SLM-DB: Single-Level Key-Value Store with Persistent Memory

Olzhas Kaiyrakhmet and Songyi Lee, *UNIST;* Beomseok Nam, *Sungkyunkwan University;* Sam H. Noh and Young-ri Choi, *UNIST*

https://www.usenix.org/conference/fast19/presentation/kaiyrakhmet

This paper is included in the Proceedings of the
17th USENIX Conference on File and Storage Technologies (FAST '19).
February 25–28, 2019 • Boston, MA, USA
978-1-931971-48-5

SLM-DB: Single-Level Key-Value Store with Persistent Memory

Olzhas Kaiyrakhmet, Songyi Lee
UNIST

Beomseok Nam
Sungkyunkwan University

Sam H. Noh, Young-ri Choi
UNIST

Abstract

This paper investigates how to leverage emerging byte-addressable persistent memory (PM) to enhance the performance of key-value (KV) stores. We present a novel KV store, the *Single-Level Merge DB* (SLM-DB), which takes advantage of both the B+-tree index and the Log-Structured Merge Trees (LSM-tree) approach by making the best use of fast persistent memory. Our proposed SLM-DB achieves high read performance as well as high write performance with low write amplification and near-optimal read amplification. In SLM-DB, we exploit persistent memory to maintain a B+-tree index and adopt an LSM-tree approach to stage inserted KV pairs in a PM resident memory buffer. SLM-DB has a single-level organization of KV pairs on disks and performs selective compaction for the KV pairs, collecting garbage and keeping the KV pairs sorted sufficiently for range query operations. Our extensive experimental study demonstrates that, in our default setup, compared to LevelDB, SLM-DB provides 1.07 - 1.96 and 1.56 - 2.22 times higher read and write throughput, respectively, as well as comparable range query performance.

1 Introduction

Key-value (KV) stores have become a critical component to effectively support diverse data intensive applications such as web indexing [15], social networking [8], e-commerce [18], and cloud photo storage [12]. Two typical types of KV stores, one based on B-trees and the other based on Log-Structured Merge Trees (LSM-tree) have been popularly used. B-tree based KV stores and databases such as KyotoCabinet [2] support fast read (i.e., point query) and range query operations. However, B-tree based KV stores show poor write performance as they incur multiple small random writes to the disk and also suffer from high write amplification due to dynamically maintaining a balanced structure [35]. Thus, they are more suitable for read-intensive workloads.

LSM-tree based KV stores such as BigTable [15], LevelDB [3], RocksDB [8] and Cassandra [28] are optimized to efficiently support write intensive workloads. A KV store based on an LSM-tree can benefit from high write throughput that is achieved by buffering keys and values in memory and sequentially writing them as a batch to a disk. However, it has challenging issues of high write and read amplifications and slow read performance because an LSM-tree is organized with multiple levels of files where usually KV pairs are merge-sorted (i.e., compacted) multiple times to enable fast search.

Recently, there has been a growing demand for data intensive applications that require high performance for both read and write operations [14, 40]. Yahoo! has reported that the trend in their typical workloads has changed to have similar proportions of reads and writes [36]. Therefore, it is important to have optimized KV stores for both read and write workloads.

Byte-addressable, nonvolatile memories such as phase change memory (PCM) [39], spin transfer torque MRAM [21], and 3D XPoint [1] have opened up new opportunities to improve the performance of memory and storage systems. It is projected that such persistent memories (PMs) will have read latency comparable to that of DRAM, higher write latency (up to 5 times) and lower bandwidth (5∼10 times) compared to DRAM [19, 23, 24, 27, 42]. PM will have a large capacity with a higher density than DRAM. However, PM is expected to coexist with disks such as HDDs and SSDs [23, 25]. In particular, for large-scale KV stores, data will still be stored on disks, while the new persistent memories will be used to improve the performance [4, 20, 23]. In light of this, there have been earlier efforts to redesign an LSM-tree based KV store for PM systems [4, 23]. However, searching for a new design for KV stores based on a hybrid system of PM and disks, in which PM carries a role that is more than just a large memory write buffer or read cache, is also essential to achieve even better performance.

In this paper, we investigate how to leverage PM to enhance the performance of KV stores. We present a novel KV store, the *Single-Level Merge DB* (SLM-DB), which takes advantage of both the B+-tree index and the LSM-tree approach by making the best use of fast PM. Our proposed SLM-DB achieves high read performance as well as high write performance with low write amplification and near-optimal read amplification. In SLM-DB, we exploit PM to maintain a B+-tree for indexing KVs. Using the persistent B+-tree index, we can accelerate the search of a key (without depending on Bloom filters). To maintain high write throughput, we adopt the LSM-tree approach to stage inserted KV pairs in a PM resident memory buffer. As an inserted KV pair is persisted immediately in the PM buffer, we can also eliminate the write ahead log completely while providing strong data durability.

In SLM-DB, KV pairs are stored on disks with a *single-level* organization. Since SLM-DB can utilize the B+-tree for searches, it has no need to keep the KV pairs in sorted order, which significantly reduces write amplification. However, obsolete KV pairs should be garbage collected. Moreover, SLM-DB needs to provide some degree of sequentiality of KV pairs stored on disks in order to provide reasonable performance for range queries. Thus, the selective compaction scheme, which only performs restricted *merge* of the KV pairs organized in the single level, is devised for SLM-DB.

The main contributions of this work are as follows:

- We design a single-level KV store that retains the benefit of high write throughput from the LSM-tree approach and integrate it with a persistent B+-tree for indexing KV pairs. In addition, we employ a PM resident memory buffer to eliminate disk writes of recently inserted KV pairs to the write ahead log.

- For selective compaction, we devise three compaction candidate selection schemes based on 1) the live-key ratio of a data file, 2) the leaf node scans in the B+-tree, and 3) the degree of sequentiality per range query.

- We implement SLM-DB based on LevelDB and also integrate it with a persistent B+-tree implementation [22]. SLM-DB is designed such that it can keep the B+-tree and the single-level LSM-tree consistent on system failures, providing strong crash consistency and durability guarantees. We evaluate SLM-DB using the db_bench microbenchmarks [3] and the YCSB [17] for real world workloads. Our extensive experimental study demonstrates that in our default setup, compared to LevelDB, SLM-DB provides up to 1.96 and 2.22 times higher read and write throughput, respectively, and shows comparable range query performance, while it incurs only 39% of LevelDB's disk writes on average.

The rest of this paper is organized as follows. Section 2 discusses LSM-trees with the issue of slow read performance and high read/write amplification and also discusses PM technologies for KV stores. Section 3 presents the design and implementation of SLM-DB, Section 4 discusses how KV store operations are implemented in SLM-DB, and Section 5 discusses the recovery of SLM-DB on system failures. Section 6 evaluates the performance of SLM-DB and presents our experimental results, and Section 7 discusses issues of PM cost and parallelism. Section 8 discusses related work, and finally Section 9 concludes the paper.

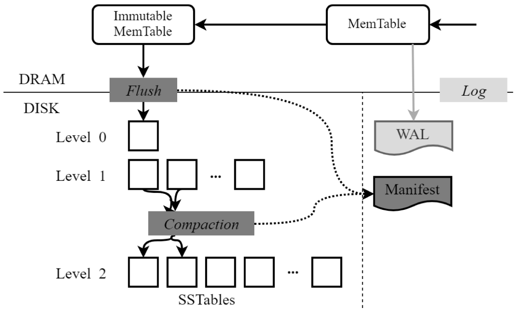


Figure 1: LevelDB architecture.

2 Background and Motivation

In this section, we first discuss an LSM-tree based KV store and its challenging issues by focusing on LevelDB [3]. Other LSM-tree based KV stores such as RocksDB [8] are similarly structured and have similar issues. We then discuss considerations for using PM for a KV store.

2.1 LevelDB

LevelDB is a widely used KV store inspired by Google's Bigtable [15], which implements the Log-Structured Merge-tree (LSM-tree) [33]. LevelDB supports basic KV store operations of *put*, which adds a KV pair to a KV store, *get*, which returns the associated value for a queried key, and *range query*, which returns all KV pairs within a queried range of keys by using *iterators* that scan all KV pairs. In LevelDB's implementation, the LSM-tree has two main modules, *MemTable* and *Immutable MemTable* that reside in DRAM and multiple levels of *Sorted String Table* (SSTable) files that reside in persistent storage (i.e., disks), as shown in Figure 1.

The memory components of MemTable and Immutable MemTable are basically sorted skiplists. MemTable buffers newly inserted KV pairs. Once MemTable becomes full, LevelDB makes it an Immutable MemTable and creates a new MemTable. Using a background thread, it flushes recently inserted KV pairs in the Immutable MemTable to the disk as an on-disk data structure SSTable where sorted KV pairs are stored. Note that the deletion of a KV pair is treated as an update as it places a deletion marker.

Table 1: Locating overhead breakdown (in microseconds) of a read operation in LevelDB

KV store	File search	Block search	Bloom filter	Unnecessary block read
LevelDB w/o BF	1.28	19.99	0	40.62
LevelDB w BF	1.33	19.38	7.22	13.58
SLM-DB	1.32		0	0

During the above insertion process, for the purpose of recovery from a system crash, a new KV pair must first be appended to the write ahead log (WAL) before it is added to MemTable. After KV pairs in Immutable MemTable are finally dumped into the disk, the log is deleted. However, by default, LevelDB does not commit KV pairs to the log due to the slower write performance induced by the `fsync()` operations for commit. In our experiments when using a database created by inserting 8GB data with a 1KB value size, the write performance drops by more than 12 times when `fsync()` is enabled for WAL. Hence, it trades off durability and consistency against higher performance.

For the disk component, LevelDB is organized as multiple levels, from the lowest level L_0 to the highest level L_k. Each level, except L_0, has one or more sorted SSTable files in which key ranges of the files in the same level do not overlap. Each level has limited capacity, but a higher level can contain more SSTable files such that the capacity of a level is generally around 10 times larger than that of its previous level.

To maintain such hierarchical levels, when the size of a level L_x grows beyond its limit, a background *compaction* thread selects one SSTable file in L_x. It then moves the KV pairs in that file to the next level L_{x+1} by performing a *merge sort* with all the SSTable files that overlap in level L_{x+1}. When KV pairs are being sorted, if the same key exists, the value in L_{x+1} is overwritten by that in L_x, because the lower level always has the newer value. In this way, it is guaranteed that keys stored in SSTable files in one level are unique. Compaction to L_0 (i.e., flushing recently inserted data in Immutable MemTable to L_0) does not perform merge sort in order to increase write throughput, and thus, SSTables in L_0 can have overlapped key ranges. In summary, using compaction, LevelDB not only keeps SSTable files in the same level sorted to facilitate fast search, but it also collects garbage in the files.

LevelDB also maintains the SSTable metadata of the current LSM-tree organization in a file referred to as MANIFEST. The metadata includes a list of SSTable files in each level and a key range of each SSTable file. During the compaction process, changes in the SSTable metadata such as a deleted SSTable file are captured. Once the compaction is done, the change is first logged in the MANIFEST file, and then obsolete SSTable files are deleted. In this way, when the system crashes even during compaction, LevelDB can return to a consistent KV store after recovery.

2.2 Limitations of LevelDB

Slow read operations For read operations (i.e., point queries), LevelDB first searches a key in MemTable and then Immutable MemTable. If it fails to find the key in the memory components, it searches the key in each level from the lowest one to the highest one. For each level, LevelDB needs to first find an SSTable file that may contain the key by a binary search based on the starting keys of SSTable files in that level. When such an SSTable file is identified, it performs another binary search on the SSTable file index, which stores the information about the first key of each 4KB data block in the file. Thus, a read operation requires at least two block reads, one for the index block and the other for the data block. However, when the data block does not include that key, LevelDB needs to check the next level again, until it finds the key or it reaches the highest level. To avoid unnecessary block reads and reduce the search cost, LevelDB uses a Bloom filter for each block.

Table 1 presents the overhead breakdown (in microseconds) for locating a KV pair in LevelDB with and without a Bloom filter for a random read operation. For the results, we measure the read latency of LevelDB for the random read benchmark in db_bench [3] (which are microbenchmarks built in LevelDB). In the experiments, we use 4GB DRAM and run a random read workload right after creating a database by inserting 20GB data with a 1KB value size (without waiting for the compaction process to finish, which is different from the experiments in Section 6.3). The details of our experimental setup are discussed in Section 6.

The locating overhead per read operation includes time spent to search an SSTable file that contains the key (i.e., "File search"), to find a block in which the key and its corresponding value are stored within the file (i.e., "Block search"), and to check the Bloom filter (BF). Moreover, included in the locating overhead is time for, what we refer to as the "Unnecessary block read". Specifically, this refers to the time to read blocks unnecessarily due to the multi-level search based on the SSTable index where BF is not used and, where BF is used, the time to read the false positive blocks. As shown in the table, with our proposed SLM-DB using the B+-tree index, which we discuss in detail later, the locating overhead can be significantly reduced to become almost negligible. In the above experiment, the average time of reading and processing a data block for the three KV stores is 682 microseconds.

Figure 2 shows the overhead for locating a KV pair for a random read operation over varying value sizes. The figure shows the ratio of the locating overhead to the total read operation latency. We observe that the overhead increases as the size of the value increases. When a Bloom filter is used in LevelDB, the locating overhead becomes relatively smaller, but the overhead with a Bloom filter is still as high as up to 36.66%. In the experiment, the random read workload is exe-

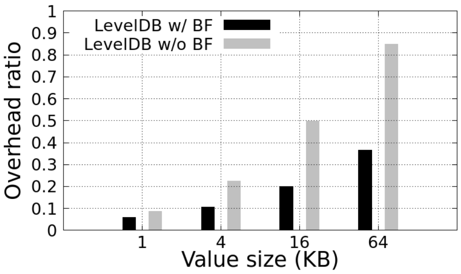


Figure 2: Overhead for locating a KV pair for different size of values as a fraction of the read operation latency.

cuted while the compaction of SSTable files from the lowest to the highest levels is in progress. With a larger value size, the number of files in multiple levels that LevelDB needs to check to see if a given key exists, and consequently the number of queries to a Bloom filter for the key, increases. Therefore, for a 64KB value size, LevelDB with a Bloom filter shows 6.14 times higher overhead largely incurred by unnecessary block reads, compared to a 1KB value size.

High write and read amplification Well-known issues of any LSM-tree based KV store are high write and read amplification [30, 31, 35, 40]. It maintains hierarchical levels of sorted files on a disk while leveraging sequential writes to the disk. Therefore, an inserted KV needs to be continuously merge-sorted and written to the disk, moving toward the highest level, via background compaction processes. For an LSM-tree structure with level k, the write amplification ratio, which is defined as the ratio between the total amount of data written to disk and the amount of data requested by the user, can be higher than $10 \times k$ [30, 31, 40].

The read amplification ratio, which is similarly defined as the ratio between the total amount of data read from a disk and the amount of data requested by the user, is high by nature in an LSM-tree structure. As discussed above, the cost of a read operation is high. This is because LevelDB may need to check multiple levels for a given key. Moreover, to find the key in an SSTable file at a level, it not only reads a data block but also an index block and a Bloom filter block, which can be much larger than the size of the KV pair [30].

2.3 Persistent Memory

Emerging persistent memory (PM) such as phase change memory (PCM) [39], spin transfer torque MRAM [21], and 3D XPoint [1] is byte-addressable and nonvolatile. PM will be connected via the memory bus rather than the block interface and thus, the failure atomicity unit (or granularity) for write to PM is generally expected to be 8 bytes [29, 42]. When persisting a data structure in PM, which has a smaller failure atomicity unit compared to traditional storage devices, we must ensure that the data structure remains consistent even when the system crashes. Thus, we need to carefully update or change the data structure by ensuring the memory write ordering.

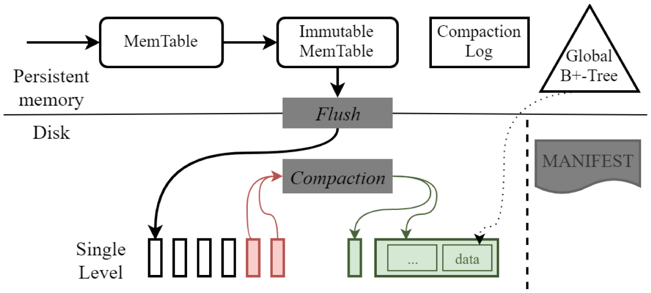


Figure 3: SLM-DB architecture.

However, in modern processors, memory write operations may be reordered in cache line units to maximize the memory bandwidth. In order to have ordered memory writes, we need to explicitly make use of expensive memory fence and cache flush instructions (`CLFLUSH` and `MFENCE` in Intel x86 architecture) [16, 22, 23, 29, 34, 42]. Moreover, if the size of the data written to PM is larger than 8 bytes, the data structure can be partially updated in system failure, resulting in an inconsistent state after recovery. In this case, it is necessary to use well-known techniques like logging and Copy-on-Write (CoW). Thus, a careful design is required for data structures persisted in PM.

PM opens new opportunities to overcome the shortcomings of existing KV stores. There has been a growing interest to utilize PM for KV stores [4, 23, 41]. LSM-tree based KV stores have been redesigned for PM [4, 23]. However, it is also important to explore new designs for PM based KV stores. In this work, we investigate a design of KV stores, which employs an index persisted in PM.

3 Single-Level Merge DB (SLM-DB)

This section presents the design and implementation of our Single-Level Merge DB (SLM-DB). Figure 3 shows the overall system architecture of SLM-DB. SLM-DB leverages PM to store MemTable and Immutable MemTable. The persistent MemTable and Immutable MemTable allow us to eliminate the write ahead log (WAL), providing stronger durability and consistency upon system failures. SLM-DB is organized as a single level L_0 of SSTable files, unlike LevelDB, hence the name Single Level Merge DB (SLM-DB). Thus, SLM-DB does not rewrite KV pairs stored on disks to merge them with pairs in the lower level, which can occur multiple times. Having the persistent memory component and single-level disk component, write amplification can be reduced significantly.

To expedite read operations on a single-level organization of SSTable files, SLM-DB constructs a persistent B+-tree index. Since the B+-tree index is used to search a KV pair stored on a disk, there is no need to fully sort KV pairs in the level, in contrast with most LSM-tree based KV stores.

Algorithm 1 Insert(*key*, *value*, *prevNode*)

```
$curNode$ := NewNode($key$, $value$);
$curNode.next$ := $prevNode.next$;
mfence();
clflush($curNode$);
mfence();
$prevNode.next$ := $curNode$;
mfence();
clflush($prevNode.next$);
mfence();
```

However, obsolete KV pairs in the SSTable files that have been updated by fresh values should be deleted to avoid disk space waste. Moreover, SLM-DB needs to maintain a sufficient level of sequentiality of KVs (i.e., a degree of how well KV pairs are stored in sorted order) in SSTables so that it can provide reasonable range query performance. Therefore, a selective compaction scheme, which selectively merges SSTables, is integrated with SLM-DB. Also, to keep the B+-tree and (single-level) LSM-tree consistent on system failures, the state of on-going compaction needs to be backed up by the compaction log stored in PM.

We implement SLM-DB based on LevelDB (version 1.20). We inherit the memory component implementation of MemTable and Immutable MemTable with modifications to persist them in PM. We keep the on-disk data structure of SSTable and the file format as well as the multiple SSTable file compaction (i.e., merge-sort) scheme. We also utilize the LSM-tree index structure, which maintains a list of valid SSTable files and SSTable file metadata, the mechanism to log any change in the LSM-tree structure to the MANIFEST file, and the recovery scheme of LevelDB. We completely change the random read and range query operations of the LevelDB implementation using a persistent B+-tree.

Among the many persistent B-tree implementations, we use the FAST and FAIR B-tree [22][1] for SLM-DB. Particularly, FAST and FAIR B-tree was shown to outperform other state-of-the-art persistent B-trees in terms of range query performance because it keeps all keys in a sorted order. It also yields the highest write throughput by leveraging the memory level parallelism and the ordering constraints of dependent store instructions.

3.1 Persistent MemTable

In SLM-DB, MemTable is a persistent skiplist. Note that a persistent skiplist has been discussed in previous studies [22, 23]. Skiplist operations such as insertion, update, and deletion can be done using an atomic 8-byte write operation. Algorithm 1 shows the insertion process to the lowest level of a skiplist. To guarantee the consistency of KVs in MemTable, we first persist a new node where its next pointer is set by calling memory fence and cacheline flush instructions. We then update the next pointer, which is 8 bytes, of its previous node and persist the change. Updating an existing KV pair in MemTable is done in a similar way, without in-place update of a value (similar to LevelDB's MemTable update operation). By having the PM resident MemTable, SLM-DB has no need to depend on WAL for data durability. Similarly, no consistency guarantee mechanism is required for higher levels of a skiplist as they can be reconstructed easily from the lowest level upon system failures.

3.2 B+-tree Index in PM

To speed up the search of a KV pair stored in SSTables, SLM-DB employs a B+-tree index. When flushing a KV pair in Immutable MemTable to an SSTable, the key is inserted in the B+-tree. The key is added to a leaf node of the B+-tree with a pointer that points to a PM object that contains the location information about where this KV pair is stored on the disk. The location information for the key includes an SSTable file ID, a block offset within the file, and the size of the block.

If a key already exists in the B+-tree (i.e., update), a fresh value for the key is written to a new SSTable. Therefore, a new location object is created for the key, and its associated pointer in the B+-tree leaf node is updated to point to the new PM object in a failure-atomic manner. If a deletion marker for a key is inserted, the key is deleted from the B+-tree. Persistent memory allocation and deallocation for location objects is managed by a persistent memory manager such as PMDK [5], and obsolete location objects will be garbage collected by the manager. Note that SLM-DB supports string-type keys like LevelDB does and that the string-type key is converted to an integer key when it is added to the B+-tree.

Building a B+-tree In SLM-DB, when Immutable MemTable is flushed to L_0, KV pairs in Immutable MemTable are inserted in the B+-tree. For a flush operation, SLM-DB creates two background threads, one for file creation and the other for B+-tree insertion.

In the file creation thread, SLM-DB creates a new SSTable file and writes KV pairs from Immutable MemTable to the file. Once the file creation thread flushes the file to the disk, it adds all the KV pairs stored on the newly created file to a queue, which is created by a B+-tree insertion thread. The B+-tree insertion thread processes the KV pairs in the queue one by one by inserting them in the B+-tree. Once the queue becomes empty, the insertion thread is done. Then, the change of the LSM-tree organization (i.e., SSTable metadata) is appended to the MANIFEST file as a log. Finally, SLM-DB deletes Immutable MemTable.

Scanning a B+-tree SLM-DB provides an iterator, which can be used to scan all the keys in the KV store, in a way similar to LevelDB. Iterators support `seek`, `value`, and `next` methods. The `seek(k)` method positions an iterator in the KV store such that the iterator points to key `k` or the smallest

[1] Source codes are available at https://github.com/DICL/FAST_FAIR

key larger than k if k does not exist. The `next()` method moves the iterator to the next key in the KV store and the `value()` method returns the value of the key, currently pointed to by the iterator.

In SLM-DB, a B+-tree iterator is implemented to scan keys stored in SSTable files. For the `seek(k)` method, SLM-DB searches key k in the B+-tree to position the iterator. In the FAST+FAIR B+-tree, keys are sorted in leaf nodes, and leaf nodes have a sibling pointer. Thus, if k does not exist, it can easily find the smallest key that is larger than k. Also, the `next()` method is easily supported by moving the iterator to the next key in B+-tree leaf nodes. For the `value()` method, the iterator finds the location information for the key and reads the KV pair from the SSTable.

3.3 Selective Compaction

SLM-DB supports a selective compaction operation in order to collect garbage of obsolete KVs and improve the sequentiality of KVs in SSTables. For selective compaction, SLM-DB maintains a *compaction candidate list* of SSTables. A background compaction thread is basically scheduled when some changes occur in the organization of SSTable files (for example, by a flush operation), and there are a large number of seeks to a certain SSTable (similar to LevelDB). In SLM-DB, it is also scheduled when the number of SSTables in the compaction candidate list is larger than a certain threshold. When a compaction thread is executed, SLM-DB chooses a subset of SSTables from the candidate list as follows. For each SSTable s in the list, we compute the overlapping ratio of key ranges between s and each t of the other SSTables in the list as $\frac{MIN(s_p,t_q)-MAX(s_1,t_1)}{MAX(s_p,t_q)-MIN(s_1,t_1)}$, where the key ranges of s and t are $[s_1,\ldots s_p]$ and $[t_1,\ldots t_q]$, respectively. Note that if the computed ratio is negative, then s and t do not overlap and the ratio is set to zero. We compute the total sum of overlapping ratios for s. We then decide to compact an SSTable s' with the maximum overlapping ratio value with SSTables in the list, whose key ranges are overlapped with s'. Note that we limit the number of SSTables that are simultaneously merged so as not to severely disturb foreground user operations.

The compaction process is done by using the two threads for file creation and B+-insertion described above. However, when merging multiple SSTable files, we need to check if each KV pair in the files is valid or obsolete, which is done by searching the key in the B+-tree. If it is valid, we merge-sort it with the other valid KV pairs. If the key does not exist in the B+-tree or the key is currently stored in some other SSTable, we drop that obsolete KV pair from merging to a new file.

During compaction, the file creation thread needs to create multiple SSTable files unlike flushing Immutable MemTable to L_0. The file creation thread creates a new SSTable file (of a fixed size) as it merge-sorts the files and flushes the new file to disk. It then adds all the KV pairs included in the new file to the queue of the B+-tree insertion thread. The file creation thread starts to create another new file, while the insertion thread concurrently updates the B+-tree for each KV pair in the queue. This process continues until the creation of merge-sorted files for compaction is completed. Finally, after the B+-tree updates for KV pairs in the newly created SSTable files are done, the change of SSTable metadata is committed to the MANIFEST file and the obsolete SSTable files are deleted. Note that SLM-DB is implemented such that B+-tree insertion requests for KVs in the new SSTable file are immediately queued right after file creation, and they are handled in order. In this way, when updating the B+-tree for a KV in the queue, there is no need to check the validity of the KV again.

To select candidate SSTables for compaction, SLM-DB implements three selection schemes based on *the live-key ratio* of an SSTable, *the leaf node scans* in the B+-tree, and *the degree of sequentiality per range query*. For the selection based on the live-key ratio of an SSTable, we maintain the ratio of valid KV pairs to all KV pairs (including obsolete ones) stored in each SSTable. If the ratio for an SSTable is lower than the threshold, called the `live-key threshold`, then the SSTable contains too much garbage, which should be collected for better utilization of disk space. For each SSTable s, the total number of KV pairs stored in s is computed at creation, and initially the number of valid KV pairs is equal to the total number of KV pairs in s. When a key stored in s is updated with a fresh value, the key with the fresh value will be stored in a new SSTable file. Thus, when we update a pointer to the new location object for the key in the B+-tree, we decrease the number of valid KV pairs in s. Based on these two numbers, we can compute the live-key ratio of each SSTable.

While the goal of the live-key ratio based selection is to collect garbage on disks, the selection based on the leaf node scans in the B+-tree attempts to improve the sequentiality of KVs stored in L_0. Whenever a background compaction is executed, it invokes a leaf node scan, where we scan B+-tree leaf nodes for a certain fixed number of keys in a round-robin fashion. During the scan, we count the number of unique SSTable files, where scanned keys are stored. If the number of unique files is larger than the threshold, called the `leaf node threshold`, we add those files to the compaction candidate list. In this work, the number of keys to scan for a leaf node scan is decided based on two factors, the average number of keys stored in a single SSTable (which depends on the size of a value) and the number of SSTables to scan at once.

For the selection based on the degree of sequentiality per range query, we divide a queried key range into several sub-ranges when operating a range query. For each sub-range, which consists of a predefined number of keys, we keep track of the number of unique files accessed. Once the range query operation is done, we find the sub-range with the maximum number of unique files. If the number of unique files is larger than the threshold, called the `sequentiality degree threshold`, we add those unique files to the compaction candidate list. This feature is useful in improving sequentiality especially for requests with Zipfian distribution (like YCSB [17]) where some keys are more frequently read and scanned.

For recovery purposes, we basically add the compaction candidate list, the total number of KV pairs, and the number of valid KV pairs for each SSTable to the SSTable metadata (which is logged in the MANIFEST file). In SLM-DB, the compaction and flush operations update the B+-tree. Therefore, we need to maintain a compaction log persisted in PM for those operations. Before starting a compaction/flush operation, we create a compaction log. For each key that is already stored in some SSTable but is written to a new file, we add a tuple of the key and its old SSTable file ID to the compaction log. Also, for compaction, we need to keep track of the list of files merged by the on-going compaction. The information about updated keys and their old file IDs will be used to recover a consistent B+-tree and live-key ratios of SSTables if there is a system crash. After the compaction/flush is completed, the compaction log will be deleted. Note that some SSTable files added to the compaction candidate list by the selection based on the leaf node scans and the degree of sequentiality per range query may be lost if the system fails before they are committed to the MANIFEST file. However, losing some candidate files does not compromise the consistency of the database. The lost files will be added to the list again by our selection schemes.

4 KV Store Operations in SLM-DB

Put: To put a KV pair to a KV store, SLM-DB inserts the KV pair to MemTable. The KV pair will eventually be flushed to an SSTable in L_0. The KV pair may be compacted and written to a new SSTable by the selective compaction of SLM-DB.

Get: To get a value for a given key *k* from a KV store, SLM-DB searches MemTable and Immutable MemTable in order. If *k* is not found, it searches *k* in the B+-tree, locates the KV pair on disk (by using the location information pointed to by the B+-tree for *k*), reads its associated value from an SSTable and returns the value. If SLM-DB cannot find *k* in the B+-tree, it returns "not exist" for *k*, without reading any disk block.

Range query: To perform a range query, SLM-DB uses a B+-tree iterator to position it to the starting key in the B+-tree by using the `seek` method and then, scans a given range using `next` and `value` methods. KV pairs inserted to SLM-DB can be found in MemTable, Immutable MemTable or one of the SSTables in L_0. Therefore, for the `seek` and `next` methods, the result of the B+-tree iterator needs to be merged with the results of the two iterators that search the key in MemTable and Immutable MemTable, respectively, to determine the final result, in a way similar to LevelDB.

"Insert if not exists" and **"Insert if exists"**: "Insert if not exists" [36], which inserts a key to a KV store only if the key does not exist, and "Insert if exists", which updates a value only for an existing key, are commonly used in a KV store. Update workloads such as YCSB Workload A [17] are usually performed on an existing key such that for a non-existing key, the KV store returns without inserting the key [9]. To support these operations, SLM-DB simply searches the key in the B+-tree to check for the existence of the given key. In contrast, most LSM-tree based KV stores must check multiple SSTable files to search the key in each level in the worst case.

5 Crash Recovery

SLM-DB provides a strong crash consistency guarantee for in-memory data persisted in PM, on-disk data (i.e. SSTables) as well as metadata on SSTables. For KV pairs recently inserted to MemTable, SLM-DB can provide stronger durability and consistency compared to LevelDB. In LevelDB, the write of data to WAL is not committed (i.e., `fsync()`) by default because WAL committing is very expensive and thus, some recently inserted or updated KVs may be lost on system failures [23, 26, 32]. However, in SLM-DB, the skiplist is implemented such that the linked list of the lowest level of the skiplist is guaranteed to be consistent with an atomic write or update of 8 bytes to PM, without any logging efforts. Therefore, during the recovery process, we can simply rebuild higher levels of the skiplist.

To leverage the recovery mechanism of LevelDB, SLM-DB adds more information to the SSTable metadata such as the compaction candidate list and the number of valid KV pairs stored for each SSTable along with the total number of KV pairs in the SSTable. The additional information is logged by the MANIFEST file in the same way as for the original SSTable metadata.

When recovering from a failure, SLM-DB performs the recovery procedure using the MANIFEST file as LevelDB does. Also, similar to NoveLSM [23], SLM-DB remaps the file that represents a PM pool and retrieves the root data structure that stores all pointers to other data structures such as MemTable, Immutable MemTable, and B+-tree through support from a PM manager such as PMDK [5]. SLM-DB will flush Immutable MemTable if it exists. SLM-DB also checks if there is on-going compaction. If so, SLM-DB must restart the compaction of the files that are found in the compaction log.

For flush, SLM-DB uses the information on an updated key with its old SSTable file ID to keep the number of valid KV pairs in each SSTable involved in the failed flush consistent. In case of compaction, it is possible that during the last failed compaction, the pointers in the B+-tree leaf nodes for some subset of valid KV pairs have been committed to point to new files. However, when the system restarts, the files are no longer valid as they have not been committed to the MANIFEST file. Based on the information of keys and their old SSTable file IDs, SLM-DB can include these valid KVs and update the B+-tree accordingly during the restarted compaction. Note that for PM data corruption caused by hardware errors, the recovery and fault-tolerance features such as checksum and data replication can be used [7].

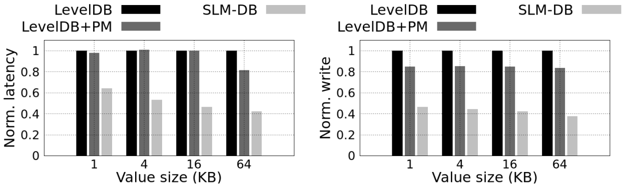
6 Experimental Results

6.1 Methodology

In our experiments, we use a machine with two Intel Xeon Octa-core E5-2640V3 processors (2.6Ghz) and Intel SSD DC S3520 of 480GB. We disable one of the sockets and its memory module and only use the remaining socket composed of 8 cores with 16GB DRAM. For the machine, Ubuntu 18.04 LTS with Linux kernel version 4.15 is used.

When running both LevelDB and SLM-DB, we restrict the DRAM size to 4GB by using the mem kernel parameter. As PM is not currently available in commercial markets, we emulate PM using DRAM as in prior studies [23, 29, 41]. We configure a DAX enabled ext4 file system and employ PMDK [5] for managing a memory pool of 7GB for PM. In the default setting, the write latency to PM is set to 500ns (i.e., around 5 times higher write latency compared to DRAM [23] is used). PM write latency is applied for data write persisted to PM with memory fence and cacheline flush instructions and is emulated by using `Time Stamp Counter` and spinning for a specified duration. No extra read latency to PM is added (i.e., the same read latency as DRAM is used) similar to previous studies [23, 41]. We also assume that the PM bandwidth is the same as that of DRAM.

We evaluate the performance of SLM-DB and compare its performance with that of LevelDB (version 1.20) over varying value sizes. For all experiments, data compression is turned off to simplify analysis and avoid any unexpected effects as in prior studies [23, 30, 35]. The size of MemTable is set to 64MB, and a fixed key size of 20 bytes is used. Note that all SSTable files are stored on an SSD. For LevelDB, default values for all parameters are used except the MemTable size and a Bloom filter (configured with 10 bits per key) is enabled. In all the experiments of LevelDB, to achieve better performance, we do not commit the write ahead log trading off against data durability. For SLM-DB, the `live-key threshold` is set to 0.7. If we increase this threshold, SLM-DB will perform garbage collection more actively. For the leaf node scan selection, we scan the average number of keys stored in two SSTable files and the `leaf node threshold` is set to 10. Note that the average number of keys stored in an SSTable varies depending on the value size. For selection based on the sequentiality degree per range query, we divide a queried key range into sub-ranges of 30 keys each, and the `sequentiality degree threshold` is set to 8. If we increase the `leaf node threshold` and `sequentiality degree threshold`, SLM-DB will perform less compaction. For the results, the average value of three runs is presented.



(a) Write latency (b) Total amount of write

Figure 4: Random write performance comparison.

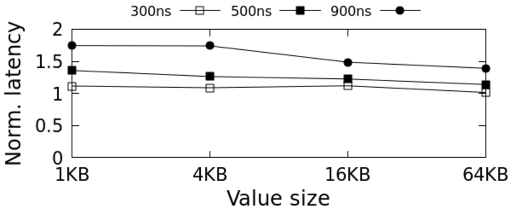
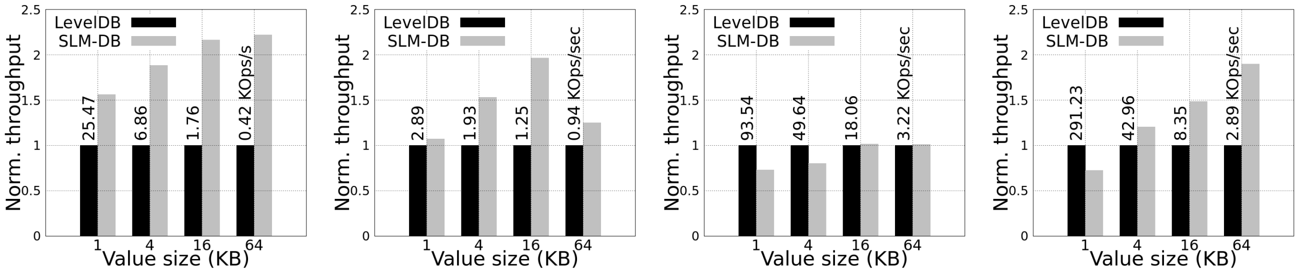


Figure 5: SLM-DB write latency over various PM write latencies, normalized to that with DRAM write latency.

To evaluate the performance of SLM-DB, we use the db_bench benchmarks [3] as microbenchmarks and the YCSB [17] as real world workload benchmarks. The benchmarks are executed as single-threaded workloads as LevelDB (upon which SLM-DB is implemented) is not optimized for multi-threaded workloads, a matter that we elaborate on in Section 7. In both of the benchmarks, for each run, a random write workload creates the database by inserting 8GB data unless otherwise specified, where N write operations are performed in total. Then, each of the other workloads performs $0.2 \times N$ of its own operations (i.e., 20% of the write operations) against the database. For example, if 10M write operations are done to create the database, the random read workload performs 2M random read operations. Note that the size of the database initially created by the random write workload is less than 8GB in db_bench, since the workload overwrites (i.e., updates) some KV pairs.

6.2 Using a Persistent MemTable

To understand the effect of using a PM resident MemTable, we first investigate the performance of a modified version of LevelDB, i.e., LevelDB+PM, which utilizes the PM resident MemTable without the write ahead log as in SLM-DB. Figure 4 shows the performance of LevelDB, LevelDB+PM, and SLM-DB for the random write workload from db_bench over various value sizes. In Figures 4(a) and 4(b), the write latency and total amount of data written to disk normalized to those of LevelDB, respectively, are presented.



(a) Random Write (b) Random Read (c) Range Query (d) Sequential Read

Figure 6: Throughput of SLM-DB normalized to LevelDB with the same setting for db_bench.

As in the figures, in general, the write latency of LevelDB+PM is similar to that of LevelDB, but the total amount of write to disk is reduced by 16% on average as no write ahead log is used. When a large value size is used as in the case of 64KB, the write latency of LevelDB+PM is reduced by 19%. LevelDB+PM also achieves stronger durability of data as inserted KVs are persisted immediately in MemTable. For SLM-DB, the write latency and total amount of data written are reduced by 49% and 57%, compared to LevelDB, on average. This is because SLM-DB further reduces write amplification by organizing SSTables in a single level and performing restricted compaction.

Figure 5 presents the effects of PM write latency on the write performance of SLM-DB. In the figure, the write operation latencies of SLM-DB with PM write latencies of 300, 500 and 900ns, normalized to that of SLM-DB with DRAM write latency, are presented for the random write workload of db_bench. In SLM-DB, as the PM write latency increases, the write performance is degraded by up to 75% when the 1KB value size is used. However, the effect of long PM write latency is diluted as the value size becomes larger.

6.3 Results with Microbenchmarks

Figure 6 shows the operation throughputs with SLM-DB for random write, random read, range query, and sequential read workloads, normalized to those with LevelDB. In the figures, the numbers presented on the top of the bars are the operation throughput of LevelDB in KOps/s. The range query workload scans short ranges with an average of 100 keys. For the sequential read workload, we sequentially read all the KV pairs in increasing order of key values on the entire KV store (which is created by a random write workload). For the random read, range query, and sequential read workloads, we first run a random write workload to create the database, and then wait until the compaction process is finished on the database before performing their operations.

From the results, we can observe the following:

- For random write operations, SLM-DB provides around 2 times higher throughput than LevelDB on average over all the value sizes. This is achieved by significantly reducing the amount of data written to disk for compaction. Note that in our experiments, the overhead of inserting keys to the B+-tree is small and also, the insertion to B+-tree is performed by a background thread. Therefore, the insertion overhead has no effect on the write performance of SLM-DB.
- For random read operations, SLM-DB shows similar or better performance than LevelDB depending on the value size. As discussed in Section 2.2, the locating overhead of LevelDB is not so high when the value size is 1KB. Thus, the read latency of SLM-DB is only 7% better for 1KB values. As the value size increases, the performance difference between SLM-DB and LevelDB increases due to the efficient search of the KV pair using the B+-tree index in SLM-DB. However, when the value size becomes as large as 64KB, the time spent to read the data block from disk becomes long relative to that with smaller value sizes. Thus, the performance difference between SLM-DB and LevelDB drops to 25%.
- For short range query operations, LevelDB with full sequentiality of KVs in each level can sequentially read KV pairs in a given range, having better performance for 1KB and 4KB value sizes. In case of a 1KB value size, a 4KB data block contains 4 KV pairs on average. Therefore, when one block is read from disk, the block is cached in memory and then, LevelDB benefits from cache hits on scanning the following three keys without incurring any disk read. However, in order to position a starting key, a range query operation requires a random read operation, for which SLM-DB provides high performance. Also, it takes a relatively long time to read a data block for a large value size. Thus, even with less sequentiality, SLM-DB shows comparable performance for range queries. Note that when the scan range becomes longer, the performance of SLM-DB generally improves. For example, we ran additional experiments of the range query workload with an average of 1,000 key ranges for the 4KB value size. In this case, SLM-DB throughput was 57.7% higher than that of LevelDB.
- For the sequential read workload to scan all KV pairs,

Table 2: db_bench latency of SLM-DB in microseconds/Op

Value size	1KB	4KB	16KB	64KB
Random Write	25.14	77.44	262.41	1065.68
Random Read	323.56	338.25	406.94	851.98
Range Query	14.68	25.15	54.41	307.71
Sequential Read	4.74	19.35	80.74	182.28

SLM-DB achieves better performance than LevelDB, except for the 1KB value size.

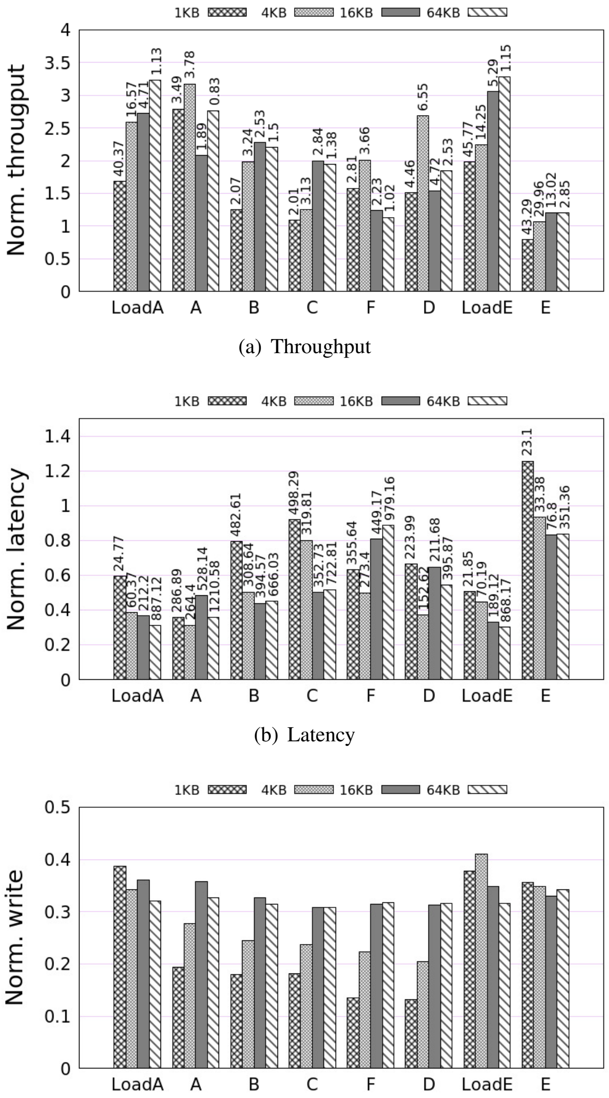
While running the random read, range query, and sequential read workloads, LevelDB and SLM-DB perform additional compaction operations. We measure the total amount of disk write of LevelDB and SLM-DB from the creation of a database to the end of each workload. By selectively compacting SSTables, the total amount of disk write of SLM-DB is only 39% of that of LevelDB on average for the random read, range query, and sequential read workloads. Note that for LevelDB with db_bench workloads, the amount of write for WAL is 14% of its total amount of write on average.

Recall that SLM-DB adds an SSTable to the compaction candidate list for garbage collection only when more than a certain percentage of KV pairs stored in the SSTable are obsolete, and it performs selective compaction for SSTables with poor sequentiality. We analyze the space amplification of SLM-DB for the random write workload in db_bench. Over all the value sizes, the size of the database on disk for SLM-DB is up to 13% larger than that for LevelDB. Finally, we show the operation latency performance of SLM-DB in Table 2.

6.4 Results with YCSB

YCSB consists of six workloads that capture different real world scenarios [17]. To run the YCSB workloads, we modify db_bench to run YCSB workload traces for various value sizes (similar to [35]).

Figures 7(a), 7(b) and 7(c) show the operation throughput, latency, and the total amount of write with SLM-DB, normalized to those with LevelDB over the six YCSB workloads [17]. In Figures 7(a) and 7(b), the numbers presented on top of the bars are operation throughput in KOps/s and operation latency in microseconds/Op of SLM-DB, respectively. For each workload, the cumulative amount of write is measured when the workload finishes. For the results, we load the database for workload A by inserting KVs, and continuously run workload A, workload B, workload C, workload F, and workload D in order. We then delete the database, and reload the database to run workload E. Workload A performs 50% reads and 50% updates, Workload B performs 95% reads and 5% updates, Workload C performs 100% reads, and Workload F performs 50% reads and 50% read-modify-writes. For these workloads, Zipfian distribution is used. Workload D performs 95% reads for the latest keys and 5% inserts. Workload E performs 95% range query and 5% inserts with Zipfian distribution.



(a) Throughput

(b) Latency

(c) Total amount of write

Figure 7: YCSB performance of SLM-DB normalized to LevelDB with the same setting.

In Figure 7(a), the throughputs of SLM-DB are higher than those of LevelDB for all the workloads over varying value sizes, except for workload E with a 1KB value size. For 4∼64KB value sizes, the performance of SLM-DB for workload E (i.e., short range queries) is 15.6% better than that of LevelDB on average due to the fast point query required for each range query and the selective compaction mechanism that provides some degree of sequentiality for KV pairs stored on disks. For workload A, which is composed of 50% reads and 50% updates, updating a value for a key is performed only when the key already exists in the database. Thus, this update is the “insert if exists” operation. For this operation, SLM-DB efficiently checks the existence of a key through a B+-tree search. On the other hand, checking the existence of a key is an expensive operation in LevelDB. If the key does not exist, LevelDB needs to search the key at every level. Therefore, for workload A, SLM-DB achieves 2.7 times higher throughput than LevelDB does on average.

As shown in Figure 7(c), the total amount of write in SLM-DB is much smaller than that of LevelDB in all the workloads. In particular, with a 1KB value size, SLM-DB only writes 13% of the data that LevelDB writes to disk while executing up to workload D. Note that for LevelDB with YCSB workloads, the amount of write for WAL is 11% of its total amount of write on average.

6.5 Other Performance Factors

In the previous discussion, we mainly focused on how SLM-DB would perform for target workloads that we envision for typical KV stores. Also, there were parameter and scheme choices that were part of the SLM-DB design. Due to various limitations, we were not able to provide a complete set of discussion on these matters. In this section, we attempt to provide a sketch of some of these matters.

Effects of varying live-key ratios In the above experiments, the live-key ratio is set to 0.7. As the ratio increases, SLM-DB will perform garbage collection more aggressively. We run experiments of the random write and range query workloads of db_bench with a 1KB value size over varying live-key ratios of 0.6, 0.7, and 0.8. With ratio=0.7, the range query latency decreases by around 8%, while write latency increases by 17% due to more compaction compared to ratio=0.6. With ratio=0.8, the range query latency remains the same as that with ratio=0.7. However, with ratio=0.8, write performance is severely degraded (i.e., two times slower than ratio=0.6) because the live-key ratio selection scheme adds too many files to the candidate list, making SLM-DB stall for compaction. Compared to ratio=0.6, with ratio=0.7 and ratio=0.8, the database sizes (i.e., space amplification) decrease by 1.59% and 3.05%, whereas the total amounts of disk write increase by 7.5% and 12.09%, respectively.

Effects of compaction candidate selection schemes The selection schemes based on the leaf node scans and sequentiality degree per range query can improve the sequentiality of KVs stored on disks. Using YCSB Workload E, which is composed of a large number of range query operations, with a 1KB value size, we analyze the performance effects of these schemes by disabling them in turn. First, when we disable both schemes, the latency result becomes more than 10 times longer than with both schemes enabled. Next, we disable the sequentiality degree per range query, which is only activated by a range query operation, while keeping the leaf node scans for selection. The result is that there is a range query latency increase of around 50%. Finally, we flip the two selection schemes and disable the leaf node scans and enable the sequentiality degree per range query scheme. In this case, the result is around a 15% performance degradation. This implies that selection based on the leaf node scans will play an important role for real world workloads that are composed of a mix of "point queries, updates and occasional scans" as described in the study by Sears and Ramakrishnan [36].

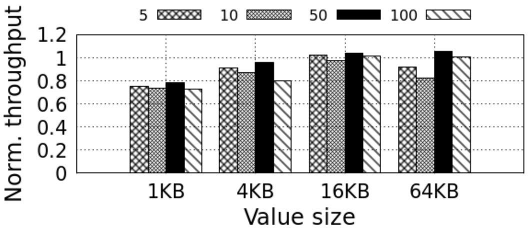


Figure 8: Range query performance of SLM-DB over various key ranges normalized to LevelDB with the same setting.

Short range query Figure 8 shows the range query performances of db_bench with SLM-DB over various key ranges, 5, 10, 50, and 100, normalized to those of LevelDB. For small key ranges such as 5 and 10, the performance trend over different value sizes is similar to that of the random read workload shown in Figure 6(b) as the range query operation depends on random read operations needed to locate the starting key.

Smaller value sizes We evaluate the performance of SLM-DB for a database with a 128 byte value size for random write, random read, and range query workloads in db_bench. Note that with this setting, the total number of write operations becomes so large that, for the range query workload, we choose to execute for only 1% of the write operations due to time and resource limitations. For these experiments, we find that write performance of SLM-DB is 36.22% lower than that of LevelDB. The reason behind this is PM write latency, where it is set to 500ns. With DRAM write latency, in fact, write performance of SLM-DB becomes 24.39% higher, while with 300ns PM write latency, it is only 6.4% lower than LevelDB. With small value sizes, we see the effect of PM write latency on performance. Even so, note that SLM-DB provides reasonable write performance with strong durability of data, considering that the performance of LevelDB with `fsync` enabled for WAL is more than 100 times lower than that with `fsync` disabled for WAL. For random read operations, SLM-DB improves performance by 10.75% compared to LevelDB. For range query operations, with the key range sizes of 50 and 100, performances of SLM-DB are 17.48% and 10.6% lower, respectively, than those of LevelDB. However, with the key range sizes of 5 and 10, SLM-DB becomes more than 3 times slower than LevelDB as LevelDB takes advantage of the cache hits brought about by the high sequentiality of KVs stored on disk.

LevelDB with additional B+-tree index We implement a version of LevelDB that has an additional B+-tree index stored in PM as SLM-DB. This version utilizes the B+-tree index for random read operations and the B+-tree iterators for range query operations. We evaluate the performance of LevelDB with the B+-tree index over various value sizes using db_bench as in Section 6.3. For the random write workload, the performance of LevelDB with the B+-tree is almost the same as that of LevelDB. However, for the random read workload, the performance is almost the same as that of SLM-DB. For the range query workload, it shows 6.64% higher performance than LevelDB on average over all value sizes, as it leverages not only the full sequentiality of KVs in each level, but also the B+-tree iterator instead of multiple iterators of LevelDB (each of which iterates each level in an LSM-tree).

Effects of PM bandwidth In the above experiments, it is assumed that PM bandwidth is the same as that of DRAM (i.e., 8GB/s). We run experiments of SLM-DB with a 1KB value size over various PM bandwidths, 2GB/s, 5GB/s and 8GB/s. Note that for these experiments, we used a different machine with two Intel Xeon Octa-core E5-2620V4 processors (2.1Ghz) to decrease the memory bandwidth by thermal throttling.

Write performance of SLM-DB is affected not only by MemTable insertion of a KV pair but also by background compaction, which can stall write operations. For compaction, the performance degradation caused by lower PM bandwidths for B+-tree insertion is around 5%. However, the effect of PM bandwidth on compaction performance is negligible as file creation is the performance dominating factor. For MemTable insertion, performance with 2GB/s is degraded by 9% compared to that with 8GB/s. Therefore, the final write performance with 2GB/s is around 6% lower than that with 8GB/s. With 5GB/s, the performance of MemTable insertion is not degraded, showing final write performance similar to that with 8GB/s. We also find that with value sizes of 4, 16, and 64KB, write performance of SLM-DB is not affected by PM bandwidth.

For the random read workload, the time to search MemTable and Immutable MemTable and to query the B+-tree is very small comprising just 0.27% of the total time of a read operation for all PM bandwidths used in the experiments. Thus, the effect of PM bandwidth on read performance of SLM-DB is negligible. Note that in our experiments, PM and DRAM read latency is assumed to be the same. However, we believe this has only minor implications on the final read latency as the time to read data from PM during a read operation is very small compared to the time to read a block from disk.

Larger DBs We evaluate the performance of LevelDB and SLM-DB for a database that is created by inserting 20GB data over various value sizes using db_bench. Note that we use 4GB DRAM for the experiments. The write and read throughputs of SLM-DB are 2.48 and 1.34 times higher, respectively, than those of LevelDB (on average), while the range query throughput is 10% lower. Recall that for the database with 8GB data insertion (in Section 6.3), SLM-DB showed 1.96 and 1.45 times higher write and read throughput, respectively, and 11% lower range query throughput than LevelDB.

7 Discussion

Persistent memory cost SLM-DB utilizes extra PM for MemTable, Immutable MemTable, the B+-tree index, and compaction log. For MemTable and Immutable MemTable, its size is user configurable and constant. On the other hand, PM consumed for the B+-tree index depends on the value size, as for a fixed sized database, with a smaller value size, the number of records increases, which results in a larger B+-tree index. Finally, the compaction log size is very small relative to the size of the B+-tree index.

In our experiments of the database with 8GB data insertion, the total amount of PM needed for MemTable and Immutable Memtable is 128MB. For the B+-tree, a total of 26 bytes are used per key in the leaf nodes: (integer type) 8 bytes for the key, 8 bytes for the pointer, and 10 bytes for the location information. Thus, the total amount of PM needed for the B+-tree is dominantly determined by the leaf nodes. In particular, with the 1KB and 64KB value size, SLM-DB uses around 700MB and 150MB of PM, respectively. The cost of PM is expected to be cheaper than DRAM [20], and so we will be able to achieve high performance KV stores with a reasonably small extra cost for PM in many cases.

Parallelism SLM-DB is currently implemented as a modification of LevelDB. Thus, LevelDB and consequently, SLM-DB as well, has some limitations in efficiently handling multi-threaded workloads. In particular, for writes, there is a writer queue that limits parallel write operations, and for reads, a global lock needs to be acquired during the operations to access the shared SSTable metadata [11]. However, by design, SLM-DB can easily be extended for exploiting parallelism; a concurrent skiplist may replace our current implementation, lock-free search feature of FAST and FAIR B-tree for read operations may be exploited, and multi-threaded compaction can be supported [8]. We leave improving parallelism of SLM-DB as future work.

8 Related Work

KV stores utilizing PM have been investigated [4, 6, 23, 41]. HiKV assumes a hybrid memory system of DRAM and PM for a KV store, where data is persisted to only PM, eliminating the use of disks [41]. HiKV maintains a persistent hash index in PM to process read and write operations efficiently and also has a B+-tree index in DRAM to support range query operations. Thus, it needs to rebuild the B+-tree index when the system fails. Unlike HiKV, our work considers a system in which PM coexists with HDDs or SSDs similar to NoveLSM [23]. Similarly to HiKV, it is possible for SLM-DB to have the B-tree index in DRAM and the hash table index in PM. However, for a hybrid system that has both PM and disks, the final read latency is not strongly affected by index query performance as disk performance dominates, unless the database is fully cached. Thus, in such a hybrid setting, using a hash table rather than a B-tree index does make a meaningful performance difference, even while paying for the additional overhead of keeping two index structures. pmemkv [6] is a key-value store based on a hybrid system of DRAM and PM, which has inner nodes of a B+-tree in DRAM and stores leaf nodes of the B+-tree in PM.

NoveLSM [23] and NVMRocks [4] redesign an LSM-tree based KV store for PM. NoveLSM proposes to have an immutable PM MemTable between the DRAM MemTable and the disk component, reducing serialization and deserialization costs. Additionally, there is a mutable PM MemTable, which is used along with the DRAM MemTable to reduce stall caused by compaction. Since the DRAM MemTable with WAL and the mutable PM MemTable are used together, the commit log for the DRAM MemTable and versions of keys need to be carefully maintained to provide consistency. When a large mutable PM MemTable is used, heavy writes are buffered in MemTable and flush operations from Immutable MemTable to disk will occur less frequently. However, to handle a database with a size larger than the total size of DRAM/PM MemTables and Immutable MemTables, KVs in Immutable MemTable will eventually need to be flushed to disk. In NVMRocks, the MemTable is persisted in PM, eliminating the logging cost like SLM-DB, and PM is also used as a cache to improve read throughput [4]. In both NoveLSM and NVMRocks, PM is also used to store SSTables. However, in our work, we propose a new structure of employing a persistent B+-tree index for fast query and a single level disk component of SSTable files with selective compaction, while leveraging the memory component similar to an LSM-tree. In general, SLM-DB can be extended to utilize a large PM MemTable [23], multiple Immutable MemTables [8], and a PM cache [4], orthogonally improving the performance of the KV store.

Optimization techniques to enhance the performance of an LSM-tree structure for conventional systems based on DRAM and disks have been extensively studied [10, 11, 30, 35, 37, 38, 40]. WiscKey provides optimized techniques for SSDs by separating keys and values [30]. In WiscKey, sorted keys are maintained in an LSM-tree while values are stored in a separate value log without hierarchical levels similar to SLM-DB, reducing I/O amplification. Since decoupling keys and values hurts the performance of range queries, it utilizes the parallel random reads of SSDs to efficiently prefetch the values. HashKV similarly separates keys and values stored on SSDs as WiscKey does [14]. It optimizes garbage collection by grouping KVs based on a hash function for update-intensive workloads. LOCS looks into improving the performance of an LSM-tree based KV store by leveraging open-channel SSDs [38].

VT-tree [37] proposes stitching optimization that avoids rewriting already sorted data in an LSM-tree, while maintaining the sequentiality of KVs sufficiently to provide efficient range query performance similar to SLM-DB. However, VT-tree still needs to maintain KV pairs in multiple levels, and does not focus on improving read performance. LSM-trie focuses on reducing write amplification, especially for large scale KV stores with small value sizes by using a trie structure [40]. However, LSM-trie does not support range query operations as it is based on a hash function. PebblesDB proposes the Fragmented Log-Structured Merge Trees, which fragments KVs into smaller files, reducing write amplification in the same level [35]. FloDB introduces a small in-memory buffer on top of MemTable, which optimizes the memory component of an LSM-tree structure and supports skewed read-write workloads effectively [11]. TRIAD also focuses on skewed workloads by keeping hot keys in memory without flushing to disk [10]. The fractal index tree is investigated to reduce I/O amplification for B+-tree based systems [13].

There have been several studies to provide optimal persistent data structures such as a radix tree [29], a hashing scheme [43], and a B+-tree [16, 22, 34, 42] in PM. They propose write optimal techniques while providing consistency of the data structures with 8-byte failure atomic writes in PM.

9 Concluding Remarks

In this paper, we presented the Single-Level Merge DB (SLM-DB) that takes advantage of both the B+-tree index and the LSM-tree approach by leveraging PM. SLM-DB utilizes a persistent B+-tree index, the PM resident MemTable, and a single level disk component of SSTable files with selective compaction. Our extensive experimental study demonstrates that SLM-DB provides high read and write throughput as well as comparable range query performance, compared to LevelDB, while achieving low write amplification and near-optimal read amplification.

Acknowledgments

We would like to thank our shepherd Ajay Gulati and the anonymous reviewers for their invaluable comments. This work was partly supported by Institute for Information & Communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No. 2015-0-00590, High Performance Big Data Analytics Platform Performance Acceleration Technologies Development) and National Research Foundation of Korea (NRF) funded by the Korea government(MSIT) (NRF-2018R1A2B6006107 and NRF-2016M3C4A7952634). Young-ri Choi is the corresponding author.

References

[1] Intel and micron produce breakthrough memory technology. https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/.

[2] Kyoto Cabinet: a straightforward implementation of DBM. http://fallabs.com/kyotocabinet/.

[3] Leveldb. https://github.com/google/leveldb.

[4] NVMRocks: RocksDB on Non-Volatile Memory Systems. http://istc-bigdata.org/index.php/nvmrocks-rocksdb-on-non-volatile-memory-systems/.

[5] pmem.io Persistent Memory Programming. https://pmem.io/.

[6] pmemkv. https://github.com/pmem/pmemkv.

[7] Recovery and Fault-Tolerance for Persistent Memory Pools Using Persistent Memory Development Kit (PMDK). https://software.intel.com/en-us/articles/recovery-and-fault-tolerance-for-persistent-memory-pools-using-persistent-memory.

[8] RocksDB. https://rocksdb.org/.

[9] YCSB on RocksDB. https://github.com/brianfrankcooper/YCSB/tree/master/rocksdb.

[10] Balmau, O., Didona, D., Guerraoui, R., Zwaenepoel, W., Yuan, H., Arora, A., Gupta, K., and Konka, P. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)* (2017).

[11] Balmau, O., Guerraoui, R., Trigonakis, V., and Zablotchi, I. FloDB: Unlocking Memory in Persistent Key-Value Stores. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)* (2017).

[12] Beaver, D., Kumar, S., Li, H. C., Sobel, J., and Vajgel, P. Finding a Needle in Haystack: Facebook's Photo Storage. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2010).

[13] Bender, M. A., Farach-Colton, M., Fineman, J. T., Fogel, Y. R., Kuszmaul, B. C., and Nelson, J. Cache-oblivious Streaming B-trees. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)* (2007).

[14] Chan, H. H. W., Li, Y., Lee, P. P. C., and Xu, Y. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)* (2018).

[15] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2006).

[16] Chen, S., and Jin, Q. Persistent B+-trees in Non-volatile Main Memory. *Proceedings of the VLDB Endowment 8*, 7 (Feb. 2015), 786–797.

[17] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)* (2010).

[18] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (2007).

[19] Dulloor, S. R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., and Jackson, J. System Software for Persistent Memory. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)* (2014).

[20] Eisenman, A., Gardner, D., AbdelRahman, I., Axboe, J., Dong, S., Hazelwood, K., Petersen, C., Cidon, A., and Katti, S. Reducing DRAM Footprint with NVM in Facebook. In *Proceedings of the 13th EuroSys Conference (EuroSys)* (2018).

[21] Huai, Y. Spin-Transfer Torque MRAM (STT-MRAM) : Challenges and Prospects. *AAPPS bulletin 18*, 6 (Dec. 2008), 33–40.

[22] Hwang, D., Kim, W.-H., Won, Y., and Nam, B. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *Proceedings of the 16th Usenix Conference on File and Storage Technologies (FAST)* (2018).

[23] Kannan, S., Bhat, N., Gavrilovska, A., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)* (2018).

[24] Kannan, S., Gavrilovska, A., Gupta, V., and Schwan, K. HeteroOS - OS design for heterogeneous memory management in datacenter. In *Proceedings of the 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)* (2017).

[25] Kim, H., Seshadri, S., Dickey, C. L., and Chiu, L. Phase Change Memory in Enterprise Storage Systems: Silver Bullet or Snake Oil? *ACM SIGOPS Operating Systems Review 48*, 1 (May 2014), 82–89.

[26] Kim, W.-H., Kim, J., Baek, W., Nam, B., and Won, Y. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *Proceedings of the 21th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2016).

[27] Küay, E., Kandemir, M., Sivasubramaniam, A., and Mutlu, O. Evaluating STT-RAM as an energy-efficient main memory alternative. In *Proceedings of the 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2013).

[28] Lakshman, A., and Malik, P. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review 44*, 2 (Apr. 2010), 35–40.

[29] Lee, S. K., Lim, K. H., Song, H., Nam, B., and Noh, S. H. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies (FAST)* (2017).

[30] Lu, L., Pillai, T. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST)* (2016).

[31] Marmol, L., Sundararaman, S., Talagala, N., and Rangaswami, R. NVMKV: A Scalable, Lightweight, FTL-aware Key-Value Store. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)* (2015).

[32] Mei, F., Cao, Q., Jiang, H., and Tintri, L. T. LSM-tree Managed Storage for Large-scale Key-value Store. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC)* (2017).

[33] O'Neil, P., Cheng, E., Gawlick, D., and O'Neil, E. The Log-structured Merge-tree (LSM-tree). *Acta Informatica 33*, 4 (June 1996), 351–385.

[34] Oukid, I., Lasperas, J., Nica, A., Willhalm, T., and Lehner, W. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)* (2016).

[35] Raju, P., Kadekodi, R., Chidambaram, V., and Abraham, I. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)* (2017).

[36] SEARS, R., AND RAMAKRISHNAN, R. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2012).

[37] SHETTY, P. J., SPILLANE, R. P., MALPANI, R. R., ANDREWS, B., SEYSTER, J., AND ZADOK, E. Building Workload-Independent Storage with VT-Trees. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)* (2013).

[38] WANG, P., SUN, G., JIANG, S., OUYANG, J., LIN, S., ZHANG, C., AND CONG, J. An Efficient Design and Implementation of LSM-tree Based Key-value Store on Open-channel SSD. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)* (2014).

[39] WONG, H. . P., RAOUX, S., KIM, S., LIANG, J., REIFENBERG, J. P., RAJENDRAN, B., ASHEGHI, M., AND GOODSON, K. E. Phase Change Memory. *Proceedings of the IEEE 98*, 12 (Dec 2010), 2201–2227.

[40] WU, X., XU, Y., SHAO, Z., AND JIANG, S. 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 ATC)* (2015).

[41] XIA, F., JIANG, D., XIONG, J., AND SUN, N. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)* (2017).

[42] YANG, J., WEI, Q., CHEN, C., WANG, C., YONG, K. L., AND HE, B. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)* (2015).

[43] ZUO, P., AND HUA, Y. A Write-Friendly and Cache-Optimized Hashing Scheme for Non-Volatile Memory Systems. *IEEE Transactions on Parallel and Distributed Systems 29*, 5 (May 2018), 985–998.