



저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:



저작자표시. 귀하는 원저작자를 표시하여야 합니다.



비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 [이용허락규약\(Legal Code\)](#)을 이해하기 쉽게 요약한 것입니다.

[Disclaimer](#)

Master's Thesis

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

Olzhas Kaiyrakhmet

Computer Science and Engineering

The Graduate School of UNIST

2019

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

Olzhas Kaiyrakhmet

Computer Science and Engineering

Graduate School of UNIST

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

A thesis
submitted to the Graduate School of UNIST
in partial fulfillment of the
requirements for the degree of
Computer Science and Engineering

Olzhas Kaiyrakhmet

12.12.2018

Approved by



Advisor

Young-ri Choi

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

Olzhas Kaiyrakhmet

This certifies that the thesis of Olzhas Kaiyrakhmet is approved.

12.12.2018

signature



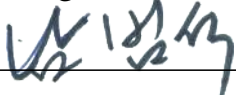
Young-ri Choi: Advisor

signature



Sam H. Noh: Thesis Committee Member #1

signature



Beomseok Nam: Thesis Committee Member #2

Abstract

This work investigates possible integration of persistent random access memory (PM) support to key-value (KV) stores in order to provide more supreme performance and consistency than currently existing solutions. Due to byte-addressable and persistent benefits of the new memory, a novel type of KV-store has been born out of a blend B+-tree and Log Structured Merge Tree data structures, which is called Single-Level Merge DB (SLM-DB). Both of the trees' benefits have been embraced to the new design improving IO throughput, and as well as decreasing write and read disk amplification. New architecture exploits PM to reside B+-tree to index data stored in the disk and write buffer to store all new updates, which similar to LSM-tree approach. Data stored in the disk are partitioned to files and organized as a single level, unlike to LSM-tree, and SLM-DB does operate selective compaction on KV pairs in order to do garbage collection and maintain data sequentiality. Such architectural approach made SLM-DB have a more superior performance from LevelDB in read throughput for 1.96 and in write throughput for 2.2, with commensurate range query result, while incurring only 39% of competitor's disk write on average. Additionally, KV-store's architecture secures full consistency of the data, which ensures full recovery after any crash.

Contents

I	Introduction	1
II	Background	4
2.1	Key-Value Store Operations	4
2.2	Log Structure Merge Tree	4
2.3	LevelDB	5
2.4	Limitations of LevelDB	7
2.5	Persistent Memory	9
2.6	B+-tree	10
III	Architecture	11
3.1	Persistent Memtable	12
3.2	B+-tree Index on PM	13
3.3	Selective Compaction	14
3.4	Crash Recovery	16
IV	Evaluation	18
4.1	Methodology	18
4.2	Using a Persistent Memtable	19

4.3	PM sensitivity	20
4.4	Results with Microbenchmarks	20
4.5	Results with YCSB	22
4.6	Analysis	24
V	Related Work	27
VI	Conclusion	29
	References	30

List of Figures

1	LSM-tree with $K+1$ components	4
2	LevelDB architecture	5
3	Locating overhead over various value sizes	8
4	SLM-DB architecture	11
5	Insertion to a persistent skip-list	12
6	Random write performance comparison	19
7	SLM-DB throughput performance with different PM latency over value sizes compared to LevelDB	20
8	db_bench performance of SLM-DB normalized to LevelDB with the same setting	21
9	YCSB performance of SLM-DB normalized to LevelDB with the same setting	23

List of Tables

1	Read operations' overhead breakdown (in us)	8
2	Comparison of different memory technologies	9
3	YCSB throughput (operations/sec) results for LevelDB and SLM-DB	24
4	YCSB latency (us/operations) results for LevelDB and SLM-DB	24

I Introduction

Industry reports a fast pace growing demand on data-intensive applications that can handle large read and write throughput [1, 2]. That issue was presented by Yahoo!, stating that the requirement of application is becoming data driven by each day [3]. That leads to intensive research and development of key-value stores.

A key-value store, or key-value database, is a data storing paradigm with the idea of managing associated data. Data that can be stored varies from small words to documents and pictures, each needing its own way of handling it. With an extension of key-value stores with a query language, NoSQL database can be implemented, such as this [4, 5]. That is why KV-stores have been playing a crucial role in applications, such as web indexing [6], social networking [7], online shopping [5], and cloud photo storage [8]. Typical key-value stores are using LSM-tree [9], B-tree [10] and Hash-Table [11]. However, latter one lacks range query operation due to not maintaining sequential indexing structure. Therefore, LSM-tree and B-tree data structures are more popular to be used for wide-range purpose applications.

LSM-tree and B-tree fundamentally differ from each other, where each has winning point. First one excels on write and sequential read, where second performs better for random read [12]. B-tree based KV-stores, such as KyotoCabinet [13], incur small random writes to disk during key-value insertion operations and undergo high write amplification to maintain stable structure [14]. Therefore, B-tree is more suitable for read-intensive applications.

LSM-tree based KV-stores are more suited to application uses with high write throughput demand. Applications like Bigtable [6], LevelDB [15], RocksDB [7] and Cassandra [4] are widely used for such purposes, varying from single node database to distributed storage. Tree benefits its high write performance to an in-memory buffer and sequential disk writes. Write operations load key-value pairs to memory and concurrently unload buffer in batches, writing them to disk storage making the sequential insert. Due to that, structure undergoes high read and write amplification clustering key-value pairs stored together multiple times. The reason is LSM-tree organized as multiple levels of files, where files batches get merge-sorted from one level to next, to facilitate a fast search.

Key-value store performance can be increased by using better performing hardware, like replacing HDD by SSD. However, only that is not enough to fully exploit the benefits of faster hardware, but a redesign of architecture is needed [16, 17]. Therefore, research of key-value stores for new persistent memory gains momentum [1, 18, 19]. Persistent memory (PM) is prominent hardware that is going to be both non-volatile and byte-addressable, such as phase change memory [20], spin transfer torque MRAM [21],

and 3D Xpoint [22]. It is projected for the PM to be having near-DRAM read latency, with higher write latency for up to 5 times and lower bandwidth for up to 5~10 times compared to DRAM [23–27]. New memory is going to have a large capacity, and it is expected to coexist with SSD and HDD [24, 25]. With all stated above, persistent memory opposes present storage technologies, like flash-memory and hard-disk, making it appealing. Withal, it will make more outstandingly attractive to have application work with it efficiently, using all assets of upcoming memory.

This work presents a new KV store that adopts PM to improve IO throughput, the *Single-Level Merge DB* (SLM-DB). Taking advantage of both data structures, B+-tree and LSM-tree, and as well as new memory architecture, making this DB achieve outstanding results. It performs high results on random query read operations with high write throughput. On top of that, SLM-DB does low write amplification and near-optimal read amplification, since it does one block read operation per read query. This is accomplished due to the persistent B+-tree index that finds a KV pair without any additional disk access, which is not the case in most of the LSM-tree based databases. High write throughput managed by using LSM-tree technique of in-memory buffering of KV updates, persisting in PM. Also, DB stores all necessary metadata in persistent memory. By doing that, SLM-DB can avoid any redundant logging, obtains strong consistency of the system and faster recovery time.

In SLM-DB, key-value pairs are mainly stored on disks organized in *single-level*. Having B+-tree in the system, DB does not have the requirement to keep all KV pairs in sorted order, which significantly reduces write amplification. Still, obsolete KVs are in need to be garbage collected. Additional to it, it is preferred to keep some scale of sequentiality of KV pairs on disks to produce adequate results for range queries. Therefore, a compaction scheme performs a range bound *merge* of key-value pairs stored on disks, called *selective compaction*.

The main contributions of this work are as follows:

- Taking advantage of new memory’s benefits to maintain most of the critical parts of the system. Those are B+-tree index, in-memory buffer and disks storage’s metadata. First, one gives advantage on indexing KV pairs and relaxes a restriction of fully sorted ordered data, reducing write amplification. Second, it does buffer recent updates in persistent memory and extra data logging. And the latter gives the advantage of saving critical information about disk organization in new memory and employ it for fast recovery after a crash.
- Developing mechanism of selective compaction that deletes unused key-value pairs and keeps sufficient sequentiality of data for scan operations. The mechanism is based on the following three schemes: (1) live-key data ratio of a file; (2) a leaf node scans in the B+-tree for data

sequentiality; (3) a degree of sequentiality per user's range query request. This approach prevents a huge amount of data rewriting and produces reasonable results for range queries.

- Fast recovery mechanism after a system failure, which totally relies on metadata and logs saved on persistent memory. There is no more rely upon disk logs and MANIFEST file, therefore SLM-DB can load the system state almost in an instant and recovering some of the error states from replaying the log. This feature is an extension of the prior study [28].
- Implementing SLM-DB on top of LevelDB codebase and using *Hwang et al* [24] work of persistent B+-tree. The implementation is designed to be fully consistent on system failure with fast recovery procedure and strong guaranteed data durability. Evaluation of the work by *db_bench* microbenchmark [12, 15] and YCSB benchmark [29] have shown superior performance result up to 1.96 times in read throughput and up to 2.2 times in write throughput compared to LevelDB, and with commensurate performance on range query. During the experiment, SLM-DB has done a less total disk write operations, only 39% and 27% of those of LevelDB on average, for *db_bench* and YCSB workloads respectively.

II Background

In this section will be discussed background topics of this study. Firstly will be described common key-value store operations. Next, transcending to log-structured merge tree and it's famous implementation - LevelDB. Furthermore, we will be touching about persistent memory and some tricks to work with. The section will be concluded by covering B+-tree.

2.1 Key-Value Store Operations

Get. Get(key) operation retrieves most relevant data associated with the key. If there is no such data, an error message gets returned.

Put. Put(key, value) operation inserts value data into the storage, persisting it, and creating an associated index by key to the value. If there is data already exists in the storage to the associated key, data gets updated.

Iterators. Most of the key-value stores provide iterators over the entire data. Data usually sorted by associated index keys, allowing it to iterate in sequential order by next() operation. Also, iterator allows to jump over to certain key to iterate, calling seek() operation. Additional to it, a user can a specify special way of sorting data by setting a comparator.

Range Query. RangeQuery(key1, key2) operation returns all the data with index keys in the range between key1 and key2. Often, a range query is implemented by using iterator by calling its function seek() and iterating until key2.

2.2 Log Structure Merge Tree

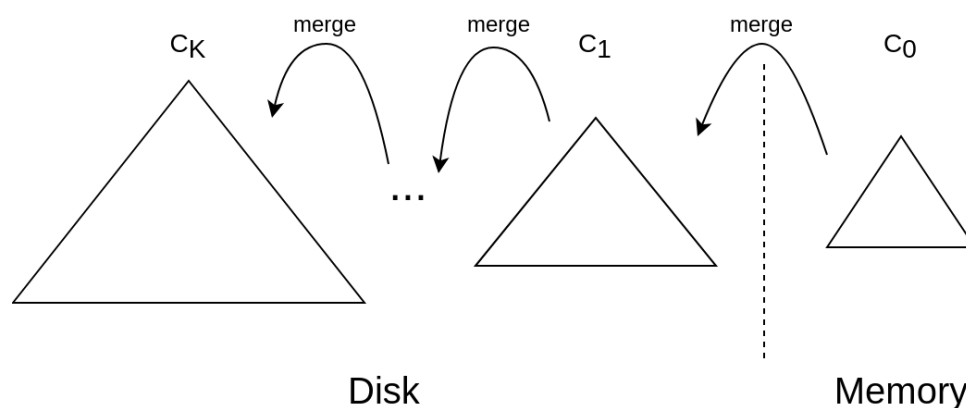


Figure 1: LSM-tree with K+1 components

Log Structured Merge tree (LSM-tree) is a data structure with a high-performance attribute for large volume data insertion that provides indexed access to it. LSM-tree usually maintains key-value pair type of data storing. The data structure is divided into levels or components, where each corresponds to its own data structure to effectively manage data in an underlying storage medium.

Conceptually, Log Structure Merge Tree consists of two components, C_0 and C_1 . C_0 does reside in memory and usually referred to as write in-memory buffer. C_1 is a disk-based component, where it holds most of the data in the tree since C_0 cannot hold much of data inside of expensive memory.

All instruction always accesses C_0 first and C_1 next. For insertions of key-values, the tree inserts pair directly to C_0 , buffering them into small but fast memory. A large volume of insertion makes C_0 grow in size and when it gets to the certain size threshold. Therefore, memory component will concurrently move large batch data to C_1 , merging them together and doing sequential disk insert. For a read instruction, the tree will query for data C_0 first, and if not found, will query C_1 . This way LSM-tree exploits the benefits of the leveled approach, sequential disk writes and optimized data structure for underlying storage mediums, which makes it achieve fast write performance and reasonable read performance.

In real implementations, LSM-tree does have more components than two, which is also suggested by the data structure authors. Figure 1 shows an LSM-tree of $K + 1$ components divided into in-memory component C_0 and disk residing components from C_1 to C_K . Whenever some component C_i is getting larger than a certain threshold for it, it moves batches of data to C_{i+1} , merging data. Each component i has larger data capacity than $i - 1$, making component K largest. Due to the multi-component approach, each data retrieval should query components from 0 to K until found, making read operation quite expensive.

2.3 LevelDB

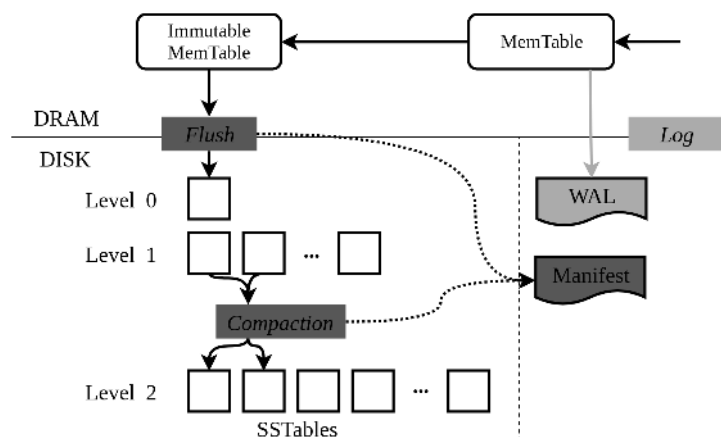


Figure 2: LevelDB architecture

LevelDB is most popular LSM-tree implementation inspired by Google's BigTable [6] that is widely used in production for many applications. LevelDB supports all basic KV store operations: *put*, *get*, *delete*. Also, it provides an *iterator*, which iterates lexicographical among keys and being able to retrieve values. From Figure 2, LevelDB is divided into memory and disk components, as defined in LSM-tree. Memory components are Memtable and Immutable Memtable. Main disk components are *Sorted String Table* (SSTable) files that organized in multiple level structure. Additionally, data log and manifest log are persisted on disk to guarantee recoverable state.

Memtables are implemented using a skip-list data structure, which guarantees $O(\log n)$ average time for search and insertion, keeping the ordered sequence of elements. Memtable is used as write in-memory buffer to speed up input operation. In LevelDB's Memtable, any delete operation is treated as an update with tombstone marker. When Memtable's size caps a threshold, it is marked as immutable and replaced by new Memtable to continue buffer incoming writes. In the meantime, Immutable Memtable gets scheduled for disk flush in a background thread, creating new SSTable file. That new SSTable file goes directly to L_0 of disk component in LevelDB.

All key-value updates do not go directly to Memtable, but first being persisted on Write Ahead Log (WAL). This is done to guarantee recovery of Memtable in case of system crash. Only after being logged, KV pair will be inserted into Memtable. So, when Memtable gets flushed to disk, data log will be deleted together with in-memory buffer. However, LevelDB does not commit updates to the WAL by default, calling `fsync()` operation. This is done to trade off durability and consistency for write performance.

Disk component of LevelDB is arranged as multiple level structure, from the lowest L_0 level to the highest L_K level. Any level L_i is 10 times larger in size capacity than the next lower level L_{i-1} . Each level consists of SSTable files, and in levels from L_1 to L_K , files' key ranges do not overlap with other to be arranged in sorted order. The exception is L_0 , as Immutable Memtable flushes new SSTable into the level without doing any merge because doing so will stall Memtable to be marked as Immutable. That will lead to pause all update operations in the system. Therefore, LevelDB skips expensive merge operation between Immutable Memtable and L_0 .

In order to keep levels L_1 to L_K organized, LevelDB does compaction operation on the background. Compaction is the operation that merges one file from L_i with L_{i+1} , keeping L_i to be oversized. File from L_i and overlapping files from L_{i+1} are merged together, performing merge sort algorithm. All older key-value pairs from L_{i+1} will be overwritten by newer key-value pairs from L_i . For compaction, file from L_i is picked in a round-robin manner and Level i is chosen from the most oversized level ratio.

Compaction for L_0 is almost similar, except when it tries to merge L_0 with L_1 , more than one file from L_0 are picked. There are several files that overlap with each other, which are picked from L_0 and being merged with other overlapping files from L_1 . In the end, LevelDB uses compaction to keep files sorted in level for quick search and to perform garbage collection in files.

LevelDB persists LSM-tree organization in a log file called MANIFEST. Each time there is a change of structure in LSM-tree, like new file(s) created by flush or compaction, that change is logged to manifest file. Logged information includes changes in levels and new SSTable files' metadata, which consists of size, file name and a key range of the file. Once the state gets logged, any existing obsolete files are purged from the system. Therefore, after each flush and compaction operation, changes in LSM-tree structure are committed to log, persisting LevelDB's state to the disk for recovery.

2.4 Limitations of LevelDB

Slow read operations. When LevelDB does read operation (i.e., point query), it sequentially searches memory components first and if not found, looks in disk components. So, priority is given first to Memtable, and after to Immutable Memtable, only then it searches in the disk from L_0 to L_K . That sequence check from most recent updated component to least recently updated one, retrieving freshest key-value pair if the same key exists in the system. To query Memtables, it is a straightforward skip-list search operation. When it searches in the disk, LevelDB needs to find an SSTable file that contains needed key-value pair and that's quite expensive for the system.

LevelDB does not contain any per-key indexing, so it needs to search for a file from L_0 to L_K first. For L_0 system does a linear search that might contain the key, and for other levels - binary search. The reason is that files in L_0 might overlap, whereas in levels L_1 to L_K do not and sorted. By finding a file does not guarantee the existence of the key in the file. LevelDB looks for index block of the file, which stores the information of each data block's first key in the file, to find corresponding data block of the file by doing a binary search. To avoid unnecessary data block read, which is 4KB or larger, the system can check Bloom filter of that data block. Only then it reads the data block and tries to retrieve key-value pair from there. If not found, LevelDB looks for the next file and does that until the key-value pair is discovered.

It is noticeable, LevelDB does many disk read operation in order to find a key-value pair for user's get operation. For example, Table 1 drop downs all overhead for searching random KV pair in the system with and without Bloom filter. Results were evaluated by db_bench benchmark [15] on 4GB DRAM machine setting with 1KB value size and 8GB data pre-inserted into the database. Details of the environment that have been used are discussed in Section IV. Table sections description: File search

KV store	File search	Block search	Bloom filter	Unnecessary block read
LevelDB w/o BF	0.79	3.25	0	13.60
LevelDB w BF	0.78	3.05	0.88	10.69
SLM-DB	0.98		0	0

Table 1: Read operations' overhead breakdown (in us)

- time spent on finding an SSTable file containing searched key; Block search - operation time used for locating(indexing) a corresponding block in the file; Bloom filter - time to check bloom filter; and "Unnecessary block read" refers to overhead read of the file and block when the key does not exist there. The last one causes the most overhead due to not having per-key indexing and a need to query levels L_0 to L_K of disk component making expensive disk operation. As is shown in the table, SLM-DB does insignificantly low overhead locating needed block. After discovering and reading the right data block from the disk, KV pair gets extracted from the block. For the above results, the average time used to read and process a data block for all three databases - 320.5 microseconds.

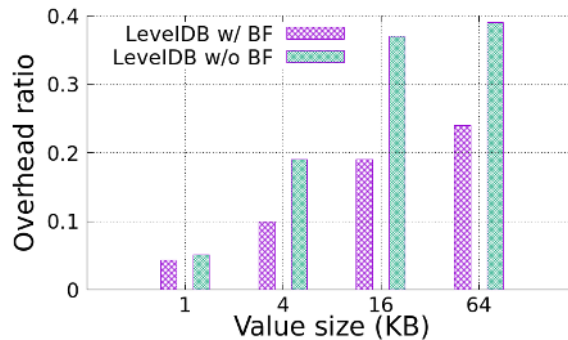


Figure 3: Locating overhead over various value sizes

Figure 3 shows a comparison of LevelDB with and without Bloom filter for overhead portion ratio from random read operation over various value sizes. The ratio is search overhead's percentage to the total read operation latency. The table clearly shows us the direct correlation of value size increase and read operation's overhead. LevelDB does much less overhead for large value sizes with enabled Bloom filter. However, it still high, taking around 24% of the total read time as overhead will significantly grow with value size. This is a disadvantage of LSM-tree approach for a read operation that requires a system to do the search of several disk components until the right key-value pair is found.

High write and read amplification.

The community has well investigated an issue of LSM-trees related to high write and read amplification [2, 14, 17, 30]. High write amplification caused by constantly keeping files sorted in all levels of disk component hierarchy by continuously doing a merge-sort batch of files. Background thread does merge

files from level to level without a stop until files are fully organized, while at the same time serving incoming KV pair insertions on the foreground. The write amplification ratio, which is described as the ratio difference between the total size of data written to disk and the total amount of data requested to insert by a user, can be higher than $10 \times k$ for an LSM-tree with k levels [2, 17, 30].

2.5 Persistent Memory

Category	Read latency	Write latency	Random accessing	Data persistence
DRAM	60ns	60ns	High	False
PCM	50 ~ 70ns	150 ~ 1000ns	High	True
ReRAM	25ns	500ns	High	True
NAND Flash	35us	350us	Low	True

Table 2: Comparison of different memory technologies

Emerging persistent memory (PM) technologies, such as Phase Change Memory(PCM) [20], Resistive Memory (ReRAM) [31], Spin Transfer Torque MRAM [21], and 3D Xpoint [22] can provide faster persistence than traditional Disk and Flash as well as being byte-addressable. Table 2 shows the characteristics of different memory technologies. Read latency is expected to be similar to DRAM, but write latency to be longer. The high performance of PM is going to be achieved by using the same bus interface as DRAM. For persistence of data on PM, the atomic transaction should be used when there are memory updates. Failure atomicity unit of PM is expected to be 8 bytes [32, 33]. This should be taken into consideration when designing persistent data structures, so every update can ensure consistency in case of a crash.

In current modern processor, memory write operations might be reordered in the unit of a cache line to maximize memory bandwidth. Also, updates are done in CPU stay in cache until it gets evicted. Therefore, every update can not ensure data to be written to memory directly. Thus, expensive memory mfence and cache flush instructions (CLFLUSH and MFENCE in Intel x86 architecture) need to be called explicitly [19, 24, 32–35]. Also, need to be aware of data size to be flushed, as atomicity unit is 8 bytes. If data size to be flushed to persistent memory is larger, then only partial data update will be done in case of failure leading to inconsistency of the data. For that, techniques like COW and logging are used. So, a software architecture for PM should be designed carefully in order to guarantee consistency.

Persistent Memory opens doors to many possible opportunities in the software world. Especially to the ones that deal data-intensive computations. That is way, it is crucial to delivering a fast performing KV store.

2.6 B+-tree

Initially, B-tree was designed to be stored on disk, while being buffered into memory in order to provide efficient and persistent indexing. That allows B-tree to perform a single disk seek each time an uncached data is requested by a user. Modification B-tree, B+tree, does change the structure of nodes, so key-value pairs are only stored in leaf nodes. That makes B+-tree key-value iteration more efficient. While performing efficiently for read and scan requests, it lacked performance on updates, which required to do expensive disk write operation each time. Therefore, it has a lower demand for high write throughput application, unlike LSM-tree.

With an advance of storage technologies, like SSD, update operations in B-trees made less overhead. However, upcoming new persistent memory makes B-tree very appealing. Due to new memory architecture, it became possible to avoid the long living disadvantage of B-trees, slow persistent update performance [24, 33–35].

III Architecture

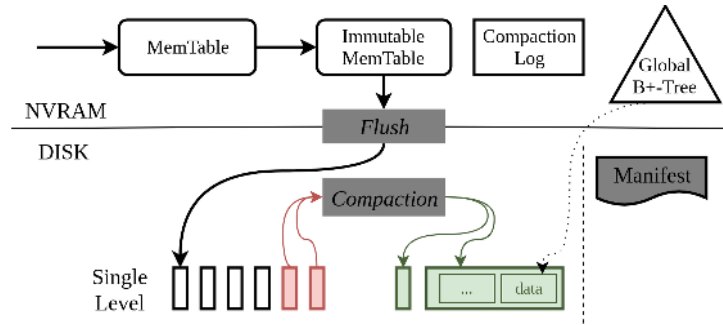


Figure 4: SLM-DB architecture

Design and implementation of our key-value database, Single-Level Merge DB (SLM-DB), is explained in this section. Overall picture to system architecture is on Figure 4. Alike LevelDB, SLM-DB has Memtable and Immutable Memtable, but with a change of making them reside on persistent memory. Therefore, making those component persistent grants ability to avoid write-ahead log (WAL), and providing strong durability and consistency on any system failure. The developed key-value database has a disk component organized as a single level of SSTable files, which is different to all common LSM-tree databases. Thus, it has been named as Single-Level Merge DB. By having an only single level, the system is not required to do merge operation from level to level and so, making less key-value pair rewrites. This architecture with persistent buffer component and single-level disk organization allows to significantly reduce write amplification and have improved write performance. This presented design architecture has been revised and added with the extra feature of persistent LSM-tree, compared to the prior work [28].

In order to be able to read data for an associated key from the single level disk component, SLM-DB has a persistent B+-tree index for that. With that, every key in the system is indexed, so no need of having fully sorted order in the level to find a key-value pair. Yet, by doing that system does not trigger compaction, which does garbage collection of obsolete key-value pairs. Additionally, not having to sort the level, hurts the performance of range queries, which does benefit on sequentiality of key-value pairs. Accordingly, a selective compaction scheme, which selectively merges SSTable files, is developed in the SLM-DB. Besides, SLM-DB is fully consistent on system failure, as it ensures that B+-tree and (single-level) LSM-tree is being consistent by having a back up during every compaction, which is a compaction log stored on PM.

Source code implementation is done on top of the LevelDB (version 1.20). Memtable implementation has been upgraded to persist it on PM. SSTable file format and merge-sort algorithm of multiple SSTable files have been left untouched. The system saves all needed information, like the list of valid files and

file metadata, on the PM, so it can have fast recovery from any crash. Also, read logic implementation (i.e. random read and range queries) have been completely changed from vanilla LevelDB by using a persistent B+-tree.

For an implementation of persistent B+-tree, FAST and FAIR B+-tree [24] have been chosen to be in SLM-DB. It has shown an outperforming result from other state-of-the-art persistent B-trees in range query workloads since it kept all keys in sorted order. Moreover, FAST and FAIR B+-tree achieves the highest write throughput by taking advantage of the memory level parallelism and the ordering constraints of dependent store instructions.

3.1 Persistent Memtable

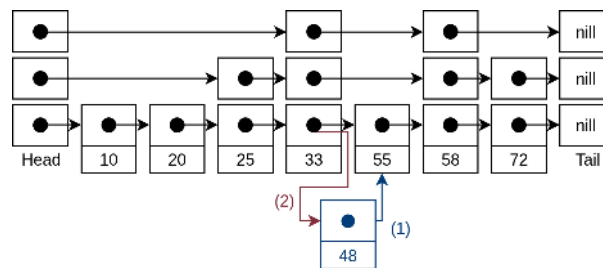


Figure 5: Insertion to a persistent skip-list

Algorithm 1 Insert(key, value, prevNode)

- 1: curNode := NewNode(key, value);
 - 2: curNode.next := prevNode.next;
 - 3: mfence();
 - 4: clflush(curNode);
 - 5: mfence();
 - 6: prevNode.next := curNode;
 - 7: mfence();
 - 8: clflush(prevNode.next);
 - 9: mfence();
-

In SLM-DB, Memtable is implemented using a skip-list data structure. Actual persistence of the Memtable is achieved by storing the first node level of skip-list on PM, and other link levels on DRAM. Therefore, making it persistent linked-list with non-persistent skip-list links and storing necessary data on PM. This way, data structure's insertion, update and deletion operations can be done atomically (8-byte atomicity), like it is shown in Figure 5 with insert operation process. Algorithm 1 explains in detail the insertion process to PM part of the skip-list. First, a node with the key-value pair and next pointer

is created and persisted calling memory fence and cacheline flush instructions. After, next pointer to the new node is updated on a previous node and persisted as well. Due to pointer size and atomic update unit size are both 8-byte, there are no complications in the procedure. Update of an existing key in Memtable is done in the same way as to insert operation, without doing in-place update. Deletion in Memtable is treated as an update operation, but with a tombstone mark. With all of this, SLM-DB avoids having WAL for data durability.

3.2 B+-tree Index on PM

To improve the performance of read operations, SLM-DB uses a B+-tree to for index search of KV pairs stored on SSTable files. When flushing KV pairs in Immutable Memtable to an SSTable file, each key is inserted to the B+-tree. The key is inserted to a B+-tree's leaf node with a pointer to object, which is stored on persistent memory and consists of index information about that key in the disk, like SSTable file ID, block offset and block size. That object is called index metadata.

Whenever a key is getting updated with a new value, and that data is being flushed to an SSTable file, updates in B+-tree is needed as well. New index metadata object created and persistent on PM. Then, on the leaf node of the B+-tree, the pointer of the associated key is being updated from old index metadata to newly created one, using atomic insertion. Persistent memory operations are handled by PMDK [36], controlling all the allocations and deallocations on persistent memory pool, as well as garbage collection of obsolete objects.

Note, likewise to LevelDB, the SLM-DB supports string-type keys, which are converted to integer-type key to insert into the B+-tree in this work.

Building a B+-tree. Whenever Immutable Memtable is scheduled for flush to L_0 , index of each key from new SSTable file is inserted to B+-tree as well. For flush operation, and compaction as well, SLM-DB creates two background threads, one that makes new files and other inserts indexes from that files to B+-tree.

In the file creation thread, the system creates a new SSTable file and appends all unique key-value pairs from Immutable Memtable to the file. Once creation thread has flushed file to disk storage, it creates file metadata, which consists of file's id number, size, a range of keys and a total count of keys. That metadata is flushed to PM. During the creation of the file, key and index list was created. That list is passed to another thread, B+-tree insertion thread, which processes that list and adds keys and indexes to the B+-tree. Once the list is processed and all key-indexes are inserted, the insertion thread is done. Next, new LSM-tree organization (i.e. SSTable metadata) is copied. Copied version of LSM-tree is

updated with new metadata and flushed to PM. Then, a pointer for LSM-tree data structure is changed to new updated one, making the atomic change. In the end, SLM-DB deletes Immutable Memtable.

Scanning a B+-tree. SLM-DB provides an iterator, which can easily be used to scan all KV-pairs in sorted order, which is alike to LevelDB. Iterator currently has these operations: `begin`, `valid`, `seek`, `next` and `value`. The `begin()` function makes iterator to point to the smallest key in the system. The `valid()` function identifies if an iterator is valid or not, like if it iterated until the end. The `seek(k)` method positions an iterator in the KV store, so it points to the key `k`, or the smallest key that is larger than `k`, if `k` does not exist. The method `next()` moves iterator to next KV pair, and method `value()` returns a value of the key pointed by an iterator.

In order to scan key-value pairs in disk component among SSTable files, a B+-tree iterator is implemented. Iterator of B+-tree has the same functionality of methods as system's iterator, which is explained above. In the persistent FAST+FAIR B+-tree, keys are sorted in leaf node and leaf nodes have sibling pointers. Therefore, iteration in this B+-tree is intuitive. The `value()` returns index information for the pointed key, with which the SLM-DB can read KV pair directly from disk.

3.3 Selective Compaction

SLM-DB has no requirement to keep KV pairs sorted in L_0 , as there a per-key index exists in the system. However, in order to do garbage collection of obsolete KV pairs and improve sequentiality of KV pairs, the system has a *selective compaction* scheme. SLM-DB selects certain SSTable (selection algorithm will be explained below) and it adds them to a *compaction candidate list* of SSTables. That list is handled by background compaction thread, which merges files together producing new files. Compaction thread can be scheduled when there is a change in SSTable files organization (like, Immutable Memtable is flushed to disk), or too many seek operations to a certain file (alike in LevelDB), or a number of SSTable file in the compaction candidate list is larger than a certain threshold. Before merging files, SLM-DB does pick a subset of files to merge from compaction candidates. That subset has the highest sum of an overlapping ratio between each of the files in subset compared to other subsets. The overlapping ratio between the two files is computed by comparing their key ranges. Also, compaction limits number of SSTables to be merged at once, so that foreground user operations will not be disturbed.

The compaction process is using two threads, one for file creation and other for index insertion to B+-tree, which similar to Immutable Memtable flush operation discussed above. However, compared to flush, compaction have to check each KV pair if it is obsolete or not. That is done during the merging process by querying B+-tree, that checks by comparing B+-tree index information and KV-pair's

merging file. If KV pair is not valid then it will not be appended to new SSTable file.

The difference between compaction and flush, compaction does create one or several SSTable files out merging other SSTables, unlike to flush which creates SSTable file out of Immutable Memtable. During each SSTable file creation, per key index list is created, similarly to flush operation. After writing SSTables and synchronizing it to disk, it creates file's metadata on PM. Once file's metadata is persisted, index list of the file is transferred to B+-tree insertion thread. File creation thread continues with next file creation from the merge, while B+-tree insertion thread does update indexes from the list. That process continues until all new files are created from merged files and indexes are updated. Next, a new version of LSM-tree is created making a copy of the current version and applying new changes, like deleting obsolete SSTables and adding new ones to the structure. In the end, the system atomically changes pointer from the current LSM-tree version to new and deletes obsolete SSTable files from the file system.

Selection of compaction candidates in SLM-DB is done by these three schemes: a *live-key ratio* of an SSTable, a *leaf node scan* in the B+-tree, and a *degree of sequentiality per range query*. The first scheme, the live-key ratio of an SSTable does select a candidate based on the ratio of valid KV pairs to obsolete KV ones store in each SSTable. Thus, an SSTable file will be selected as compaction candidate for garbage collection to utilize a disk space better, if it's ratio is lower than a threshold, which is called *live-key threshold*. Change of valid key count happens when a key K got updated with a new value and it is being flushed to a new file F_i . The old value of the key K is in file F_j . So, when the update of an index happens in B+-tree from file F_j to F_i , the system will decrease the count of valid keys in F_j . Therefore, SLM-DB can always compute an up-to-date live-key ratio of each SSTable based on the count of valid keys and the total count of keys.

The second scheme based on the leaf node scans in the B+-tree is used for improving the sequentiality of KV pairs stored in L_0 . Whenever there is background compaction is being executed, it enforces a leaf node scan, which scans B+-tree leaf nodes for certain fixed range of keys in a round-robin fashion. During a scan, the system counts a number of unique files being indexed by scanned keys, where keys are always sorted due to persistent FAST-FAIR B+-tree structure. If that number of unique files is larger than a threshold, called *leaf node threshold*, SLM-DB does add those files into the compaction candidate list. The number of keys to be in the range of scan relies on two factors, first is the average number of KV pairs stored in a single file (which depends on KV pair size), and second is the number of SSTable files to scan at once.

Third compaction selection scheme, the selection based on the degree of sequentiality per range query

have a similar purpose as the second one, but with a distinctive difference of increase sequentiality in user requested range query operation. Each range query is divided into sub-ranges when operated, which consists of a predefined number of keys. Each sub-range is tracking of a number of unique files being accessed, and then sub-range with maximum unique files will be picked. If unique file number in that sub-range is larger than a threshold, called `sequentiality degree threshold`, SLM-DB will add those unique files to the compaction candidate list. This feature is most useful in improving sequentiality for Zipfian distribution requests (like YCSB [29]), where some keys are more frequent to be accessed or scanned.

For recovery purposes, SLM-DB stores the state of LSM-tree, Memtable, Immutable Memtable in the persistent memory. LSM-tree stores metadata array of each SSTable (metadata consisting id, size, total key count, valid key count and a key range of the file) and compaction candidate list. Additional to that, the system has a compaction log in PM that logs all index updates done during compaction operation. The information that it logs is an SSTable file id that has a key change index from and SSTable file id that has key changed to. This is needed to recover the valid key count of files being touched during compaction because crash might happen during that operation and whole LSM-tree structure is being persisted only after compaction completed, whereas B+-tree indexes persistently updated on the go. Therefore, it is crucial to recovering actual count of valid keys in an SSTables, so it will match with a count of indexing keys to the SSTable by B+-tree. Compaction log works as a write-ahead log for B+-tree and it is implemented as persistent linked-list, thus the consistency of insert is insured by an atomic update of last node's next pointer to a new node. So first, the log entry is put into a new node and persisted, after atomic update next pointer and then make index update in the B+-tree.

3.4 Crash Recovery

SLM-DB provides strong crash consistency guarantee for data stored in the system, both on-disk data and in-memory data persisted to PM. In order to know full state of the system, SLM-DB reads PM memory pool from DAX filesystem. It uses PMDK recovery scheme, which relies on having a header pointer. Header pointer directs to PM location of root data structure, which saves root pointers of every data structure saved in PM. This way, SLM-DB easily directs through pointers and recovers PM data structure, like LSM-tree, Memtables and compaction log with ease.

For all key-value pairs inserted to Memtable, SLM-DB ensures consistency and durability of that data compared to LevelDB. LevelDB always writes logs to WAL, before writing to in-memory Memtable. However, by default, WAL is not committed (i.e. `fsync()`) after every insertion, because committing to

the filesystem is a very expensive operation, and thus, some of the recently inserted or updated KV pairs might become lost under system failure [16, 20, 21]. In contrary, SLM-DB's skip-list is implemented as it saves lower level of the skip-list (i.e. linked-list) in PM, whereas other levels are in DRAM. Since main data is in the lower level of skip-list, it can guarantee consistency with an atomic write or update of 8 bytes to PM, without any logging. During recovery, other levels of skip-list are being reconstructed without high overhead.

When recovering, SLM-DB checks if a system failure occurred during compaction or flush. If so, there is a need to recover of a valid key count in SSTable metadatas that have been touched during operation, otherwise, B+-tree indexes might be invalid. For that, the system relies on compaction log, through which it recomputes actual valid key count in SSTable files. The system makes sure, that last log entry is valid with an updated in the B+-tree, a crash might have happened between log commit and B+-tree update operation. As to preserve consistency during recovery, SLM-DB copies LSM-tree state and makes changes to it, updating valid key count, mark for deletion obsolete files and adding new files that have been created during compaction or flush. In the end, recovery finishes by changing the pointer of current LSM-tree to the new one, then removing obsolete files from the filesystem, and deleting old LSM-tree with compaction log from PM.

IV Evaluation

4.1 Methodology

For evaluation of the work, two socket machine with 2x Intel Xeon E5-2640v3 processor, 2x 16GB DRAM and Intel SSD DC S3520 (480GB) has been used. One socket has been totally disabled, processor and memory, and remaining 8 core with 16GB DRAM was used. The machine has been set up with Ubuntu 18.04 LTS and kernel 4.15. As PM is not available on the market, PM emulation has been done similar to here [18, 19, 32]. Memory has been restricted by mem kernel parameter to 4GB, and 8GB has been allocated to emulated PM by memmap kernel parameter. Emulated PM region was configured as an ext4 file system with enabled DAX. Persistent memory pool was managed by PMDK [36] and allocated to have 7GB size pool. In default settings, write latency of PM is set to 500ns, which is 5 times higher than DRAM [19]. Emulation of PM write latency is done by adding CPU pause and counting delay time by Time Stamp Counter after data is persisted to the memory by memory fence and cacheline flush instructions. No extra read latency has been added to PM, thus it had same read latency as DRAM. By default, no bandwidth restriction has been applied to PM, as further below evaluation showed no significant in Section 4.3.

Evaluation of SLM-DB is done in comparison with LevelDB (version 1.20) over varying value sizes. Data compression has been turned off for all experiments to avoid any undue impact on the results [30]. Memtable size is configured to 64MB size, and key size is fixed to 20 bytes. All SSTable files are stored on the SSD for both, SLM-DB and LevelDB. For LevelDB, Bloom filter is set to the size of 10 bytes and other parameters are set to default. By default, LevelDB does not commit entries to the WAL to achieve better performance for less data consistency.

For SLM-DB, the `live-key` threshold is set to be 0.7. Increasing that threshold configuration will lead to more active garbage collection by the system. For leaf node scan selection, it was set up to scan the average number of keys stored in two SSTables, and `leaf node` threshold is set to 10. The number of key-value pairs stored in files depends on value size. For selection based on the sequentiality degree per range query, SLM-DB divides total range to sub-ranges of 30 keys, which is the order of system's persistent B+-tree, and `sequentiality degree` threshold is set to 8. The increase of `leaf node` threshold and `sequentiality degree` threshold will lead to less number of compactions done in SLM-DB. All results presented are the average value of three runs.

To evaluate the performance of SLM-DB against LevelDB, `db_bench` [15] was used as a microbenchmark, and YCSB [29] was used as a real-world workload benchmark. For both benchmarks, each run

has created a database by inserting 8GB data, where N number of insert operation performed in total. Then, next workloads perform $0.2 \times N$ operations on the database (for instance, if 10M write operations have been performed, random read workload will perform 2M operations). Note: db_bench does overwrite some portion of the data during write phase (i.e. updates some KV pairs), so the final size of the database after db_bench write workload is a less than 8GB.

4.2 Using a Persistent Memtable

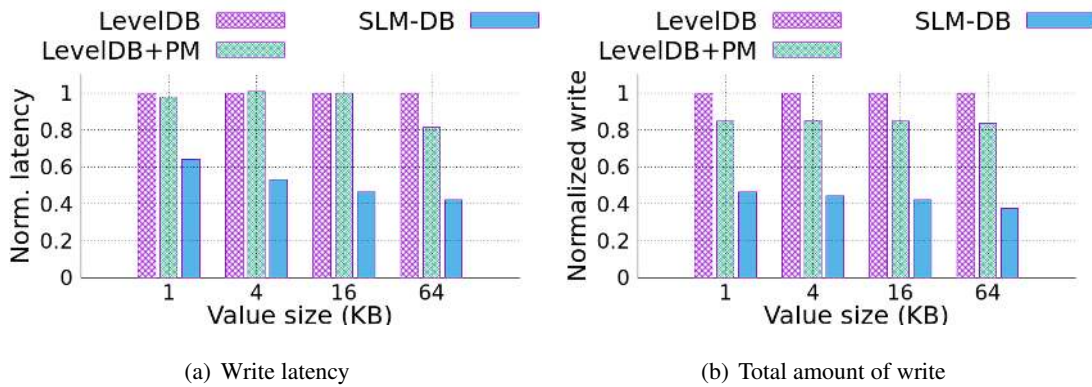


Figure 6: Random write performance comparison

In order to understand the effect of using a PM resident Memtable, LevelDB has been modified to have persistent Memtable without the write-ahead log, similarly to SLM-DB, which was named as LevelDB+PM. Performances of LevelDB, LevelDB+PM, and SLM-DB on random write workload over various value sizes are present in Figure 6 for comparison. Figure 6(a) shows a comparison of write latency and Figure 6(b) shows a comparison of total amount data written to disk (both comparisons are normalized to LevelDB result).

Generally, LevelDB+PM has similar write latency performance to LevelDB, when it's the total amount of write to disk has been reduced by 16% on average, since no WAL has been used. Only when value inserted to the database gets large as the size of 64KB, the write latency is being reduced by 19% due to avoiding logging. Additionally, LevelDB+PM achieves stronger data durability, because of strong write consistency to persistent Memtable. For SLM-DB, write latency got reduced by 49% and total disk write is reduced by 57%, on average compared to LevelDB. SLM-DB achieves such results due to significantly reducing write amplification by organizing SSTables in a single level and executing compaction selectively.

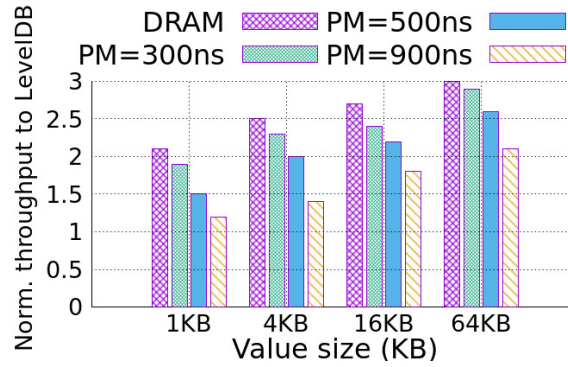


Figure 7: SLM-DB throughput performance with different PM latency over value sizes compared to LevelDB

4.3 PM sensitivity

Figure 7 presents the random write workload throughput result of SLM-DB over different memory latencies and compared to LevelDB. SLM-DB has been tested on different memory write latencies, same as DRAM, 300ns, 500ns, and 900ns. Evaluation is done using random write benchmark of db_bench. In all of the PM latency cases, even with 900ns, SLM-DB outperforms LevelDB on random write. With PM have same latency as DRAM, SLM-DB reaches up to 3 times better throughput then LevelDB, and with 900ns PM latency up to 2.1 times. From this results, it is observed that with value size becoming larger, the effect of a long PM write latency is being diluted.

The same way an evaluation has been done to see the system sensitivity to PM bandwidth. Bandwidth have been tested are 2GB/s, 5GB/s, and 8GB/s, similar to [19]. Write operations had a fluctuation of 5%, as it most of the write performance does not depend on Memtable insertion but on a background thread, which stalls write operation if it is over-scheduled by flush and compaction operations. Thus, as a Memtable being resident on PM, it should have affected the performance of the write operations, but due to stalls that are done by background thread it does not. For read performance, there has been no effect on the result as well. The main reason is that the major time of the read operation is done by disk and PM modules (i.e. Memtable read and B+-tree query) have shown to be the part of only 0.3 % from the total read.

4.4 Results with Microbenchmarks

Figure 8 displays throughput performance results of SLM-DB for random write, random read, range query, and sequential read workloads, which are normalized to the performance of LevelDB. In each workload figure, the number presented on top of the bars are throughput result numbers of LevelDB in thousands of operations per second. Range query workload has tested short range with an average

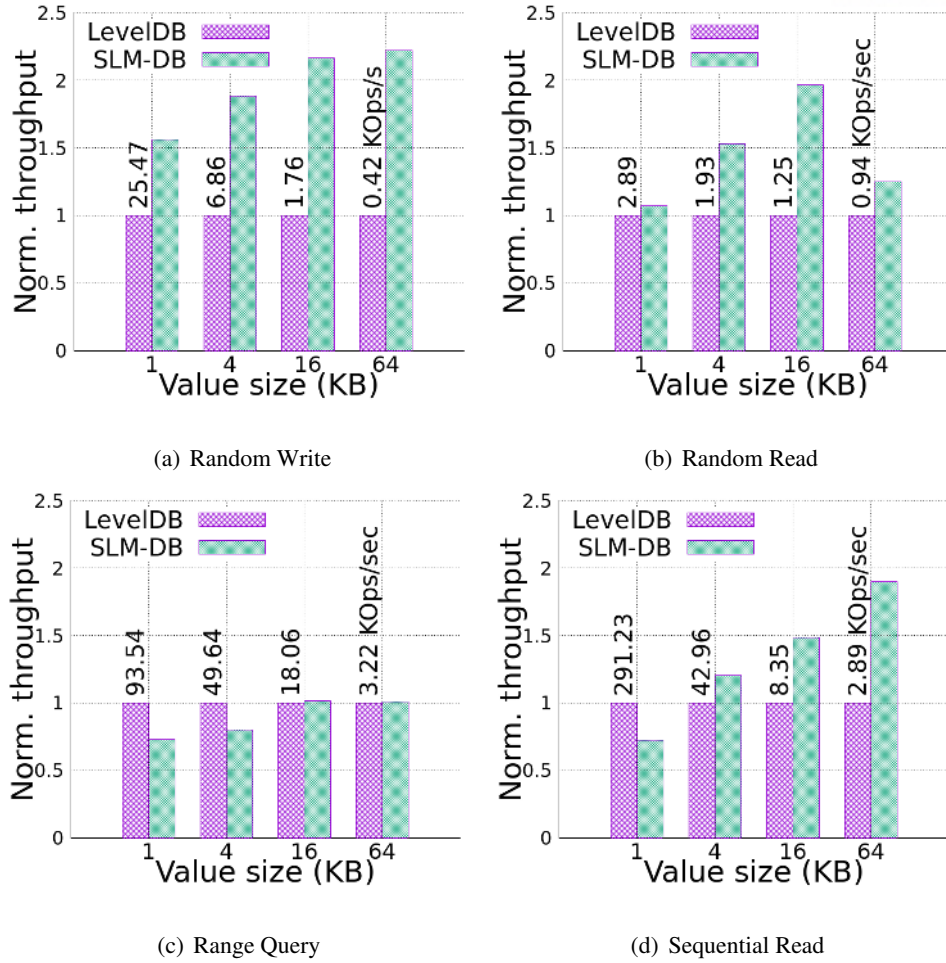


Figure 8: db_bench performance of SLM-DB normalized to LevelDB with the same setting

count of 100 keys. For the random read, range query, and sequential read workloads, write workload have been run to populate a database with data, and then paused before continuing, waiting for all compaction processes to finish.

From the results, the following can be observed:

- For random write operations, SLM-DB outperforms LevelDB in an average of 2 times higher across different value sizes. This is achievable by SLM-DB, because of the notable decrease of data rewrites done by compaction. As B+-tree key and index insertion happen on background, overhead that it produces is minimal. Thus, B+-tree insertion does not interfere with the performance of SLM-DB
- For random read operations, SLM-DB performs similarly or better than LevelDB, which depends on the value size. As was discussed above in Section 2.3, the overhead of finding data block in LevelDB is not high, when the value size is 1KB. Therefore, the SLM-DB surpasses LevelDB's throughput by only 7%. With an increase of value size, the performance difference between SLM-

DB and LevelDB start to have a larger gap due to the more efficient search of the key-value pair using persistent B+-tree index. However, as value size gets large as 64KB, the time to data block read becomes significantly large compared to smaller value sizes. Therefore, the performance gap between SLM-DB and LevelDB gets down to 25%.

- For short range query operations, LevelDB, with fully sorted order of KV pairs in each level, can read KV pairs sequentially in a given range and have better performance for 1KB and 4KB value sizes. For 1KB value size, a data block with a default size of 4KB can contain up to 4 KV pairs. Thus, when the block has been read from the disk, it is cached to the memory and next key-value pair can be retrieved from the same cached block during the scan. Nevertheless, in order to start scanning range, need to perform a random read query to the first key in a range. There, SLM-DB provides high performance. Also, it takes a relatively long time to read a data block for a large value size. Thus, SLM-DB can have comparable performance for range queries even with fewer disk data sequentiality. For instance, when running a range query workload with the range of 1000 keys with 4KB value sizes, SLM-DB's throughput performs better than LevelDB's for about 57.7%.
- For sequential read workload, it scans all KV pair sequentially from the database. SLM-DB achieves better performance than LevelDB, except for 1KB value size.

While running the random read, range query and sequential read workloads, both LevelDB and SLM-DB performed additional compaction operations on disk components. Therefore, total disk writes from the creation of the database until the end of each workload have been measured for both. SLM-DB performed 2.56 times less disk writes on average of all workloads than LevelDB, because of selectively compacting SSTables. Even excluding WAL, LevelDB would make on average only 14% less of total amount disk writes.

4.5 Results with YCSB

YCSB benchmark consists of six workloads, where each captures different real-world scenario [29]. In order to efficiently run the YCSB workloads, the db_bench source code has been changed to run traces of YCSB's workloads with various value sizes (alike to [14]).

Figures 9 (a) and (b) show the throughput of operations and the total amount of disk writes for both LevelDB and SLM-DB over the six YCSB workloads [29]. In figure 9(a), numbers presented on top of the bars are the operation throughput of SLM-DB in operations per second. After the end of each workload run, the cumulative amount of disk writes have been measured and charted on figure 9(b).

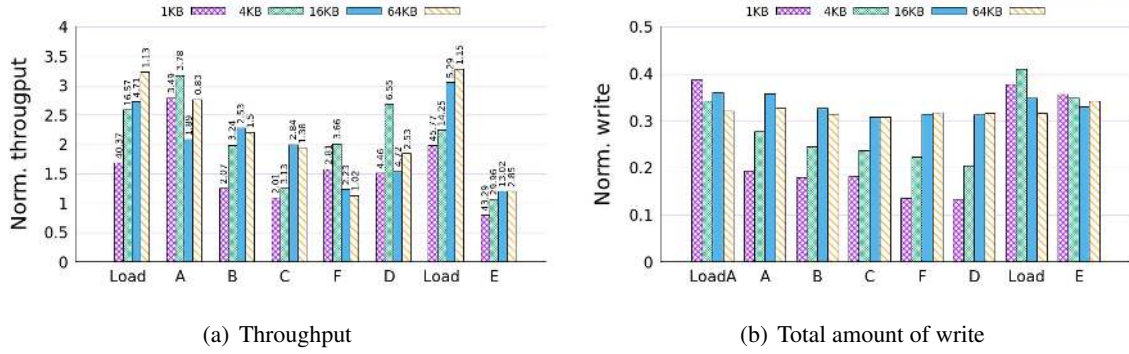


Figure 9: YCSB performance of SLM-DB normalized to LevelDB with the same setting

The execution of YCSB benchmark was suggested by authors in this order: load database, workload A, workload B, workload C, workload F, workload D, purge database, load database, workload E.

Workloads properties:

- Load: 100% insert, Zipfian distribution
- Workload A: 50% read and 50% update, Zipfian distribution
- Workload B: 95% read and 5% update, Zipfian distribution
- Workload C: 100% read, Zipfian distribution
- Workload D: 95% read and 5% insert, latest read
- Workload E: 95% scan and 5% insert, Zipfian distribution
- Workload F: 50% read and 50% read-modify-write, Zipfian distribution

In Figure 9(a), SLM-DB shows higher throughput performances than those of LevelDB for all workloads, except for workload E with 1KB value size. For other value sizes in workload E, SLM-DB does better than LevelDB for 15.6% on average. That's because each range query needs a random query for the first key in range, in which SLM-DB is superior and also providing some level of sequentiality for disk data. For workload A composed of 50% read and 50% update, the update operation is performed only when the updating key exists in the database. In other words, the update is “insert if only exists” operation. For this, SLM-DB needs only to execute query operation on B+-tree, without having to access the disk components. On contrary, LevelDB has to search all the way to disk component, retrieving the KV pair from disk, to make sure that key exists. Therefore, for workload A, SLM-DB is able to achieve on average 2.7 times higher performance than LevelDB does. Detailed throughput results are displayed on Table ??, and details of latency results are on Table ??.

Figure 9(b) shows clearly that SLM-DB makes a much smaller amount of disk writes than LevelDB in all the workloads. Especially after executing workload D with 1KB values size, SLM-DB shows 7.7 times less of disk writes. If WAL is excluded from the LevelDB in YCSB workloads, the total amount of disk writes are reduced by only 11%.

	1kb		4kb		16kb		64kb	
	LevelDB	SLM-DB	LevelDB	SLM-DB	LevelDB	SLM-DB	LevelDB	SLM-DB
Load	23,919.8	40,370.3	6,409.3	16,565.2	1,726.5	4,712.4	350.5	1,127.2
A	1,250.0	3,485.7	1,185.0	3,782.2	911.5	1,893.4	295.1	826.1
B	1,647.2	2,072.1	1,633.5	3,240.0	1,111.2	2,534.4	678.1	1,501.4
C	1,842.3	2,006.9	2,503.8	3,126.9	1,419.1	2,835.0	714.9	1,383.5
F	1,778.8	2,811.9	1,824.2	3,657.6	1,796.0	2,226.4	905.0	1,021.3
D	2,956.2	4,464.4	2,436.7	6,552.1	3,059.5	4,724.1	1,367.9	2,526.1
Load	23,145.1	45,765.2	6,367.8	14,247.0	1,734.3	5,287.5	345.6	1,151.9
E	54,366.5	43,293.8	28,011.5	29,955.1	10,836.0	13,020.3	2,375.9	2,846.1

Table 3: YCSB throughput (operations/sec) results for LevelDB and SLM-DB

	1kb		4kb		16kb		64kb	
	LevelDB	SLM-DB	LevelDB	SLM-DB	LevelDB	SLM-DB	LevelDB	SLM-DB
LoadA	41.8	24.8	156	60.4	579.2	212.2	2853.3	887.1
A	800	286.9	843.9	264.4	1097	528.1	3388.9	1210.6
B	607.1	482.6	612.2	308.6	899.9	394.6	1474.7	666
C	542.8	498.3	399.4	319.8	704.7	352.7	1398.9	722.8
F	562.2	355.6	548.2	273.4	556.8	449.2	1105	979.2
D	338.3	224	410.4	152.6	326.8	211.7	731	395.9
LoadE	43.2	21.9	157	70.2	576.6	189.1	2893.2	868.2
E	18.4	23.1	35.7	33.4	92.3	76.8	420.9	351.4

Table 4: YCSB latency (us/operations) results for LevelDB and SLM-DB

4.6 Analysis

Space Amplification. SLM-DB marks an SSTable file as compaction candidate for garbage collection when the live-key ratio of the file hits a threshold. Therefore, if the system detects the ratio of obsolete KV pairs in several files are larger than a certain threshold, it performs the selective compaction on that files, leading to poor sequentiality. Analyze the space amplification on db_bench’s random write workload showed that SLM-DB does utilize 13% more of disk space.

Persistent memory cost. Persistent memory is utilized for Memtable, Immutable Memtable, B+-tree index and LSM-tree with compaction log, in SLM-DB. As 8GB data is inserted into the database in all of the experiments, a smaller value size leads to a larger number of records, which increases the size of the B+-tree index and compaction log. With 1KB value size, SLM-DB utilizes utmost 750MB of PM, whereas with 64KB value size - 200MB. The cost of PM is expected to be cheaper than DRAM (similar discussion about NVM block devices [37]), and so, SLM-DB achieves higher performance with a reasonable small cost of extra PM.

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 workload E of YCSB, which is composed of a large number of range query operations, the effects of these schemes have been analyzed by disabling them in turn. Disabling both of the schemes lead to increase the result of latency in 10 times, compared when both enabled. Disabling only the sequentiality degree per range query, which activates only during range query operations, while other is enabled, lead to 50% latency increase on average. When another way around, with the sequentiality degree per range query is enabled and the leaf node scan is disabled, the result showed around 15% performance degradation. This implies that selection based on the leaf node scans will play an important role for a real-world workload that is composed of a mix of “point queries, updates and occasional scans” as was described in [3].

Effects of varying live-key ratios In all the above experiment results shown, the threshold of 0.7 is used for the live-key ratio. As ratio increases, SLM-DB performs more aggressive garbage collection. The experiments conducted to see the difference of effect over varying live-key ratios of 0.6, 0.7 and 0.8. To analyze, db_bench’s random write and range query workloads with 1KB were executed. With ratio=0.7, the range query performance is increased by 8%, while decreasing performance of write by 17%, compared to ratio=0.6. With ratio=0.8, no difference in performance of range query is observed in contrast to ratio=0.7. However, write performance is severely degraded (for instance, twice slower than ratio=0.6), due to SLM-DB being overflowed by compaction candidates, which makes it stall incoming write to finish some compaction.

Recovery cost. In order to analyze the cost of recovery for SLM-DB, the worst case scenario has been assumed to happen, and that is when Memtable and Immutable Memtable are being full and compaction is in process. To recover Memtable, need to read root pointer of linked-list and start iterative rebuilding skip-list nodes. That is not integrated into the current system yet, but the prototype algorithm run showed 0.02ms rebuild time for each full Memtable (64MB and 1KB value size). Similarly have done with LSM-tree recovery, using prototype algorithm to measure it’s recovery time. For that, the average number of

files being compacted and being produced after compaction has been used. The algorithm goes through all compaction log, checking B+-tree index changes from old to a new file and recovering live-key count of files. In the end, it creates and persists a new LSM-tree version. All of LSM-tree recovery takes around 0.05ms, which results in total recovery time to little less than 0.1ms. On contrary, LevelDB spends 0.5ms on average to recover after a failure and can take longer than that with the increased size of Memtable.

V Related Work

There are several works on KV stores utilizing PM [18,19,38]. HiKV key-value store assumes utilization of hybrid memory system of DRAM and PM, where data is persisted and save only on PM, which eliminates any usage of disks [18]. HiKV maintains a persistent hash index on PM to process both read and write operations efficiently and also supporting a range query operations by maintaining B+-tree index on DRAM. Therefore, there is need of rebuilding the B+-tree in need if the system happens to get a crush or failure. Contrary to HiKV, this work considers a system where PM coexists with HDDs or SSDs, which is similar to NoveLSM [19].

NVMRocks [38] and NoveLSM [19] optimized and redesigned an LSM-tree to use the advantage of persistent memory to store key-value pairs. In NVMRocks, the Memtables are fully stored on PM, eliminating any logging cost and also adding PM cache layer to improve read performance. NoveLSM [19] proposes to have a persistent Memtable with DRAM Memtable, so they co-exist to reduce stall time caused by compaction. Since DRAM Memtable must have a WAL, there must be careful maintenance of consistency needed, so a version of KV pairs in both Memtables will be up-to-date after a failure recovery. In both of the works, NoveLSM and NVMRocks, PM is used to store some portion or all of the SSTables. However, this work proposes a new structure of employing a persistent B+-tree index for the fast query, persistent LSM-tree-like structure to save up-to-date SSTable files statuses, and a single level disk component of SSTable files with selective compaction. Using this design structure, the system significantly is able to reduce read and write amplification compared to most of traditional LSM-tree based KV stores, while providing high read and write throughput.

Optimization techniques to enhance the overall performance of an LSM-tree structure have been intensively studied for conventional memory and disks systems [2, 14, 16, 30, 39–41]. Also, there are works proposing optimization for certain H/W like SSD. For instance, WiscKey [30] provides an optimized technique for SSD by separating keys and values, where the value stored without any hierarchical level similarly to SLM-DB, which in the end reduces I/O amplification. Since decoupling keys and values hurts the performance of range queries, WiscKey utilizes the parallel random reads of SSD to efficiently prefetch the values. Another similar example is HashKV [1], which also does separate keys and values stored on SSDs. HashKV uses a hash function to group KV to optimize garbage collection for update-intensive workloads. Other work named LOCKS does leverage open-channel SSDs to increase the performance of KV-stores based on LSM-tree.

Further, works on optimizing design structure, like VT-tree [41], that uses stitching optimization that avoids rewriting already sorted data and maintaining sufficient sequentiality of KVs to provide efficient

performance of range queries, which similar to SLM-DB. However, VT-tree still needs to maintain KV pairs in multiple levels, and read performance was out focus. Redesign from tree to trie, LSM-trie [2] focuses on reducing write amplification, having a particular focus on large-scale KV-stores with small value sizes. However, LSM-trie has no support of scan workloads as it is based on hash function. On the other hand, PebblesDB [14] proposes a Fragmented LSM-tree, which fragments KVs into smaller files and reduces write amplification at the same level. Introduction of a small in-memory buffer on top of the Memtable by FloDB [40] optimized the memory component in LSM-tree, which increased support on skewed read-write workloads. TRIAD [39] have done similarly focusing on skewed workloads by keeping hot keys in the memory without flushing to the disk.

Some work on optimizing B+-tree based system have been done as well. The fractal index tree investigates the reduction of I/O amplification. Also, with PM introduction to the arena, works on optimization of persistent B+-tree have become large [24, 33–35]. Other indexes on PM have been evolving as well, like radix tree [32] and a hashing scheme [27]. Above PM data structures have proposed write optimal techniques while providing consistency and durability.

VI Conclusion

This work presents a novel and intuitive redesign of classical LSM-tree based key-value store, which is called Single-Level Merge DB. Main key uniqueness of SLM-DB lies on leveraging B+-tree index with LSM-tree together, in order to perform out better. Also, DB uses persistent memory to make use of both tree structures to the fullest, guarantying both strong consistency and superior performance than classical approaches. Moreover, storing main data structures in PM allows the system recovery by fast and steady. Also, single level structure of disk component lowers the number of merge operations executed, and also providing some level of sequentiality by having selection compaction scheme. Evaluation of this work demonstrates high read and write throughput with the comparable performance of range query, although having very low write amplification and near-optimal read amplification.

References

- [1] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. Hashkv: Enabling efficient updates in KV storage via hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 1007–1019, Boston, MA, 2018. USENIX Association.
- [2] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 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 '15, pages 71–82, Berkeley, CA, USA, 2015. USENIX Association.
- [3] Russell Sears and Raghu Ramakrishnan. blsm: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 217–228, New York, NY, USA, 2012. ACM.
- [4] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.
- [5] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon’s highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.
- [6] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.
- [7] Rocksdb. <https://rocksdb.org/>.
- [8] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in haystack: Facebook’s photo storage. In *Proceedings of the 9th USENIX Conference on Operating*

- Systems Design and Implementation*, OSDI' 10, pages 47–60, Berkeley, CA, USA, 2010. USENIX Association.
- [9] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996.
- [10] Rudolf Bayer and Edward McCreight. Organization and maintenance of large ordered indexes. In *Software pioneers*, pages 245–262. Springer, 2002.
- [11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.
- [12] Google. Leveldb benchmarks. <http://www.lmdb.tech/bench/microbench/benchmark.html>, 2011.
- [13] Kyoto cabinet: a straightforward implementation of dbm. <http://fallabs.com/kyotocabinet/>.
- [14] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17*, pages 497–514, New York, NY, USA, 2017. ACM.
- [15] Leveldb. <https://github.com/google/leveldb>.
- [16] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An efficient design and implementation of lsm-tree based key-value store on open-channel ssd. In *Proceedings of the Ninth European Conference on Computer Systems, EuroSys '14*, pages 16:1–16:14, New York, NY, USA, 2014. ACM.
- [17] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. NVMKV: A scalable, lightweight, ftl-aware key-value store. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 207–219, Santa Clara, CA, 2015. USENIX Association.
- [18] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. Hikv: A hybrid index key-value store for dram-nvm memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 349–362, Santa Clara, CA, 2017. USENIX Association.
- [19] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning lsms for nonvolatile memory with novelism. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 993–1005, Boston, MA, 2018. USENIX Association.

- [20] H. . P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, Dec 2010.
- [21] Yiming Huai. Spin-transfer torque mram (stt-mram): Challenges and prospects. 2009.
- [22] Intel and micron produce breakthrough memory technology. <https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/>.
- [23] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems, EuroSys '14*, pages 15:1–15:15, New York, NY, USA, 2014. ACM.
- [24] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200, Oakland, CA, 2018. USENIX Association.
- [25] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan. Heteroos - os design for heterogeneous memory management in datacenter. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 521–534, June 2017.
- [26] E. Küay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. Evaluating stt-ram as an energy-efficient main memory alternative. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 256–267, April 2013.
- [27] P. Zuo and Y. Hua. A write-friendly and cache-optimized hashing scheme for non-volatile memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 29(5):985–998, May 2018.
- [28] Slm-db: Single-level key-value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, Boston, MA, 2019. USENIX Association.
- [29] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghuram Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC '10*, pages 143–154, New York, NY, USA, 2010. ACM.
- [30] Lanyue Lu, Thanumalayan Sankaranarayanan Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies, FAST'16*, pages 133–148, Berkeley, CA, USA, 2016. USENIX Association.

- [31] IG Baek, MS Lee, S Seo, MJ Lee, DH Seo, D-S Suh, JC Park, SO Park, HS Kim, IK Yoo, et al. Highly scalable nonvolatile resistive memory using simple binary oxide driven by asymmetric unipolar voltage pulses. In *Electron Devices Meeting, 2004. IEDM Technical Digest. IEEE International*, pages 587–590. IEEE, 2004.
- [32] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. Wort: Write optimal radix tree for persistent memory storage systems. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies, FAST’17*, pages 257–270, Berkeley, CA, USA, 2017. USENIX Association.
- [33] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST’15*, pages 167–181, Berkeley, CA, USA, 2015. USENIX Association.
- [34] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7):786–797, February 2015.
- [35] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD ’16*, pages 371–386, New York, NY, USA, 2016. ACM.
- [36] pmem.io persistent memory programming. <https://pmem.io/>.
- [37] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing dram footprint with nvm in facebook. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys ’18*, pages 42:1–42:13, New York, NY, USA, 2018. ACM.
- [38] Nvmrocks: Rocksdb on non-volatile memory systems. <http://istc-bigdata.org/index.php/nvmrocks-rocksdb-on-non-volatile-memory-systems/>.
- [39] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 363–375, Santa Clara, CA, 2017. USENIX Association.

- [40] Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis, and Igor Zablotchi. Flodb: Unlocking memory in persistent key-value stores. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 80–94, New York, NY, USA, 2017. ACM.
- [41] Pradeep J. Shetty, Richard P. Spillane, Ravikant R. Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building workload-independent storage with vt-trees. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 17–30, San Jose, CA, 2013. USENIX.

Acknowledgements

Many thanks to everybody

