. New key-value pairs are only inserted to the smallest component C_0, which is typically small enough to be kept in DRAM. Then a data migration process called compaction migrates data from a smaller component to a larger component when the size of the smaller component reaches a threshold.

However, the most widely used leveled compaction in LevelDB [22] leads to a large write amplification, i.e., the data size written to NVM is multiple times larger than that from user applications. The left side of Figure 2 shows an example of the leveled compaction process. In LSM-Tree with leveled compaction, a component includes multiple sorted runs with disjoint key ranges. Each sorted run consists of multiple key-value pairs in sorted order with a total bounded size. There is a fixed size ratio between the numbers of the sorted runs in two adjacent components. In our

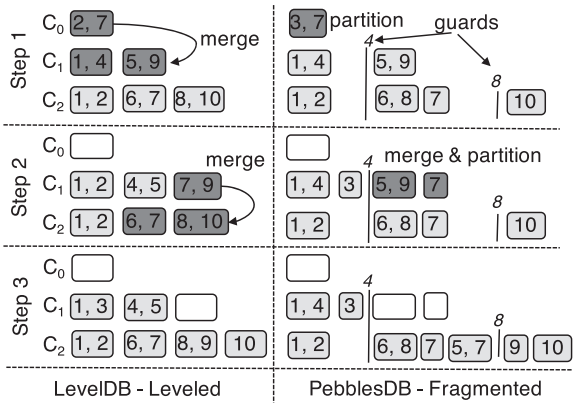


Fig. 2. LSM-Tree compaction strategies.

example, for explanations, the size ratio is 2. C_0 is the smallest component in DRAM. Its size is set for a sorted run. When migrating data, the leveled compaction is executed with a rolling-merge process to maintain the disjoint key ranges. In our example in Step 1 of the left part of Figure 2, the leveled compaction will firstly read {2,7} from C_0. Then, {1,4} and {5,9} from C1 will also be read since they are overlapped with {2,7} in C_0. The merging results {1,2}, {4,5}, and {7,9} are written to C_1 in Step 2. However, the size of C_1 is larger than its threshold. A new compaction will be triggered to merge {7,9} from C_1 and {6,7} and {8,10} from C_2.

The read amplification for an LSM-Tree using leveled compaction is small since only one sorted run needs to be checked for each component during a search. However, the rolling merge operation leads to a large number of data rewrites on the same component. Monkey [15] and Dostoevsky [16] have analyzed the operations of leveled compaction. They identify that a key-value pair may be rewritten multiple times on the same component. A large number of rewrites seriously increase write amplification. Considering the high write overhead and the short endurance of NVM, a larger write amplification degrades the insertion/update performance and shortens the lifetime of NVM devices significantly.

Tiered compaction migrates data between components with a smaller write amplification, but it results in a larger read amplification for searching. Tiered compaction is less used since it also requires a large transient space for a compaction operation [25, 71]. A type of fragmented compaction (the right part of Figure 2) is proposed by PebblesDB [58] to reduce write amplification. In fragmented compaction, components are partitioned with guard keys. A guard in a smaller component will also be a guard for larger components. The number of guards in a component increases with the growth of the component size. Sorted runs in the same guard can have overlapped key ranges.

Besides the size ratio of components, PebblesDB also sets a maximal number of sorted runs in a guard. When reaching the maximal number, sorted runs in a guard will be merged and sorted. The merge results will be partitioned and written based on the guards of the next component. In our example in the right part of Figure 2, the fragmented compaction will read {3,7} from C_0. Instead of reading overlapping sorted runs from C_1, fragmented compaction partitions {3,7} directly based on the guard 4 in C_1 in Step 1. Then the partition results {3} and {7} are added to the corresponding guards directly.

Fragmented compaction reduces the write amplification significantly since data will not be rewritten in the same component. In addition, it requires a smaller transient space and time for single compaction comparing with tiered compaction. However, fragmented compaction still

introduces a relatively larger read amplification since multiple sorted runs have to be potentially checked in each component. Since fragmented compaction performs better than tiered compaction, in this article, we will mainly compare the proposed Accumulative Compaction with fragmented compaction.

3 BENEFITS OF LSM-TREE ON NVM

Existing studies redesign LSM-Tree for DRAM-NVM hybrid systems [35]. NoveLSM identified that maintaining batch updates of LSM-Tree is also significant for NVM since the speed of NVM is multiple times slower than that of DRAM. However, the benefits of LSM-Tree for NVM are not comprehensively discussed. Therefore, in this section, we discuss the benefits of LSM-Tree for persistent memory systems.

Designing consistent and durable data structures, e.g., hashing index [17, 53, 77], radix tree [39], and B+Tree [1, 12, 27, 55, 73], has been an essential research topic for persistent memory systems. However, compared to B+Tree variants, some of them, e.g., hash index, radix tree, and so on, are less effective for range queries. Therefore, our analysis focuses on comparing LSM-Tree to B+Tree, which can provide efficient operations for both single-point look-ups and range queries. Compared to the existing B+Tree designs, LSM-Tree has two major benefits on NVM.

High write efficiency while providing good read efficiency. Some B+Tree designs for NVM sort the keys [27] or deploy sorted indirect indexes [12] in their nodes. Therefore, a key can be found by binary searches within nodes. In addition, range queries can be executed with sequential reads. However, inserting new keys in these designs will result in large amounts of data shifts and movements. The extra writes for the data movements will degrade the write performance and potentially shorten the lifetime of certain types of NVM devices. Other designs of B+Tree on NVM do not keep the keys in nodes sorted all the time [1, 12]. Inserting a new key will not move the existing keys in tree nodes. Therefore, write overheads are reduced. However, the search performance is degraded significantly since linear searches are required for the unsorted updates [1].

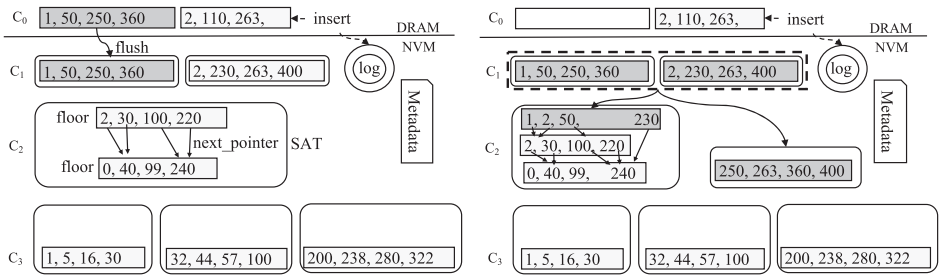
Some other designs deploy both sorted and unsorted nodes [55, 73]. They store sorted inner nodes in DRAM and unsorted leaf nodes on NVM. After a system failure, all inner nodes will be rebuilt using leaf nodes. Existing researches [27] consider that they are not precisely persistent data structures since they consume significant DRAM space and require long recovery time after a system failure.

LSM-Tree can achieve good write efficiency since it only performs log-structured writes on NVM. In the meantime, LSM-Tree can also have a good read performance, especially for range queries, since key-value pairs are sorted in each component. Our final evaluations show that LSM-Tree can achieve better performance in write-intensive workloads compared to the state-of-the-art B+Tree designs on NVM, although the single-point searches of LSM-Tree are less efficient.

High space efficiency. B+Tree reserves free space in each node for future updates. Significant space overheads are introduced for NVM. LSM-Tree is space-efficient since it only requires a small write buffer to receive new updates. Since the sizes of components are increased with a size ratio, e.g., 10 in LevelDB, only a small portion of data has multiple versions. In our final evaluations, LSM-Tree key-value stores achieve smaller space overhead than the state-of-the-art B+Tree key-value store. Considering that the capacity of NVM is limited compared to traditional storage devices, the space-efficient LSM-Tree is still an alternative index structure on NVM.

4 NVLSM WITH ACCUMULATIVE COMPACTION

In this section, we discuss NVLSM, an LSM-Tree key-value store using accumulative compaction to reduce write amplification without significantly increasing read amplification.



(a) Step 1: flushing sorted run from DRAM (b) Step 2: merging C_1 and building floors on C_2

Fig. 3. Accumulative compaction in NVLSM.

4.1 Accumulative Compaction

Accumulative compaction fully leverages the byte-addressability of NVM. The principal of our design is not to rewrite key-value pairs on the same component before it is migrated to the next component while accelerating the search efficiency. It is accomplished by having a pointer associated with each key-value pair. Figure 3 shows the overall design of NVLSM. As shown in Figure 3(a), NVLSM keeps C_0 in DRAM. New key-value pairs will be inserted and sorted in the DRAM buffer of C_0. When the DRAM buffer is full, the data will be flushed to C_1 directly by writing them to NVM-based memory. C_1 includes a small number of sorted runs with overlapped key ranges. In each C_i ($i > 1$), NVLSM stores key-value pairs with a new data structure called **Skip-Array Tree** (**SAT**). SATs in the same component have disjoint key ranges.

The SAT structure is specially designed for accumulative compaction. An SAT organizes key-value pairs in sorted runs with multiple floors using pointers. A floor is a sorted array of key-value pairs. When a sorted run is migrated to the next component, it will be added to an SAT by adding a new top floor. Therefore, existing data on the SAT will not be rewritten. To accelerate the searching process among the floors in an SAT, each key-value pair in the newly migrated sorted run has a next pointer pointing to the first key which is not smaller than the current key in the existing floors. The target key for the next pointer will be found from the old top floor. If the old top floor does not have an eligible key, we will continue to search the next bottom floor. Therefore, the keys pointed by the next pointer can be on any existing floors. In the SAT of C_2 of Figure 3(a), the pointers of keys 2 and 30 on the top floor are pointing to key 40 on the bottom floor, while in Figure 3(b), the next pointer of key 230 points to the key 240 on the next bottom floor. We will discuss the detailed algorithm later in Section 4.2. With the help of the next pointers, seeking a key in an SAT can be executed by a cascading search starting from the top floor down to the bottom floor following the next pointers.

Figure 3(b) demonstrates the process of compacting key-value pairs from C_1 to C_2. C_1 consists of multiple sorted runs with overlapped key ranges. We firstly merge all key-value pairs in C_1 into a single large sorted run before partitioning them. If C_2 has no SATs (a sorted run is also considered as an SAT), the merging results will be partitioned into several sorted runs with the default size. Otherwise, the merging results are partitioned based on the key ranges of the SATs in C_2. Each partitioned will be inserted into the SAT with the overlapped key-range in C_2. The insertion is done by updating the next pointer of each key-value pair. Therefore, a new top floor (sorted run) is added to the existing SAT with links to the lower floors of this SAT.

With the help of the cascading searching following the pointers, searching among too many floors still decreases the search performance. Therefore, we set a maximum number of floors in each SAT. If the number of floors in an SAT reaches the maximal value, the floors in the SAT

will be merged into one single sorted run, and then partitioned and added to SATs in the next component. To not increase the total number of components, if the current component is the last component that still has space capacity, the sorted results will be written back to the current component without adding a new component.

The accumulative compaction fully leverages the byte-addressable data accesses of NVM to achieve small read and write amplification simultaneously. By utilizing the new SAT structure, it achieves a similar write amplification as fragmented compaction and a similar read compaction as leveled compaction. It reduces the write amplification since the existing data in the component (except the last component) will never be rewritten during a "compaction" until the maximal number of floors is reached. In the meantime, the cascading searching on byte-addressable NVM within an SAT limits the increase of read amplification.

The maximal number of floors in SATs (*Max_Floor*) influences the tradeoff among write, read, and space amplification. When *Max_Floor* is set to 1, accumulative compaction becomes the same as the leveled compaction. That is, it keeps small read and space amplification while results in large write amplification. As *Max_Floor* increases, write amplification is reduced, but both read and space amplification are enlarged. However, the increase of read amplification is not significant with the help of the cascading searching process.

4.2 NVLSM Implementations

We use a **Persistent Memory Development Kit** (**PMDB**) [31] to implement NVLSM. The C++ bindings to libpmemobj [29] of PMDK use *make_persistent_atomic<ClassType>(arg0,arg1,...)*, in which *ClassType* is the declared CPP class and *arg0, arg1,...* are the arguments to construct an object of the *ClassType*, to atomically allocate space from NVM pool. With the persistent smart pointer $persistent\_ptr$, we can allocate space for objects and access members of an object on NVM in the same way with that in DRAM. An $NVLSM$ object contains an array of $Components$. Each *Component* includes an array of SATs. SATs in the same component are sorted based on their key ranges since they have disjoint key ranges. The array of components is used as metadata, i.e., searching a key will search the metadata to identify the target SAT in each component. After compaction, the metadata will also be updated since SAT may be deleted or created after compaction. When we update the metadata, we will ensure the data consistency so that NVLSM can be recovered from a system failure. We will discuss the data consistency guarantees in Section 4.4.

SAT is the core data structure in Accumulative Compaction. Figure 4 shows the proposed SAT structure. An SAT includes an array of floor pointers (p_floor_i) (each pointer is associated with a sorted array of key-value pairs), a key range *(p_range)* covered by SAT, and a set of bloom filters *(p_bf)* (one for each floor). The predefined maximal number of floors will determine the length of the pointers. A floor consists of key-value pairs with key indexes and values. Each value has a corresponding index entry. An index entry can identify the offset *(val_offset)* and length *(val_len)* of a value. Currently, we use 4-byte *val_offset* to make sure it can index a large sorted run. With a 2-byte *val_len*, the length of values can be up to 64 KB. The size of *(val_len)* is configurable. In our current implementations, the size of *(val_len)* is 2 bytes. It can be configured with larger sizes if workloads include a value size larger than 64 KB. We separate the index and value to handle values with variable lengths. The binary search on a floor will be performed in the index area. During the searching process, we only need one memory access to the index entry for each iteration. After we get the index entry of the target key, we need one more memory access to get the value. Floors in the same SAT will have an incremental *id*. A topper/newer floor in an SAT will have a larger id.

Each index entry also includes a *next pointer* pointing to the index entry of the first equal or larger key-value pair in the lower floors. The *next pointer* has two fields, *btm_floor* and *btm_offset*.

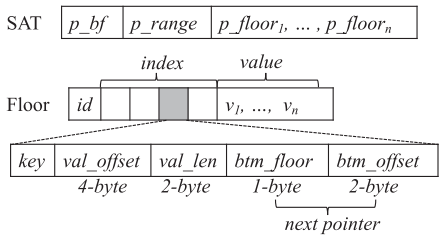


Fig. 4. SAT structure.

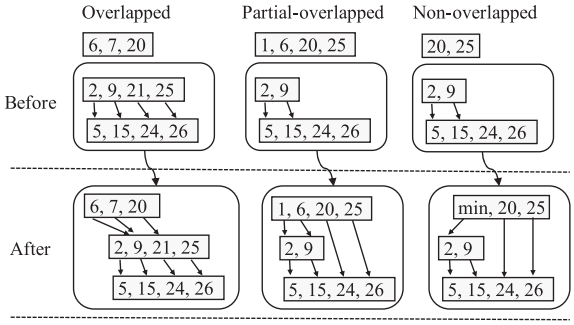


Fig. 5. Adding a new floor to an SAT.

The *btm_floor* stores the position of a lower floor in the pointer array. With one-byte *btm_floor*, the maximal number of floors in an SAT can be up to 256. *btm_offset* indicates the position of the corresponding index entry in a lower floor. The 2-byte *btm_offset* can work with an index up to 65,535 entries. Note that the size of an index entry can be adjusted based on the configuration of value lengths and sorted run size.

Adding a new floor. To support Accumulative Compaction, an SAT can accept new floors dynamically. When adding a new sorted run to an SAT, the new added sorted run will become the new top floor. To update the next pointer of the new top floor, we start from the existing top floor. The process will be executed by scanning both the new top floor and the existing top floor $floor_n$. After we find the first key in $floor_n$ which is not smaller than the current key in the new floor, we update the pointer of the current key on the new floor. If we run out all keys on $floor_n$ and the new floor still has keys with NULL pointers, we will continue to go to the next $floor_m$. The $floor_m$ and the start location to scan $floor_m$ will be determined by the next pointer of the last key of $floor_n$. Therefore, the next pointers of keys in the new top floor can point to any existing floors. Accumulative Compaction introduces extra reads to build new links between floors compared to Fragmented Compaction. However, the total number of keys to read will be limited. We may read all keys in the old top floor and may only read a small portion of keys in some other floors. Therefore, the extra reads overhead will not be significant compared to Fragmented Compaction.

Figure 5 shows the three conditions when adding a new floor—Overlapped, Partially overlapped, and Non-overlapped. To simplify the figure, we only show the keys in the index of each floor as a sorted run in the SAT and use arrow lines to represent *next pointers*. The overlapped condition means the end key of the new sorted run is smaller than the end key of the current top floor. In the example of the overlapped condition as shown in Figure 5 left column, we want to add a new sorted run which is $\{6, 7, 20\}$ to an SAT. The end key of the new sorted run is 20, and it is smaller than the end key 25 of the current top floor. All next pointers in the inserted sorted run will be set

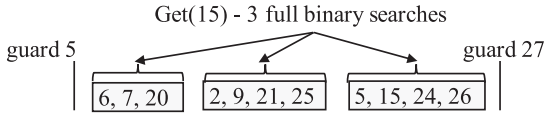


Fig. 6. Get(15) in a guard of Fragmented Compaction.

to point to the corresponding keys on the existing top floor. Note that the next pointers of both 6 and 7 on the new floor will point to 9 on the current top floor. Finally, the sorted run {6, 7, 20} becomes the new top floor.

In the partially overlapped condition (Figure 5 middle column), the end key 25 of the new floor is larger than the end key 9 of the current top floor. Moreover, the first key 1 is smaller than the end key 9 of the current top floor. After we go through every key in the current top floor, there are still keys, 20 and 25, in the new floor whose next pointers are empty. Therefore, we will continue to go to the next bottom floor following the next pointer of key 9 in the old top floor. As shown in Figure 5 middle column the next pointers of 1 and 6 point to 2 and 9, respectively, in the current top floor, and the next pointers of 20 and 25 will point to the 24 and 26 in the bottom floor.

At last, if the inserted sorted run is non-overlapped with the existing top floor (Figure 5 column 3), the start key 20 of the new floor is larger than the end key 9 of the current top floor. If we update pointer directly, the current top floor may be omitted during a cascading searching following the pointers. Therefore, we will add a virtual key *min* at the beginning of the new floor. The key *min* is a special number smaller than any key in the workload. It does not influence the range of a floor and will be ignored when merging the floors in an SAT. As shown in Figure 6 column 3, the next pointer of *min* will point to the start key 2 of the current top floor, and the next pointers of 20 and 25 in the inserted sorted run will point to the 24 and 26 in the bottom floor, respectively. In this way, the searching process will not skip the current top floor. If a key of the inserted run is larger than all existing keys, the next pointer will point to NULL.

Bloom filters and the key range of the floors in an SAT will be updated after adding a new top floor. NVLSM utilizes in-place updates for bloom filters and its key range. Although in-place updates may be risky on NVM, the compaction can tolerate the caused data inconsistency after a system failure (this will be further discussed in Section 4.4).

Cascading searching in SAT. In an LSM-Tree, searching for a target key always starts from the smallest component. The existing compaction methods reducing write amplification like tiered or fragmented compaction, result in large read amplifications. Figure 6 shows the process of Get(15) in a guard of fragmented compaction. When Get(15), the LSM-Tree can locate one guard in a component. However, a guard may include multiple sorted runs. In this example, the guard 27 includes three sorted runs. We have to perform up to three full binary searches. Therefore, the read amplification is increased.

NVLSM reduces the write amplification while limiting the increase of read amplification. In a component, we firstly identify the SAT to search based on their key ranges. By leveraging the byte-addressable data accesses of NVM, looking for a key in an SAT will be executed with a fractional cascading searching. Algorithm 1 describes the algorithm of the cascading searching in an SAT. The *{floor, start, end}* in line 1 defines the floor and the range to search.

In the beginning, we start from the top floor, and the searching range will be from beginning 0 to its end (*floor->size - 1*) (line 3). The binary search will stop at the location *stop*. The key at the *stop* location will be the last key which is not greater than the target key (*target*). If the target key is found, the searching process will be terminated (lines 4 and 5). Otherwise, we will follow the next pointer of the key at *stop* location.

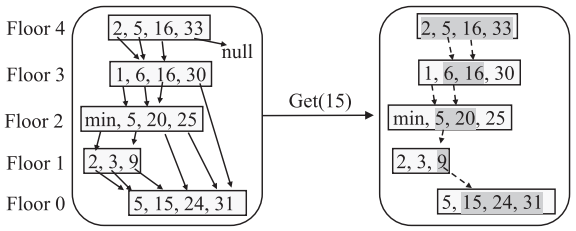


Fig. 7. The critical path of Get(15).

If the next key pointed by the next pointer is smaller than the target key (line 10), the searching range for the next floor will be limited by the next pointers of *stop* and *stop + 1*. If the next key is larger than or equal to the target key (line 22), the searching range of the next floor will be limited by the next pointers of *stop - 1* and stop.

In some special conditions, the two next floors pointed by *stop* and *stop + 1* or *stop - 1* are different (lines 16 and 17 and lines 28–33). In these conditions, we will further compare the ids of two candidate floors, *next_floor* and *next_floor_2*. The next iteration will be on the floor with the larger id.

Note that our algorithm ignores the logic handling some corner cases. For example, if the *stop* is at the end of a floor after a binary search, *next_offset_2* will also be found using *stop* instead of *stop + 1*. If the *stop* is at the beginning of a floor after a binary search, *next_offset_2* will also be found using *stop* instead of *stop - 1*.

Figure 7 uses an example to show the cascading searching process. The left part of the figure shows the current status of an SAT with five floors. The right part demonstrates the critical path (marked with the dark color) of getting key 15. If bloom filters indicate that the target key may exist in an SAT, NVLSM will do a binary search on Floor 4 (the top floor). The searching process will stop at the Key_i, which is the first key that is not greater than the target key 15. In our example, the search on Floor 4 will stop at key 5. If Key_i is equal to the target key, the searching process will terminate. Otherwise, the searching process will continue to search. In our example, we will continue to check the key in Floor 3 pointed by the next pointer of key 5 on Floor 4.

The key 6 in Floor 3 pointed by the key 5 in Floor 4 is still smaller than the target key 15. Therefore, we get the next floor and key pointed by the Key_{i+1}, 16, in Floor 4. Since keys 6 and 16 pointed by the next pointers of keys 5 and 6 in Floor 4 are on the same Floor 3. Then our next binary search will be executed on Floor 3 with the range between keys 6 and 16. As a result, the searching for Floor 3 will stop at key 6.

However, key 20 in Floor 2 pointed by key 6 in Floor 3 is larger than the target key 15. In this condition, NVLSM will check the next pointer of key 1, the previous key of key 6, in Floor 3. The searching range of Floor 2 will be between the keys 5 and 20 pointed by key 1 and key 6 in Floor 3, respectively. The search for Floor 2 will stop at key 5.

Since the key 9 in Floor 1 pointed by key 5 in Floor 2 is smaller than the target key 15, we further check the key 20 in Floor 2. However, key 20 in Floor 2 is pointing to key 24 in Floor 0. In this condition, the next search process will go to Floor 1 since it has a larger id. The searching range of Floor 1 will be between the beginning of Floor 1 and the key 9. The binary search process of Floor 1 is optimized using its key range information. Since the key range to search of Floor 1 is *<2, 9>* smaller than the key 15, the binary search will directly stop at the end of the search range key 9.

Since key 9 is the end key of Floor 1 and it is still smaller than the target key 15, the searching range of Floor 0 will be between key 15 pointed by key 9 in Floor 1 and the end key 31. The binary search in Floor 0 covers a larger range than other floors since we can only reduce the range using one pointer from Floor 1. However, this condition is less frequent.

ALGORITHM 1: Cascading searching

```
1  floor, start, end = top_floor, 0, floor->size - 1
2  while floor != NULL do
3      stop = binarySearch(floor, start, end)
4      if floor->getKey(stop) == target then
5          return floor->getValue(stop)
6      end
7      next_floor = floor->getNextFloor(stop)
8      next_offset = floor->getNextOffset(stop)
9      next_key = next_floor->getKey(next_offset)
10     if next_key < target then
11         next_floor_2 = floor->getNextFloor(stop + 1)
12         next_offset_2 = floor->getNextOffset(stop + 1)
13         if next_floor->id == next_floor_2->id then
14             floor, start, end = next_floor, next_offset, next_offset_2
15         end
16         if next_floor->id > next_floor_2->id then
17             floor, start, end = next_floor, next_offset, next_floor->size - 1
18         end
19         if next_floor->id < next_floor_2->id then
20             floor, start, end = next_floor_2, 0, next_offset_2
21         end
22     else
23         next_floor_2 = floor->getNextFloor(stop - 1)
24         next_offset_2 = floor->getNextOffset(stop - 1)
25         if next_floor->id == next_floor_2->id then
26             floor, start, end = next_floor, next_offset_2, next_offset
27         end
28         if next_floor->id > next_floor_2->id then
29             floor, start, end = next_floor, 0, next_offset
30         end
31         if next_floor->id < next_floor_2->id then
32             floor, start, end = next_floor_2,next_offset_2, next_floor_2->size - 1
33         end
34     end
35 end
```

Differences from other compaction methods. Compared to leveled compaction and fragmented compaction, the accumulative compaction fully leverages the byte-addressable data accesses of NVM to achieve small read and write amplification simultaneously. Leveled compaction will read one sorted run from C_i and all overlapping sorted runs from C_{i+1}, and merge and write them back to C_{i+1}. It keeps sorted runs in the same component with disjoint key ranges such that one sorted run only needs to be checked when searching a key. However, fragmented compaction will only read one sorted run from C_i and write the merge results to C_{i+1} directly. Therefore, the write amplification is reduced since sorted runs on C_{i+1} will not be rewritten. However, the read

amplification is increased since a component may have overlapping sorted runs. Multiple runs need to be checked for each component when searching a key. The proposed accumulative compaction utilizes SAT structure in each component. An SAT structure can manage sorted runs as floors and receive new sorted runs without rewriting the existing runs. Therefore, the write amplification is reduced. In addition, it builds links between floors to accelerate the searching process. The read amplification is not enlarged significantly although an SAT includes overlapping sorted runs.

4.3 Key-Value Pair Operations

As a key-value store, NVLSM supports major key-value operations including **Put**, **Delete**, **Get**, **Seek**, and **Next**.

Put/Delete. As shown in Figure 3(a), during a Put operation, a new key-value pair will be buffered and sorted in the write buffer of C_0 in DRAM and protected by a persistent log on NVM. If the write buffer is full, it will be flushed to C_1 on NVM directly. Thus, C_1 will have overlapped SATs with only one floor (i.e., a sorted run). After C_1 reaches the maximal size, all data in C_1 will be merged into a single sorted run and then partitioned into multiple sorted runs with a bounded size. These sorted runs will be migrated to the next component using Accumulative Compaction. The Delete is also executed as a Put operation with a value of delete mark. The invalid key-value pairs will be dropped during future merging operations.

Get. Get operation will perform a single-point look-up. NVLSM will search the key with the order of the write buffer of C_0, every SAT in C_1 and one SAT in each other components. As we discussed in Section 4.2, looking for a key in SAT will be executed with a cascading search.

Seek and Next. The Seek and Next operations are used for range queries. A Seek operation can locate the position of the first key which is not smaller than the target key. Then a Next operation will return the next larger key. An SAT has an iterator consisting of sub-iterators of the included floors. The seeking process in an SAT is also executed with a cascading searching process. After a seek operation, sub-iterators of floors will stop at the largest key that is not greater than the target key. If all keys in a floor are smaller than the target key, the sub-iterator of the floor will be invalid. If all keys in a floor are larger than the target key, the sub-iterator of the floor will stop at the first key of the floor. The Next process in an SAT will iterate all valid sub-iterators of floors simultaneously.

4.4 Data Consistency and Recovery

Unexpected system shutdowns may result in data inconsistency in NVM. Therefore, NVLSM has to ensure data consistency on NVM to recover from a system failure. Specifically, NVLSM needs to ensure data consistency during two processes: writing to the persistent log for the data stored in DRAM buffer and the compaction process. We protect the key-value pairs in the DRAM write buffer by appending them to persistent logs on NVM with a delimiter. The append operation will be followed by *clflush* and *mfence* immediately.

Meanwhile, we use a compaction log, similar to SLM-DB [34], to record an entry before each operation of ongoing compaction. Table 1 shows the log entries for compacting *SAT* 0 from C_1 to *SAT* 1 and *SAT* 2 in C_2. The start entry $\{start, C_1,\ SAT\ 0\}$ indicates NVLSM will start the compaction. Then we start to merge *SAT* 0 and add the results *Run* 1 and *Run* 2 to *SAT* 1 and *SAT*2, respectively. After safely building the new floors in *SAT* 1 and *SAT* 2, NVLSM deletes the *SAT* 0 in *C*_1 and indicates the compaction has been finished. If the system fails during compaction, NVLSM will redo the operation recorded by the last entry and continue the ongoing compaction.

Metadata records key ranges and locations of SATs included in each component. During compaction, Metadata will be updated. We use a similar mechanism with LevelDB, Log-And-Apply, to

Table 1. Compaction Log

operation	void*, void*
start	C_1, *SAT* 0
merge	*Run* 1, *Run* 2
add	*SAT* 1, *Run* 1
add	*SAT* 2, *Run* 2
delete	C_1, *SAT* 0
commit	

reduce the update overheads [52]. NVLSM keeps two versions of Metadata: a persistent version on NVM and a volatile version in DRAM. After compaction, the volatile version of Metadata will be updated directly. After a system failure, the volatile version metadata can be recovered using the persistent version metadata and the compaction log.

The compaction log introduces six cache line flushes for each compaction. Protected by the compaction log, no flushes are needed during the compaction. Compared to the inserts, compaction is triggered less frequently. The number of flushes introduced by the compaction log is insignificant compared to that from inserts. Therefore, the amortized number of cache line flushes is one for each insert operation.

4.5 Discussions and Limitations

Key-value separation. Separating keys from values can considerably reduce write amplification since values will not be involved in the read-merge-writes [8, 44]. NVM is suitable for key-value separations since it provides a good random read performance. However, Key-Value separation consumes more space based on the existing results in [44]. Therefore, the garbage collection mechanism needs to be further optimized.

After separating keys and values in all leveled compaction, fragmented and accumulative compactions are still subject to the tradeoffs as mentioned earlier among write, read, and space amplification. In our article, we have an evaluation (Figure 14) with extremely small values to represent key-value separation designs in some degree. The results indicate that accumulative compaction can still achieve better write performance compared to the existing compaction methods.

Component partition. To limit the transient space and time requirement for single compaction, accumulative compaction partitions a component into smaller key ranges. When Accumulative Compaction migrates key-value pairs into an empty component, it partitions the key-value pairs into small sorted runs based on a maximal size. If the next larger component is not empty, key-value pairs will be partitioned based on the existing data. However, Accumulative Compaction can also be compatible with other partition methods including dynamic guards of fragmented compaction.

Multi-thread implementation. RocksDB [20] improves the compaction efficiency with a multi-thread compaction. Other studies [35, 58] deploy multi-thread seeks/reads to improve the seek/search performance. All of LSM-Tree key-value stores can deploy multi-thread compaction and search to improve the write and read performance. With the same number of compaction threads, leveled compaction still has the worst efficiency since it results in writing the most data. With the same number of search processes, fragmented compaction still has the worst search performance since it has to search the largest number of sorted runs. Multi-thread implementations and concurrent controls are out of the scope of this article. We will leave it as our future work.

Accumulative compaction on block storage. The proposed accumulative compaction can also be applied to LSM-Tree on block storage by deploying block-level pointers. The write

amplification of LSM-Tree can be reduced significantly since key-value pairs will not be rewritten on the same level during a compaction. Fractional cascading searching can also achieve benefits for the searching process on block storage as studied in existing work [41, 62].

Other optimizations. Cache-conscious data structures [13] can be used in memory to boost performance by increasing CPU cache line efficiency [54, 64, 65]. The current index of floors uses a plain array to store the index entries. However, it is possible to use other cache-conscious index designs for each floor. In addition, NVLSM utilizes a legacy design of the DRAM component from LevelDB. It is compatible with existing studies [34, 35, 40] that optimize the DRAM component.

5 RELATED WORK

LSM-Tree and NVM. LSM-Tree has also been optimized for persistent memory systems. NVM-Rocks [40] and SLM-DB [34] optimize the LSM-Tree on systems with both NVM and storage. They cannot be applied on systems including only NVM. NoveLSM [35] redesigns LSM-Tree for NVM. It uses a mutable NVM memory table and parallel searches to reduce the write and read latency. It still uses leveled compaction to merge sorted files. The proposed accumulative compaction in NVLSM is a complementary compaction scheme to NoveLSM and can also be applied in NoveLSM for compacting sorted files.

Reducing write amplification in LSM-Trees. Monkey [15] and Dostoevsky [16] have analyzed the tradeoffs of read, write, and space overheads between leveled compaction and tiered compaction. Existing studies [8, 44, 75] have reduced the write amplification of leveled compaction.

Tiered compaction is used in RocksDB [20], Cassandra [37], Scylla [25], and so on. Tiered compaction significantly reduces the write amplification but enlarges the read and space amplification. In addition, tiered compaction requires a large transient space and time for a compaction operation [25]. Several existing studies propose other compaction methods to reduce write amplification. PebblesDB [57] uses fragmented compaction to migrate data. LSM-trie [69] and SlimDB [61] divide a level (component) into multiple sub-levels, although they are inefficient for range queries. LWC-Tree [74] proposes a type of lightweight compaction by appending key-value pairs from a smaller component into the sorted runs of a larger component. They can achieve a write amplification close to that of tiered compaction and require less transient space and time overheads for single compaction. However, they still result in a large read amplification.

Fractional cascading searching. Fractional cascading searching [10] is a data structure technique to reduce the searching overheads. TokuDB uses a fractal tree with cascading searching to index the stored data [36]. FD-Tree [41] and bLSM [62] use cascading searching to accelerate the searching process in FD-Tree and LSM-Tree. However, they do not target on reducing the write amplification.

6 PERFORMANCE EVALUATION

6.1 Experimental Setup

We emulate NVM with 56 GB NVDIMM [68]. By adding CPU spinning controlled by Timestamp Counter in write and read operations of the NVDIMM module, we approximately emulate the write/read latency of NVM devices. We emulate different types of NVM by varying the delay of the read and write operations. Assume that the read and write latency in DRAM is 50 ns [49, 51]. We emulate two types of NVM. NVDIMM is battery-backed DRAM [68]. It represents the NVM that provides similar read and write latencies as DRAM. PCM [38] stands for the NVM with a larger capacity, but is slower than DRAM. In our evaluations, we set the read/write latency of PCM to 100 ns/400 ns, respectively [49, 70]. By default, we use PCM timing to conduct evaluations unless with other specifications. Our evaluations will compare NVLSM to LSM-Tree key-value stores using

different compaction methods. In addition, to comprehensively understand the performance of LSM-Tree on NVM, our evaluation also includes the state-of-the-art B+Tree designs on NVM.

B+Tree key-value stores. We implement B+Tree key-value stores based on PMEMKV [30], a key-value store implemented using a Persistent Memory Development Kit (PMDK) [31]. In PMEMKV, we implement the **FAST+FAIR Tree** (**FFTree**) [27], which outperforms other major designs of B+Tree on NVM in the existing evaluations in [27]. In FFTree, we use the default node size 512 B.

LSM-Tree key-value stores. Existing LSM-Tree key-value stores are not designed for NVM. They rely on file systems to manage space and block I/O to access data. The proposed accumulative compaction, which requires byte-addressable data accesses, cannot be implemented on the existing key-value stores. In addition, file system interfaces and data serializations also introduce significant performance overheads compared to the B+Tree key-value store. Therefore, to implement accumulative compaction with byte-addressable data accesses and provide a fair comparison with the B+Tree key-value store, we implement NVLSM using PMDK, as discussed in Section 4.2, and add NVLSM to PMEMKV as a new storage engine.

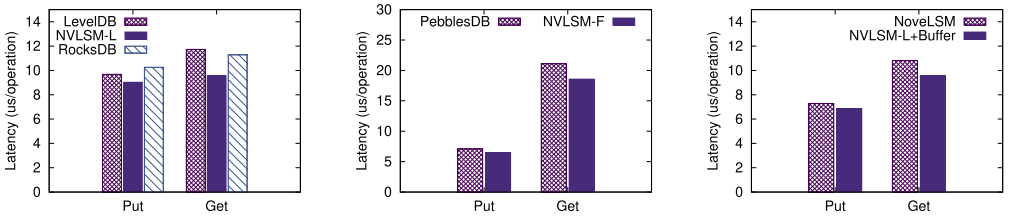
To discuss the different compaction methods in LSM-Tree and provide a fair comparison to FFTree and NVLSM, we have implemented leveled compaction and fragmented compaction in NVLSM. NVLSM-L is an LSM-Tree key-value store using leveled compaction implemented by PMDK based on PMEMKV, and NVLSM-F uses LSM-Tree with fragmented compaction. Instead of using block-based Sorted files, NVLSM-L and NVLSM-F also use a plain array, similar to a floor in an SAT in NVLSM, to store key-value pairs in each sorted run.

To validate the performance of NVLSM-L and NVLSM-F, we also deploy existing LSM-Tree key-value stores—LevelDB/RocksDB (leveled compaction) [19, 22] and PebblesDB (fragmented compaction) [58]. We also set up the state-of-the-art LSM-Tree design for NVM, NoveLSM [35]. We do not include LSM-Trees with tiered compaction either since the write amplification of tiered compaction is close to that of fragmented compaction. However, it requires a relatively larger transient space and more time for a compaction [47].

NOVA [70] is a log-structured file system for NVM systems. It achieves the best performance for append-only workloads satisfying the write behaviors of LSM-Trees. In our evaluation, we run LevelDB, RocksDB, PebblesDB, and NoveLSM on the NOVA file system with direct accesses. However, extra performance overheads are introduced by file system interfaces, data serializations, and block I/Os compared to the key-value stores based on PMEMKV.

LevelDB and NVLSM-L support only single thread. Therefore, we also use single thread in RocksDB for performance comparisons. In addition, different key-value stores, e.g., LevelDB [22], RocksDB [20], HyperLevelDB [28], and so on, apply different optimizations for many other purposes. Some of the optimizations may influence the performance of key-value stores to some degree, but they do not significantly change the tradeoffs among write, read, and space amplifications of the deployed compaction methods. For instance, multi-thread compactions in RocksDB cannot reduce the write amplification compared with single-thread compaction in LevelDB [58, 75]. NoveLSM uses a mutable NVM buffer and parallel searching to reduce write latency and read latency significantly. It still uses the leveled compaction for sorted files on NVM [35]. The proposed accumulative compaction of NVLSM can be integrated with the above optimizations for compacting sorted files. The purpose of our evaluations is to discuss the overhead tradeoffs among different compaction methods. Therefore, our evaluations do not include extra optimizations in existing key-value stores to provide a fair comparison to NVLSM-L and LevelDB and fully understand the tradeoffs of different compaction methods.

We compare NVLSM-L and NVLSM-F to LevelDB and PebblesDB, respectively. To compare NoveLSM, we also add an NVM buffer to NVLSM-L (NVLSM-L+Buffer). The NVM buffer in



(a) LevelDB/RocksDB V.S. NVLSM-L (b) PebblesDB V.S. NVLSM-F (c) NoveLSM V.S. NVLSM-L+Buffer

Fig. 8. Baseline validations.

NVLSM-L+Buffer has the same size as that of NoveLSM. The size of the DRAM write buffer is 2 MB. We disable the data compressor and set the bloom filters to 10 bits per key. The maximal size of a sorted run is 2 MB. By default, in PebblesDB and NVLSM-F, a guard includes at most 10 sorted runs. In NVLSM, the maximal number of floors in SATs is also 10 if without other specifications. In NoveLSM, similar to the ratio in [35], the size of the NVM buffer is set to about 10% of the total data size. The remaining data is stored as sorted files on NVM.

In this comparison, the key size is 16 bytes. We set the value size to 128 bytes, which is the same as in [23] since the real-world workloads are dominated with small values [4, 69]. Then we use db_bench, which is the same used in [23] to run workloads of random inserts (Put) and reads (Get). Figure 8 shows the average latency (μs/operation). NVLSM-L and NVLSM-F manage and operate the NVM device directly bypassing the file system. They use a unified data structure for both NVM and DRAM to eliminate the data serializations. NVLSM-L and NVLSM-F reduce the latency by 10%–15% in both Puts and Gets compared to LevelDB, RocksDB, and PebblesDB.

With a same-size NVM buffer, NVLSM-L+Buffer also outperforms NoveLSM. Our article focuses on the discussions on the different compaction methods. Therefore, we do not include the NVM buffers proposed by NoveLSM in the rest of our evaluations. In addition, instead of using LevelDB and PebblesDB, we use NVLSM-L and NVLSM-F to represent LSM-Tree key-value stores with leveled compaction and fragmented compaction to provide fair comparisons to NVLSM and FFTree.

6.2 Write, Read, and Space Amplification

In the first evaluation, we discuss the tradeoffs among **Write Amplification** (**WA**), **Read Amplification** (**RA**), and **Space Overheads** (**Space**) in different key-value stores. We still set key/value sizes to 16/128 bytes. A node in FFTree includes 32 keys. A sorted run in LSM-Tree key-value stores includes about 14,400 key-value pairs. We still use db_bench to run workloads of random inserts and reads. WA and RA are measured for workloads of random inserts and reads, respectively. Every workload includes 50 million operations.

To get WA and RA, we record the total size of key-value pairs written or read by key-value stores and that by db_bench, respectively. To get space overheads, we remember the number of sorted runs in LSM-Tree key-value stores and the number of nodes and values in B+Tree key-value stores.

Figure 9 shows the WA, RA, and space overheads of the evaluated key-value stores. Compared to LSM-Tree key-value stores, FFTree has the smallest RA due to the B+Tree nature of high-degree fan-out. However, the WA of FFTree is about 1.8× higher than that of NVLSM-L. Since nodes of FFTree stores sorted keys, inserting to FFTree leads to a large number of data shifts. In addition, unfilled leaf nodes increase the usage of NVM space. FFTree consumes 25.8%–48.4% larger space compared with those of LSM-Tree key-value stores.

Among all LSM-Tree key-value stores, NVLSM-L using leveled compaction achieved the smallest RA since they only check one sorted run in most components. However, their WA is enlarged

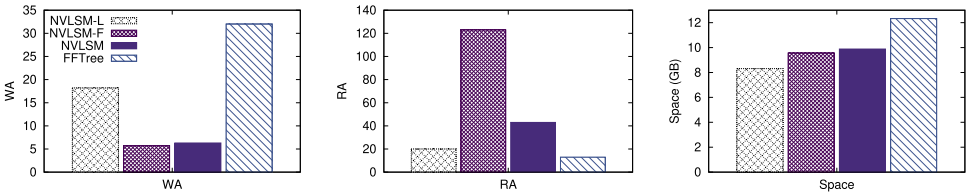


Fig. 9. Write, read amplification and space overhead.

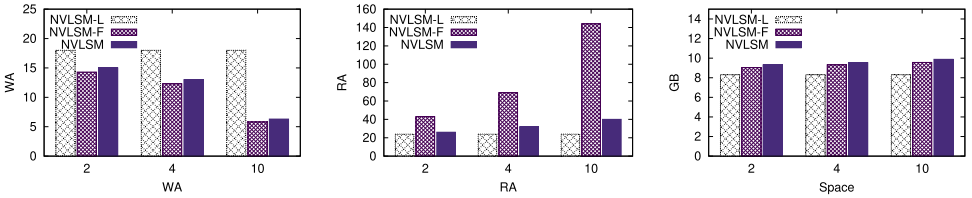


Fig. 10. Write, read amplification and space overhead with different Max_Floor.

by read-merge-writes compaction. NVLSM-F using fragmented compaction reduces WA significantly. The WAs of NVLSM-F are only about 35.4% of that in NVLSM-L. However, it suffers from a large RA since it has to check up to 10 sorted runs in each component. The RA of NVLSM-F is 6.1× larger than that of NVLSM-L. Since multiple versions of data may exist in different sorted runs in the same component, NVLSM-F occupies about 37.6% more space than NVLSM-L.

NVLSM realizes a smaller WA and RA at the same time comparing with NVLSM-L and NVLSM-F. With 128-byte values, NVLSM reduces the WA by up to 67% compared to NVLSM-L. The WA of NVLSM is close to those of NVLSM-F. Meanwhile, the RA of NVLSM is smaller than those of NVLSM-F by 69.5%. However, NVLSM has the largest space overhead since the inter-floor pointers need extra space. The increase of space is not significant with a value size of 128 bytes, i.e., it increases about 2.6% compared with that of NVLSM-F. The space overheads can be more insignificant with larger value sizes.

The maximal number of floors in an SAT (*Max_Floor*) influences the tradeoffs among WA, RA, and space overhead. More floors can reduce WA more significantly. However, it may lead to a larger RA and space overhead. Figure 10 shows the WA, RA, and Space of NVLSM-L, NVLSM-F, and NVLSM with different *Max_Floor* (*x*-axis). In this experiment, we vary *Max_Floor* with 2/4/10 (NVLSM-2/4/10). The WA and RA are still measured during the workloads of Put and Get, respectively.

Figure 10 shows the impact on the tradeoffs among WA, RA, and Space from *Max_Floor*. Results indicate that a larger *Max_Floor* can reduce WA more significantly. However, it also increases RA and Space overhead. When *Max_Floor* is 2, NVLSM-2 reduces WA by 15% compared with NVLSM-L. When *Max_Floor* is increased to 10, NVLSM-10 can reduce WA by 65% compare with NVLSM-L.

Although the RAs of both NVLSM and NVLSM-F are enlarged with the increase of *Max_Floor*, by comparing the RAs of NVLSM and NVLSM-F, we find that the inter-floor pointers and the cascading search in an SAT mitigate the RA increase significantly. When *Max_Floor* is 10, the RA of NVLSM-F is enlarged by 6× compared with that of NVLSM-L. With the help of the cascading searching, the RA of NVLSM-10 is only 1.67× that of NVLSM-L.

The multi-floor SAT increases space overhead since old versions of key-value pairs may co-exist on different floors. A larger *Max_Floor* will lead to a slightly higher space overhead. The space overhead of NVLSM is similar to NVLSM-F. When *Max_Floor* is 2, the space of NVLSM is about 12.03% higher than that of NVLSM-L. When *Max_Floor* is increased to 10, the space overhead of

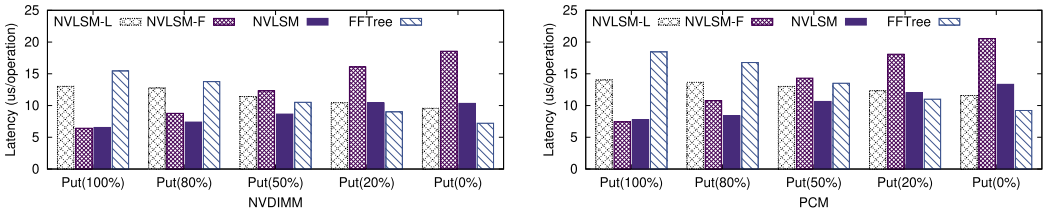


Fig. 11. Latency for single-point operations with 128-byte values.

NVLSM-10 is about 18.8% higher than that of NVLSM-L. By comparing NVLSM with NVLSM-F, we find that the inter-floor pointers of SATs do not introduce significant space overhead. The space overhead of NVLSM-10 is about 2.7% higher than NVLSM-F.

6.3 Performance Comparison

We run two types of workloads, single-point operations, and range queries, using db_bench. The workloads of single-point operations include mixed random writes and reads with different write ratios. We also run range queries with different lengths. Range(48) stands for small range queries. Range(48) firstly seek() for a random key and then next() 48 times and Range(1,024) means we call the next() function 1,024 times after seek(). We record the average latency after each workload.

6.3.1 Single-Point Operations. Figure 11 shows the latency (μs/operation) of single-point operations. In this evaluation, we still set key/value size to 16/128 bytes. We vary the write ratios among 100%, 80%, 50%, 20%, and 0% (all reads).

LSM-Tree and FFTree. LSM-Tree key-value stores provide better performance than FFTree in write-intensive workloads since inserting in FFTree leads to massive data movements. In addition, in FFTree, a search operation is performed before each insert, introducing considerable latency.

In Put(100%) and Put(80%) workloads on NVDIMM, the latency of all LSM-Tree key-value stores is shorter than FFTree by up to 57.58%. However, FFTree outperforms LSM-Tree key-value stores in all read-intensive workloads. In Put(20%) and Put(0%), the latency of FFTree is shorter than those of LSM-Tree key-value stores by up to 61.09%.

Different compaction methods. Among LSM-Tree key-value stores, NVLSM-L achieves the best performance in the random reads-only workload (Put 0%). In Put(0%) on NVDIMM, the latency of NVLSM-L is shorter than NVLSM-F and NVLSM by 48.58% and 7.4%, respectively.

NVLSM-F offers the best performance in the write-only workload. In Put(100%) on NVDIMM, the latency of NVLSM-F is shorter than those of NVLSM-L by 50.6%. NVLSM achieves similar performance to NVLSM-F in write-only workloads indicating that building links between floors in Accumulative Compaction does not introduce significant read overhead. In addition, accumulative compaction reduces write amplification without significantly increasing read amplifications.

The real workloads will include both reads and writes. NVLSM provides the best performance in mixed read and write workloads compared to other compaction methods. In Put(80%) on NVDIMM, the latency of NVLSM is shorter than those of NVLSM-L and NVLSM-F by 41.2% and 15.73%, respectively. In Put(50%) on NVDIMM, the latency of NVLSM is shorter than NVLSM-L and NVLSM-F by 24.25% and 29.79%, respectively. Figure 12 shows the throughputs of all key-value stores; in Put(80%) and Put(50%), NVLSM achieves the highest throughput. The throughput of NVLSM is 1.34×–1.47× higher than those of other key-value stores.

Latency sensitivity. From the results of NVDIMM and PCM in Figure 11, with the increases of NVM latency, NVLSM-F has the least latency increases for Put(100%) since it achieves the smallest write amplification. From NVDIMM to PCM, the latency of NVLSM-F in Put(100%) is increased by

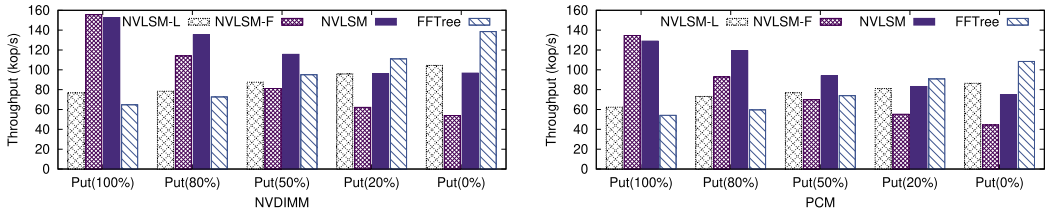


Fig. 12. Throughput for single-point operations with 128-byte values.

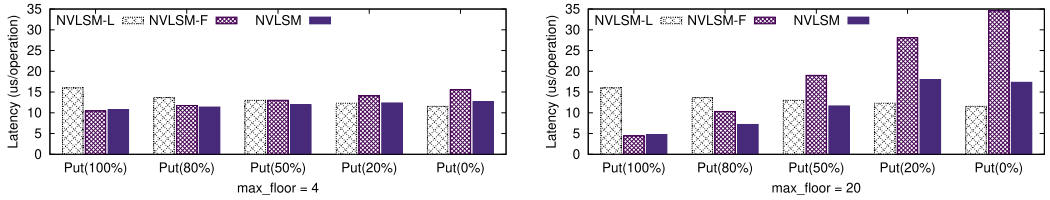


Fig. 13. Latency with different Max_Floor.

15.55%, while the latencies of NVLSM-L, NVLSM, and FFTree are increased by 23.07%, 18.32%, and 23.72%, respectively.

FFTree achieves the least latency increases in Put(0%) since it has the smallest read amplification. In Put(0%), the latency of FFTree is increased by 13.80%. The latencies of NVLSM-L, NVLSM-F, and NVLSM are increased by 21.05%, 27.77%, and 22.26%, respectively.

With the latency of NVM increases, NVLSM has the least latency increases in all other mixed workloads. The Put(50%) latency of NVLSM is increased by 19.12%, while the latencies of Put(50%) of NVLSM-L, NVLSM-F, and FFTree are increased by 25.19%, 24.38%, and 28.31%. It is because NVLSM reduces the write amplification significantly, and only less significantly increases the read amplification with the help of the cascading searching.

Different maximal numbers of floors. The parameter *Max_Floor* configures the maximal number of floors to accumulate before merging an SAT in Accumulative Compaction. In Fragmented Compaction, we also use *Max_Floor* to define the maximal number of files in a guard. As we discussed, *Max_Floor* influences the tradeoffs among read, write and space amplifications. In this evaluation, we investigate the influences on the latency from *Max_Floor*.

Figure 13 shows the results of *Max_Floor = 4 and 20*, respectively. From *Max_Floor = 4* to *Max_Floor = 20*, compared to NVLSM-L, NVLSM-F can reduce the latency more significantly in Put(100%) workloads. When *Max_Floor = 4* and *Max_Floor = 20*, NVLSM-F reduces the latency by 37.64% and 65.77%, respectively, compared to NVLSM-L in Put(100%). NVLSM can achieve similar performance improvements in Put(100%). When *Max_Floor = 4* and *Max_Floor = 20*, NVLSM reduces the latency by 35.88% and 63.27%, respectively, compared to NVLSM-L.

However, in Put (80%) workload, when *Max_Floor = 20*, the performance improvements achieved by NVLSM-F are less significant than NVLSM. Compared to NVLSM-L, NVLSM-F reduces latency by 23.07%, whereas NVLSM can reduce the latency by 46.15%. The reason is that the read performance of NVLSM-F is significantly degraded when the number of files in a guard is too large.

With the help of cascading searches, the read performance of NVLSM is less degraded compared to NVLSM-F when *Max_Floor* increases. in Put(0%) workload, when *Max_Floor* is increased from 4 to 20, the read latency of NVLSM-F becomes 2.26× larger. However, the latency of NVLSM only becomes 1.46× larger.

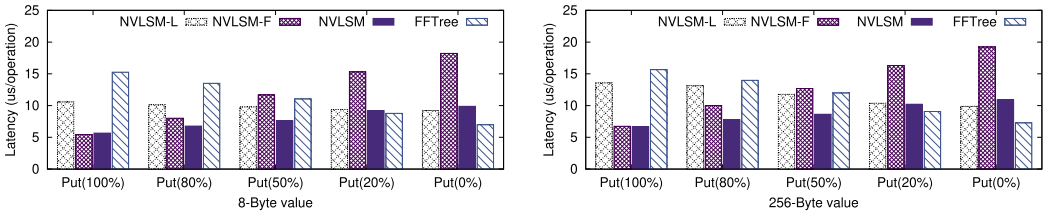


Fig. 14. Latency with different value sizes.

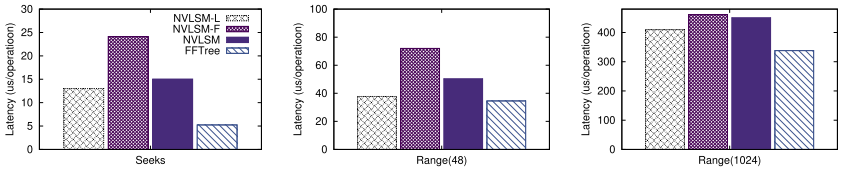


Fig. 15. Latency for range queries.

Different value sizes. In this evaluation, we study the impact on the operation latency from different value sizes. Besides 128 bytes, we increase the value size to 512 bytes. We also include a tiny value size, 8 bytes. The extremely small value can be used in some particular scenarios like the index server [6]. It can also represent the key-value separation to some degree since keys will only record the addresses of values with several bytes [44]. All workloads include 50 million random writes and 10 million random reads.

The results in Figure 14 indicate that the influences of value sizes are more significant for LSM-Tree key-value stores. When the number of key-value pairs is fixed, with larger value size, LSM-Tree key-value stores have to write more data to the device. From 8-byte values to 512-byte values, the Put(100%) latency of NVLSM-L is increased by 37.84%. NVLSM-F and NVLSM have smaller latency increases, 17.6% and 18.33%, respectively, since they have smaller write amplification. The write performance of FFTree is less impacted since the tree nodes of FFTree only store keys and locations of values. The data movements in tree nodes do not include values. Therefore, we argue that LSM-Tree key-values are more suitable for value sizes smaller than 512 bytes. If the size of the value is larger than 512 bytes, separating values from keys may significantly improve the performance.

6.3.2 Range Queries. Figure 15 shows the performance of range queries. In these evaluations, we set key/value sizes to 16/128 bytes. We firstly load 50 million key-value pairs. Then we execute 10 million Range(48) and Range(1,024), respectively. In the results, we also show the latency of the Seek operations.

Seeks. Compared with NVLSM-F, the Seek latency of NVLSM is reduced by 43% since the seeking process in NVLSM can also be accelerated by the cascading searching. NVLSM-L provides the best seeking performance among LSM-Tree key-value stores since it seeks only one sorted run in each component. The seeking latency of NVLSM is larger than that of NVLSM-L by 11.2%. NVLSM-F has the worst seek efficiency since it may search for multiple sorted runs in each component. FFTree offers the best seeking performance since a seek operation can be finished by searching the tree structure.

Small ranges. In small range queries, Range(48), the total latency is influenced by the seeking latency significantly. Thereby NVLSM provides a good performance close to NVLSM-L since the seeking process is accelerated by the cascading searching. However, NVLSM has to iterate

more sorted runs than NVLSM-L. The total latency of NVLSM for Range(48) is enlarged by 23.5% compared with NVLSM-L. However, compared with NVLSM-F, the small range query latency of NVLSM is reduced by 33.4%. NVLSM-F results in a large latency for small range queries due to its seeking overhead.

Long ranges. When executing a long-range query, the latency is dominated by the iteration process instead of the seeking process. In this case, NVLSM and NVLSM-F have a similar performance. Their latencies are larger than that of NVLSM-L by 15.3%–17.2% since they have to iterate key-value pairs among more sorted runs at the same time.

6.4 Other System Overheads

DRAM space and CPU utilization. We randomly insert 100M key-value pairs with key/value size of 16/128 bytes, and then perform 100M Get operations. We measure the DRAM space and CPU utilization for both the read and write workloads. We check the CPU utilization every 10 s. The accumulative compaction and cascading searching in NVLSM does not introduce significant CPU overheads. The differences in CPU utilization of all key-value stores are less than 10%.

LSM-Tree key-value stores need a small amount of DRAM for the write buffer and metadata. The DRAM space used by NVLSM-L, NVLSM-F, and NVLSM is about 8 MB. LevelDB and NoveLSM consume about 9 MB–10 MB DRAM. In LevelDB, each item in metadata is for an SST file, including extra file information. The size of each item in metadata of LevelDB is larger than that of NVLSM. Although PebblesDB requires a large DRAM space since it keeps bloom filters in DRAM, it can achieve a similar DRAM consumption with LevelDB by storing bloom filters on NVM.

Crash and recovery. We measure the recovery time of FFTree, LevelDB, NoveLSM, NVLSM, and PebblesDB for recovering 100M key-value pairs with key/value size of 16/128 bytes. All of them can achieve fast recovery. The recovery time is all under 1 s. Although PebblesDB can also be reopened within a second after a failure, it reconstructs bloom filters when sorted runs are accessed for the first time.

6.5 YCSB Workloads

Yahoo! Cloud Service Benchmark (**YCSB**) provides six core workloads representing different real-world scenarios [14, 72] including **L** – 100% Puts, **A** – 50% Puts, 50% Gets, **B** – 5% Puts, 95% Gets, **C** – 100% Gets, **D** – 5% Puts, 95% Gets (Latest), **E** – 5% Puts, 95% Short Ranges(100), and **F** – 50% Read-Modify-Writes, 50% Gets. In this evaluation, besides NVLSM-L, NVLSM-F, and NVLSM, we also include PebblesDB and RocksDB with a single thread. We still use key/value size of 16/128 bytes and use the same configuration in Section 6.2. We randomly load 100M key-value pairs (L) before executing each workload (A–F). We emulate NVM performance with PCM and perform 10M operations for Workloads A–F.

Figure 16 shows the evaluation results of YCSB including the latency for each workload and the total write and read sizes of NVM for all workloads. Results indicate that FFTree provides a good performance in read-intensive workloads, e.g., B and C, while LSM-Tree key-value stores achieve a small latency in write-intensive workloads.

Among all LSM-Tree key-value stores, the results of Workload L indicate NVLSM-F and PebblesDB which uses the fragmented compaction can provide the best Put performance. However, in all other workloads, the performance of NVLSM-F and PebblesDB is degraded by the long Get latency. NVLSM-L and RocksDB with leveled compaction achieve the best performance in workloads B, C, D, and E when the majority of the workloads are read operations. However, NVLSM can be tuned for read-intensive workloads. We will discuss it later.

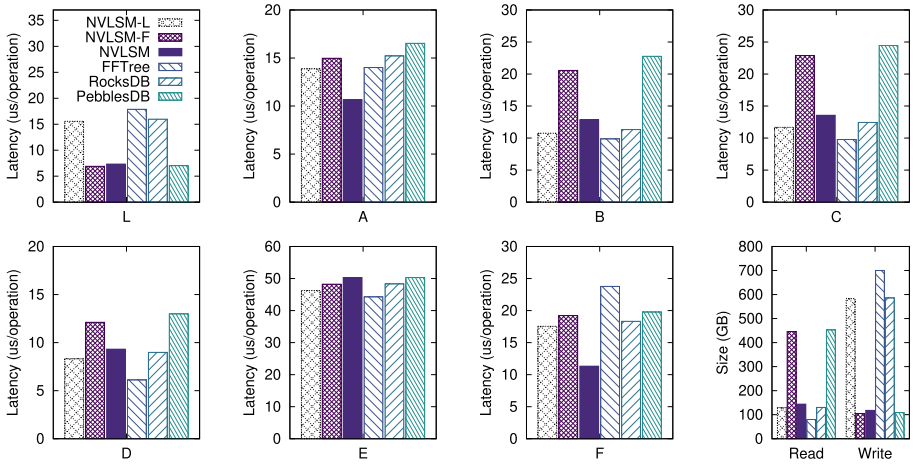


Fig. 16. Latency and Read/Write Size in YCSB Workloads.

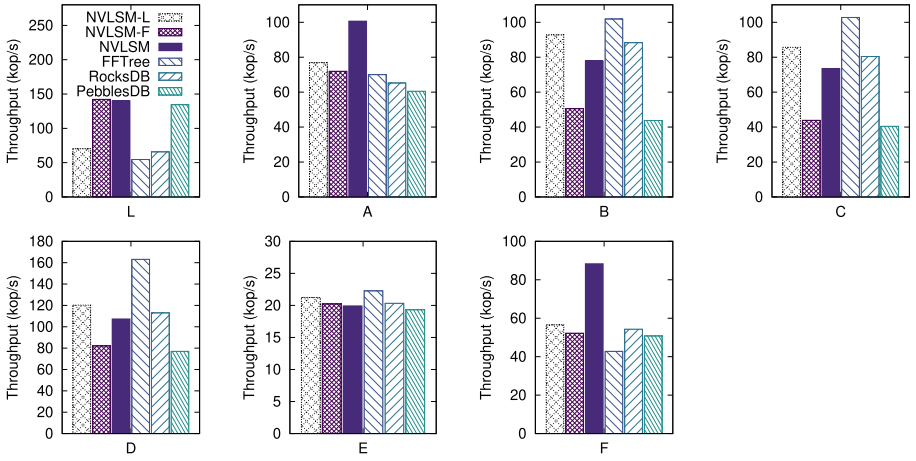


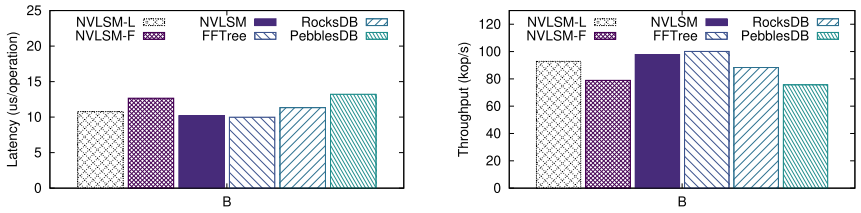
Fig. 17. Throughput in YCSB Workloads.

NVLSM achieves a Put performance similar to NVLSM-F and a Get performance comparable with NVLSM-L at the same time. In Workloads A and F, the latency of NVLSM is smaller than that of key-value stores with other compaction schemes by 48.7%–61.1%.

In the results of the total write and read sizes for all seven workloads, NVLSM is the only key-value store that reads and writes a small amount of data at the same time. Compared with NVLSM-F, NVLSM reduces the read size by 63.1%–64.5% and achieves a similar read size to that of NVLSM-L. Meanwhile, NVLSM also achieves a small write size close to that of NVLSM-F. Compared with NVLSM-L, NVLSM reduces the write size by 82.7%–83.3%.

Figure 17 shows the throughput of all key-value stores. In workloads A and F, the throughput of NVLSM is higher than that of other key-value stores by 1.26× – 1.63× and 1.49× – 2.02×, respectively.

As we discussed, by changing the number of floors in an SAT, NVLSM can be tuned for either write-intensive or read-intensive workloads. In this experiment, we take workload B (5% Puts, 95% Gets) as an example to discuss the performance with different numbers of floors in the SAT in a



(a) Latency (b) Throughput

Fig. 18. YCSB-B with max_floor = 2.

read-dominated workload. We use the same configuration in Figure 16 and decrease the number of floors in SAT to two. In addition, for NVLSM-F and PebblesDB using fragmented compaction, we also set the maximal number of SSTs in the same guard to two.

Figure 18 shows the evaluation results of the YCSB B workload. The results indicate that when we set max_floor to two, NVLSM can provide the best performance among all LSM-Base key-value stores in read-dominated workloads, although the performance differences among all schemes are small. The performance of NVLSM is comparable to that of FFTree.

7 CONCLUSION

We propose NVLSM using LSM-Tree with a new accumulative compaction scheme. Accumulative compaction fully utilizes byte-addressability of NVM to build floors in SATs when migrating sorted runs between components, and performs an efficient cascading search among floors. Our evaluations show that NVLSM reduces write amplification without significantly increasing read amplification. In our future work, we would like to improve the current design by involving both NVM and storage.

REFERENCES

[1] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. BzTree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment* 11, 5 (2018), 553–565.

[2] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. 2015. Let's talk about storage and recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 707–722.

[3] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-behind logging. *Proceedings of the VLDB Endowment* 10, 4 (2016), 337–348.

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 40. ACM, 53–64.

[5] Katelin Bailey, Luis Ceze, Steven D. Gribble, and Henry M. Levy. 2011. Operating system implications of fast, cheap, non-volatile memory. In *HotOS*, Vol. 13. 2–2.

[6] Zhichao Cao, Shiyong Liu, Fenggang Wu, Guohua Wang, Bingzhe Li, and David H. C. Du. 2019. Sliding look-back window assisted data chunk rewriting for improving deduplication restore performance. In *17th USENIX Conference on File and Storage Technologies (FAST'19)*. USENIX Association, Boston, MA, 129–142. https://www.usenix.org/conference/fast19/presentation/cao

[7] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging locks for non-volatile memory consistency. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 433–452.

[8] Helen H. W. Chan, Chieh-Jan Mike Liang, Yongkun Li, Wenjia He, Patrick P. C. Lee, Lianjie Zhu, Yaozu Dong, Yinlong Xu, Yu Xu, Jin Jiang, et al. 2018. HashKV: Enabling efficient updates in KV storage via hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC'18)*. 1007–1019.

[9] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. 2015. Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. *Proceedings of the VLDB Endowment* 8, 5 (2015), 497–508.

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

[11] Cheng Chen, Jun Yang, Qingsong Wei, Chundong Wang, and Mingdi Xue. 2016. Fine-grained metadata journaling on NVM. In *2016 32nd Symposium on Mass Storage Systems and Technologies (MSST'16)*. IEEE, 1–13.
[12] Shimin Chen and Qin Jin. 2015. Persistent B$^+$-trees in non-volatile main memory. *Proceedings of the VLDB Endowment* 8, 7 (2015), 786–797.
[13] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. 1999. Cache-conscious structure definition. In *ACM SIGPLAN Notices*, Vol. 34. ACM, 13–24.
[14] 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*. ACM, 143–154.
[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*. ACM, 79–94.
[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*. ACM, 505–520.
[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.
[18] 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 *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 205–220.
[19] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing space amplification in rocksDB. In *CIDR*, Vol. 3. 3.
[20] Facebook. 2015. RocksDB. (2015). Retrieved on 09 June, 2019 https://rocksdb.org/.
[21] Alexandre P. Ferreira, Miao Zhou, Santiago Bock, Bruce Childers, Rami Melhem, and Daniel Mossé. 2010. Increasing PCM main memory lifetime. In *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 914–919.
[22] Sanjay Ghemawat and Jeff Dean. 2011. LevelDB. Retrieved on 13 Aug., 2019 https://github.com/google/leveldb.
[23] Google. 2017. LevelDB Benchmarks. Retrieved on 13 Aug., 2019 http://www.lmdb.tech/bench/microbench/benchmark.html.
[24] Frank T. Hady, Annie Foong, Bryan Veal, and Dan Williams. 2017. Platform storage performance with 3D XPoint technology. *Proceedings of the IEEE* 105, 9 (2017), 1822–1833.
[25] Nadav Har'El. 2017. Scylla Compaction Strategies Series: Space Amplification in Size-Tiered Compaction. Retrieved on 16 Aug., 2019 https://www.scylladb.com/2018/01/17/compaction-series-space-amplification/.
[26] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand S. Aiyer, Liyin Tang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. Analysis of HDFS under HBase: A facebook messages case study. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*, Vol. 14.
[27] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable transient inconsistency in byte-addressable persistent B+-tree. In *16th USENIX Conference on File and Storage Technologies*. 187.
[28] HyperDex. 2016. HyperLevelDB: A fork of LevelDB intended to meet the needs of HyperDex while remaining compatible with LevelDB. (2016).
[29] Intel. 2018. https://pmem.io/libpmemobj-cpp/. Retrieved on 20 August, 2019 from https://pmem.io/libpmemobj-cpp/.
[30] Intel. 2018. pmemkv. Retrieved on 20 August, 2019 from https://github.com/pmem/pmemkv.
[31] Intel. 2016. Persistent Memory Development Kit. (2016). Retrieved on 20 August, 2019 from http://pmem.io/pmdk/libpmemobj/.
[32] Intel. 2017. Transactions in Persistent Memory Development Kits. (2017). Retrieved on 20 August, 2019 from http://pmem.io/2016/05/25/cpp-07.html.
[33] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-atomic persistent memory updates via JUSTDO logging. *ACM SIGARCH Computer Architecture News* 44, 2 (2016), 427–442.
[34] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. 2019. SLM-DB: Single-level key-value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST'19)*. USENIX Association, Boston, MA, 191–205. https://www.usenix.org/conference/fast19/presentation/kaiyrakhmet
[35] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for nonvolatile memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC'18)*. 993–1005.
[36] Bradley C. Kuszmaul. 2014. A comparison of fractal trees to log-structured merge (LSM) trees. *Tokutek White Paper*
[37] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.
[38] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. *ACM SIGARCH Computer Architecture News* 37, 3 (2009), 2–13.

[39] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. 2017. WORT: Write optimal radix tree for persistent memory storage systems. In *FAST*. 257–270.
[40] J. Li, A. Pavlo, and S. Dong. 2017. NVMRocks: RocksDB on non-volatile memory systems. Retrieved on 03 June, 2019 https://github.com/pmem/pmem-rocksdb.
[41] Yinan Li, Bingsheng He, Qiong Luo, and Ke Yi. 2009. Tree indexing on flash disks. In *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 1303–1306.
[42] linux. 2017. clflush. (2017). Retrieved on 15 July, 2019 from https://www.felixcloutier.com/x86/CLFLUSH.html.
[43] linux. 2017. mfence. (2017). Retrieved on 15 July, 2019 from https://www.felixcloutier.com/x86/MFENCE.html.
[44] 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 (TOS)* 13 (2017), 5.
[45] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. 2017. Improving performance and endurance of persistent memory with loose-ordering consistency. *IEEE Transactions on Parallel and Distributed Systems* (2017).
[46] Paul E. McKenney. 2005. Memory ordering in modern microprocessors, part I. *Linux Journal* 2005, 136 (2005), 2.
[47] Fei Mei, Qiang Cao, Hong Jiang, and Jingjun Li. 2018. SifrDB: A unified solution for write-optimized key-value stores in large datacenter. *ACM SoCC* (2018), 477–489.
[48] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. 2017. Atomic in-place updates for non-volatile main memories with kamino-tx. In *Proceedings of the 12th European Conference on Computer Systems*. ACM, 499–512.
[49] Sparsh Mittal and Jeffrey S. Vetter. 2016. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel and Distributed Systems* 27, 5 (2016), 1537–1550.
[50] Jeffrey C. Mogul, Eduardo Argollo, Mehul Shah, and Paolo Faraboschi. 2009. Operating system support for NVM hybrid main memory.
[51] Iulian Moraru, David G. Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. 2013. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the 1st ACM SIGOPS Conference on Timely Results in Operating Systems*. ACM, 1.
[52] Chris Mumford. 2011. LevelDB Implementations. Retrieved on 26 Aug., 2019 from https://github.com/google/leveldb/blob/master/doc/impl.md.
[53] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-optimized dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST'19)*. 31–44.
[54] Songjie Niu and Shimin Chen. 2015. Optimizing CPU cache performance for Pregel-like graph computation. In *2015 31st IEEE International Conference on Data Engineering Workshops (ICDEW'15)*. IEEE, 149–154.
[55] 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*. ACM, 371–386.
[56] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.
[57] William Pugh. 1990. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM* 33, 6 (1990), 668–677.
[58] 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*. ACM, 497–514.
[59] Simone Raoux, Feng Xiong, Matthias Wuttig, and Eric Pop. 2014. Phase change materials and phase change memory. *MRS Bulletin* 39, 8 (2014), 703–710.
[60] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. 2015. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 672–685.
[61] 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.
[62] 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*. ACM, 217–228.
[63] Clinton W. Smullen, Vidyabhushan Mohan, Anurag Nigam, Sudhanva Gurumurthi, and Mircea R. Stan. 2011. Relaxing non-volatility for fast and energy-efficient STT-RAM caches. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11)*. IEEE, 50–61.
[64] John Tse and Alan Jay Smith. 1998. CPU cache prefetching: Timing evaluation of hardware implementations. *IEEE Transactions on Computers* 47, 5 (1998), 509–526.

[65] Dean M. Tullsen and Susan J. Eggers. 1993. Limitations of cache prefetching on a bus-based multiprocessor. In *ACM SIGARCH Computer Architecture News*, Vol. 21. ACM, 278–288.
[66] Jin Wang, Yong Zhang, Yang Gao, and Chunxiao Xing. 2013. PLSM: A highly efficient LSM-tree index supporting real-time big data analysis. In *2013 IEEE 37th Annual Computer Software and Applications Conference (COMPSAC'13)*. IEEE, 240–245.
[67] Wei Wei, Dejun Jiang, Jin Xiong, and Mingyu Chen. 2017. HAP: Hybrid-memory-aware partition in shared last-level cache. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 3 (2017), 24.
[68] Dan Williams. 2018. Persistent Memory. Retrieved on 15 Aug, 2019 from https://nvdimm.wiki.kernel.org/.
[69] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based ultra-large key-value store for small data. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*. USENIX Association, 71–82.
[70] Jian Xu and Steven Swanson. 2016. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *FAST*. 323–338.
[71] Maysam Yabandeh. 2018. Compaction in RocksDB. (2018). Retrieved on 03 Aug., 2019 from https://github.com/facebook/rocksdb/wiki/Compaction.
[72] Yahoo. 2010. Core Workloads in YCSB. Retrieved on 15 Nov., 2019 from https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads.
[73] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *FAST*, Vol. 15. 167–181.
[74] Ting Yao, Jiguang Wan, Ping Huang, Xubin He, Qingxin Gui, Fei Wu, and Changsheng Xie. 2017. A light-weight compaction tree to reduce I/O amplification toward efficient key-value stores. In *Proceedings of the 33rd International Conference on Massive Storage Systems and Technology (MSST'17)*.
[75] Yinliang Yue, Bqingsheng He, Yuzhe Li, and Weiping Wang. 2017. Building an efficient put-intensive key-value store with skip-tree. *IEEE Transactions on Parallel and Distributed Systems* 28, 4 (2017), 961–973.
[76] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. Mojim: A reliable and highly-available non-volatile memory system. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 3–18.
[77] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 461–476.

Received May 2020; revised February 2021; accepted March 2021