PC-DB: Improving RocksDB via cache and persistent memory

Jiannan Cheng
Shanghai Jiaotong University

Yue Chen
Shanghai Jiaotong University

Zhicheng Wu
Shanghai Jiaotong University

Abstract

With the emergence of byte-addressable persistent memory (PM) and commercially available Intel OPTANE NVDIMM, PM offers new opportunities and research directions for improving the performance of key-value (KV) stores. We introduce a feasible solution to improve RocksDB via DRAM-based cache and persistent memory, which is called PC-DB. We replace the DRAM with PM in RocksDB and redesign the memtable, reducing write overhead and providing stronger consistency and faster recovery due to the absence of Write-Ahead-Log (WAL). To optimize the Optimistic Concurrency Control (OCC) in RocksDB, PC-DB exploits a DRAM-based cache to maintain the sequence number of outstanding transactions, providing validation in memory and avoiding to directly access disk and thus reducing the probability of transaction abort. The cache can also improve read amplification. For Two-Phase Locking (2PL), we simply use the cache to maintain the lock manager. Our expected experimental results show that our solution provides higher read and write performance, faster recovery and lower transaction abort ratio compared with RocksDB.

1 Introduction

Unlike the relational database, the key-value (KV) database does not need to know the data in the value, so it has higher flexibility. The value can be string, file or picture, and the storage content is diverse. Nowadays, KV stores is widely used in various data intensive applications, such as social network [14], e-commerce [2] and network index [1]. Log structured merge trees (LSM-tree) [12] is a common way to implement KV stores, such as BigTable [1], LevelDB [3] and RocksDB [14]. LSM-tree is mainly aimed at write intensive and few query scenarios. The core idea is to give up part of the read performance in exchange for the maximum write performance. Efficient write performance is mainly achieved by constructing a buffer in memory to turn write requests into batch processing, thus turning random writes to disks into sequential writes. In order to ensure the order of data and improve the speed of data access, LSM-tree implements a multi-level data structure, and maintains the data structure through the background thread merge-sort operation (compaction). But it also brings the problem of read and write amplification.

In order to ensure that the system can recover from the failure, Write-Ahead-Log (WAL) is needed to record the operation before writing to memory. After the data is persisted to disk, the log is deleted. Using WAL will cause write performance degradation, so by default, storage systems based on LSM-tree such as LevelDB and RocksDB disable WAL to achieve better performance, but this reduces data consistency, because when crash occurs, some data may be lost.

Because LSM-tree has the problem of read amplification [9, 10, 13], it is necessary to optimize the concurrency control protocol in KV stores based on LSM-tree. For example, in optimistic concurrency control, in order not to access the data of the disk, the verification phase will only be carried out in memory. If the version data is not in memory, the transaction will abort, which improves the abort rate of the transaction.

With the emergence of byte-addressable persistent memory (PM), there occurs new opportunities and research directions for improving the performance of KV stores. PM has recently become more and more involved in key-value stores [7,11,16], which ehances the performance of storage system. PM technologies such as PCM [15], memristors and 3D XPoint [4] promote the rapid development of PM. PM can not only replace disks such as HDDs and SSDs but also be connected via a memory bus and act like DRAM. PM is expected to achieve a comparable read performance with DRAM and the write latency of PM is about 10x higher than DRAM [16], but the cost of PM is lower than DRAM.

In this paper, we use PM to replace the memory components in the RocksDB architecture, and use persistent memory to store memtable and immutable memtable. Because PM has the characteristics of byte-addressability and persistence, it does not need Write-Ahead-Log, reducing the cost of log, and has a stronger consistency. In addition, it has faster recovery speed, because recovery does not need to scan log entry. However, modern CPUs have multiple caches, and in some cases, CPUs may reorder some memory write instructions to improve write performance, which may lead to inconsistent system state. Therefore, we use cacheline flush and memory fence instructions to guarantee the consistency of persistent memory.

In order to optimize the concurrency control of RocksDB, especially Optimistic Concurrency Control (OCC), we use DRAM-based cache to maintain the sequence number of outstanding transactions, so that we can validate the sequence number in DRAM-based cache when the version cannot be retrieved in memory during the validation phase, instead of accessing the disk. Since the sequence number is maintained by cache, it has no need to directly access disk and thus no need to abort transaction, which improves read amplification and reduces the rate of transaction abort.

To summarize, we make the following contributions in this paper.

- We explain the trade-off between performance and consistency in RocksDB and use persistent memory to store memtable and immutable memtable, so it does not require Write-Ahead-Log, which reduces logging costs, has better consistency and faster recovery.

- We show the limitation of Optimistic Concurrency Control in RocksDB, and use DRAM-based cache to improve read amplification and reduce transaction abort ratio.

- Our expected experimental results show that our solution provides higher read and write performance, faster recovery and lower transaction abort ratio compared with RocksDB.

The remainder of this paper is organized as follows. Section 2 gives an introduction on persistent memory technologies for KV store and the structure and features of RocksDB with the limitation of WAL overhead and read amplification, explaining the motivation of our work. Section 3 discusses how to leverage persistent memory in RocksDB. Section 4 presents the design details of DRAM-based cache. Section 5 shows the design of experiments and the expected experimental results. At last, Section 6 discusses the related work and Section 7 concludes the paper.

2 Background and Motivation

In this section, we first discuss the features and background of persistent memory, and the feasibility to replace DRAM with persistent memory. We also present the background and the design of RocksDB [14], providing the limitations of RocksDB.

2.1 Persistent Memory

Persistent Memory, also known as Non-volatile memory(NVM), is a byte-addressable persistent storage device between DRAM and disk in the heterogeneous memory hierarchy. PM can be attached to a memory bus socket just like DRAM, which enables it to be accessed via load and store instructions. Modern PM technologies include phase change memory (PCM) [15], memristors and 3D XPoint [4].

Embracing the feature of byte-addressability, PM achieves a comparable read performance with DRAM. However, the write latency of PM is about 10x higher than DRAM [16], but the cost of PM is lower than DRAM. These properties make PM a suitable choice for replacing DRAM.

Since the PM can be connected via a memory bus with byte-addressable feature, the PM supports atomic writes of 8 bytes [8]. Compared with the traditional storage devices using block access, PM supports more fine-grained writes. When writing persistent data, we need to ensure that the data structure is consistent, even in the event of system crash. However, modern CPUs have multiple caches and in some cases, CPUs may reorder some of the memory write instructions to improve write performance, which may lead to inconsistent system state. To keep the PM write order and data structure consistent, we need to explicitly use instructions such as **MFENCE** and **CLFLUSH** (Intel x86) [6–8] to make memory writes ordered and consistent. In addition, in the case where the size of the data written to the PM is greater than 8 bytes, if the system crashes and performs recovery, the recovered data structure may be partially updated, resulting in inconsistent state. Techniques such as logging and Copy-on-Write (CoW) can be used to handle this situation.

As a new storage device, PM offers new opportunities and research directions for optimizing KV stores. There are previous studies [7, 11, 16] that use PM in KV stores to optimize system performance, which requires to redesign the data structure, such as skiplist [6, 7]. In this work, in addition to redesigning RocksDB's [14] skiplist, we also introduce DRAM-based cache, which works with PM collaboratively to optimize the performance of the system.

2.2 RocksDB

RocksDB [14] is a persistent Key-Value store based on Log Structured Merge Tree (LSM-tree) [12]. The architecture is shown in Figure 1. LSM-tree consists of two parts: memory and Disk. In memory component, in order to improve the write throughput and change the random write to disk into sequential write, RocksDB first constructs a memory table (memtable) buffer in memory to batch write. The memtable is composed of sorted skiplist. When the data of the memtable reaches a threshold, the memtable is set to immutable, and then a new memtable is created to receive the write request. The immutable table will be flushed to disk through back-

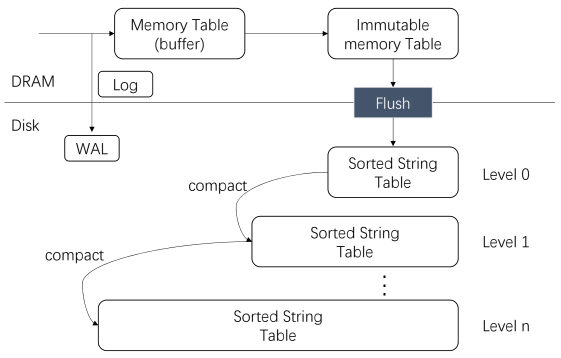

Figure 1: RocksDB architecture.

ground thread. The disk component consists of multi-level sorted string tables (SSTable), from lowest L_0 to highest L_n. Except L_0, each level has one or more sorted SSTable files, where the key ranges of files at the same level do not overlap. The capacity of each level is limited, but the higher level can contain more SSTable files. The capacity of such a level is generally about 10 times larger than that of the previous level. In order to maintain such hierarchical level and data order, when the size of a level exceeds its limitation, the background thread will perform a merge sort operation (compaction) on the level and the next level, and the data of the two levels will be sorted according to the key value. When the two levels have the same key, the value of the lower level will cover the value of the higher level, so as to ensure the uniqueness of each level's data. (except L_0, because immutable memtable is not compacted when it is flushed to disk to increase write throughput, SSTables in L_0 can have overlapping key ranges.) Generally speaking, LSM-tree optimizes the write operation and sacrifices certain read performance. However, due to the multi-level structure and data order, the read operation still maintains good performance.

Write-Ahead-Log overhead In order to ensure that the system can recover from crash, the write operation will be written to Write-Ahead-Log (WAL) before writing memtable. The log will be deleted after the immutable memtable is finally flushed to the disk. This can cause write performance degradation. When using `fsync()` for WAL and inserting 8GB of data with 1KB value size to create database, the write performance is reduced by more than 12 times [6]. Therefore, the trade-off of performance and consistency is involved here. By default, RocksDB does not log operations to get better performance, so some data may be lost when crash occurs.

Read amplification in transaction For transactional support, RocksDB supports both Two-Phase Locking (2PL) and Optimistic Concurrency Control (OCC). For 2PL, RocksDB uses a in-memory dedicated locking manager to maintain the locks, decoupling the records with locks. When accessing a record, the manager should be accessed first to acquire lock.

Because of multi-level structure in LSM-tree, RocksDB has the problem of read amplification [9, 10, 13]. When read operation occurs, it first accesses the memtable, then the immutable memtable. If the data is not in memory, it needs to access the SSTable on disk from the lowest level to the highest level. In each level, it first uses the binary search to find the SSTable where the data is located, after identifying the SSTable, it use another binary search to find the index of the data. Therefore, when retrieving a certain level, at least two binary searches are needed. If the current level cannot retrieve the data, it will go to the next level to search, and repeat the above operations until the data is found, so the overhead of read operation on disk is high. In order to reduce the overhead of accessing the disk, RocksDB uses the Bloom filter to optimize the read operation.

For OCC, verification of records on disk is slow due to the multiple levels on the disk. RocksDB uses the minimum sequence number in memory and global sequence to determine whether to validate in memory. If the sequence is not in memory, the transaction will abort to avoid disk access. In order to optimize the read amplification problem and reduce the probability of transaction abort, we use DRAM-based cache to maintain the sequence number of outstanding transactions, so that we can validate the sequence number in DRAM-based cache when the version cannot be retrieved in memory during the validation phase, instead of accessing the disk.

3 Design and Implementation of PC-DB

This section presents the design and implementation of our work PC-DB, figure 2 shows the overall system architecture of PC-DB. PC-DB makes a trade-off between the performance and the consistency, it leverages PM to store MemTable and Immutable Memtable. In this way, the Memtable and Immutable Memtable are persistent ,thus eliminating the write ahead log(WAL). RocksDB [14] by default disables the WAL in order to provide better performance, so persistent Memtable and immutable Memtable in PC-DB will provide stronger durability and consistency upon system failures.

RocksDB adopts OCC**??** for transaction serialization. For OCC, verification of records on disk is slow due to the multiple levels on the disk. RocksDB uses the minimum sequence number in memory and global sequence to determine whether to validate in memory. If the sequence is not in memory, the transaction will abort to avoid disk access. In order to optimize the read amplification problem and reduce the probability of transaction abort, we use DRAM-based cache to maintain the sequence number of outstanding transactions, so that we only need to validate the sequence number in cache in the validation phase.

To solve this problem, we use DRAM-based cache to store the outstanding txn version, thus improving the read amplification and reducing transaction abort rate.

With the processing of the PC-DB, the DRAM will be

Algorithm 1: Insert(key, value, version,preNode)

```
currentNode:=new Node(key,value,version);
mfence();
currentNode.next=preNode.next;
clflush(currentNode);
mfence();
preNode.next:=currentNode;
clflush(preNode.next);
mfence();
```

full. We should expel the version of the records when txn is finished. PC-DB has some strategies to expel the version of the records. One strategy is to set a timestamp for each record. Once we need to expel some records, we can choose these records which are timeout. There is another strategy in PC-DB to expel records, which is based on the write and read rate of pages. We divide the pages in the DRAM into cold-pages and hot-pages according to the write-read rate of the pages, and if some records need to be expelled, we choose to expel the cold pages.

3.1 Persistent MemTable

In Rock-DB, MemTable is a skiplist stored in DRAM, which is not persistent. It adopts a Write Ahead Log(WAL) for system consistency. WAL includes the key -value pairs ,their version and their checksum,which is sequentially appended in the persistent storage . However, the overheads of WAL become the bottleneck of write operations and it takes rather long time to recover from crash.

Based on the above problems, we adopt persistent memory for optimization. The memtable is stored in persistent memory ,which avoids the overheads of WAL for consistency and durability. The MemTable is a persistent skiplist, which is the same data structure in RocksDB. As we have talked before, the PM support atomic writes of 8 bytes, and in the skiplist, operations like inseration, update, and deletion can be done using an atomic 8 bytes write operation. We adopt $mfence()$ to achieve instruction serialization and $clflush()$ to guarantee that data in cache is flushed into persistent memory and can be recovered under failure. Algorithm 1 shows the insert operation. Other operations such as update and delete can also be achieved by logical insert operation.

Once the system fails, the skiplist can be recovered by locating the root of skiplist, the address of which is stored in the MANIFEST file. The files representing a PM pool is also remaped ,and retrieves the root data structure that stores all pointers to other data structures such as the MemTable, the Immutable MemTable through the support from a PM manager such as the PMDK. In the recovery process, the PC-DB will also flush the Immutable MemTable if there exits one. Because the data has already stored in persistent memory , it will be quick compared with recovering from WAL. To put a KV pair into the PC-DB, PC-DB insert the KV pair along with its transaction id into the MemTable. The KV pair will eventually be flushed to an SSTable in L_0, The KV pair may be compacted and writen to a new SSTable by the compaction of PC-DB, which is the same as RocksDB. To get a value for a given key k from a KV store, PC-DB searches MemTable and Immutable MemTable in order. If k is not found, it will search k in level0 SSTable, level1 SSTable and so on in sequence, until it returns the value of k,and its transaction id, or return null. There is a read amplification in this process, we can use Bloom filter to optimize the read operation.

3.2 DRAM-based Cache

PC-DB uses DRAM as the cache of outstanding transaction version to speed up the version verification in Optimistic Concurrency Control (OCC). This is a space for time strategy. There are two main methods to mix PM and DRAM memory. The first method is to use DRAM as the cache of PM, and the other method is to use PM and DRAM in parallel,as shown in figure 3. In PC-DB, we choose the second method, which uses DRAM as the cache of outstanding transaction version instead of PM.

Take Intel OPTANE DC Persistent Memory [5] as an example. OPTANE DC Persistent Memory offers two mode: Memory mode and App Direct Mode. When configured to memory mode, application and system sense volatile memory pools. DRAM is used to cache the most frequently accessed data, as well as the PM. It is the first way to mix PM and DRAM. When the App Direct Mode is configured, the application and the operating system clearly know that there are two types of load / storage memory in the platform, and which type of data read / write is suitable for DRAM or Intel OPTANE DC Persistent Memory. Using App Direct Mode, we can store the outstanding transaction version in the DRAM, and use PM to store the MemTable and Immutable MemTable.

Figure 4 shows the algorithm of the Optimistic Concurrency Control of RocksDB. During the commit phase, RocksDB validates the sequence number of each record in the read set. If the latest version of the record does not match the version at the time the record was read, the transaction will abort. There is an observation to introduce the DRAM-based cache: during the commit phase, if the latest version has been flushed into the SSTable, we have to retrieve the record in the SSTable, which will affect the performance due to read amplification. In RocksDB, it maintains the minimal sequence number in the memory, and gets the global sequence number when reading a record. When validating, if the earliest sequence number is smaller than the global sequence number, we only need to validate the version in memory. This approach has some weakness. It will abort some transactions which are legal when the record version has been flushed into the SSTable. There are many approaches to solve this problem. For example, we can attach an index to each record flushed to the disk, but the index will occupy much memory space, which DRAM can't afford. In the OCC optimization, in order to reduce the memory footprint, we do not create an index for each record. What we are concerned with is only the outstanding transaction version, so we can just cache the outstanding transaction version. In PC-DB, DRAM is introduced as the cache. We use the map data structure to store the outstanding transaction version, and we use the App Direct Mode of the Intel OPTANE DC Persistent Memory to store the map in the DRAM. After PC-DB commits a transaction, it should update or add the corresponding records‘ versions in the DRAM. To validate the version in the read set during the commit phase in OCC, when the version cannot be retrieved in the persistent memory, PC-DB will retrieve the key in the DRAM-based cache and obtain the corresponding version if the key exists in the DRAM. However, if the key is not in the DRAM, PC-DB will set the transaction retry after a period of time, and use a background thread to fetch the lasted version of the key from the disk, rather than just abort the transaction.

3.3 DRAM Buffer Management

If PC-DB commit a transaction, it will add the new records‘ version in the DRAM. What if the DRAM is full? There are several strategies to expel the records in the DRAM. One very straight way is to expel the records of a transaction as soon as the transaction is committed. However, if there are other transactions operating on the same records in the meanwhile, they has to wait for the background thread to fetch the versions of the records back to the DRAM, which will take rather a long time. Also, we can add a reference counter to each of these records in the DRAM. If one transaction makes operations on the record, the reference counter will add one, at the same time, if one transaction is committed, the reference counter of the records it operates on will subtract one. When we need to expel some records, we can expel the records whose reference count is zero. However, this method will transform the read operation into a write operation, which costs a lot. To solve this problem, PC-DB has two approaches. One straight and effective way is to add a valid period for each of the record in the DRAM. When we need to expel some records, we choose these records who are time out.

To manage data buffer, there are many algorithms. The most commonly used algorithm is the LRU algorithm that makes a simple assumption for all the data accesses: if a data block is accessed once, it will be accessed again. This locality metric is also called "recency", which is implemented by a LRU stack. Each data access is recorded in the LRU stack, where the top entry is the most recent accessed (MRU) and the bottom entry is the least recent accessed (LRU). The LRU entry is the evicted item as the buffer is full (reflected by the full LRU stack). If data access pattern follows the simple assumption of LRU, the strong locality data set is well kept in the buffer by LRU. However, the LRU algorithm fails to handle the following three data access patterns: (1) each data block is only accessed once in a format of sequential scans. In this situation, the buffer would be massively polluted by weak or no locality data blocks. (2) For a cyclic (loop-like) data access pattern, where the loop length is slightly larger than the buffer size, LRU always mistakenly evicts the blocks that will be accessed soon in the next loop. (3) In multiple streams of data accesses where each stream has its own probability for data re-accesses, LRU could not distinguish the probabilities among the streams.

Our second approach to evict records in the DRAM is based on the LIRS algorithm, which performs better than the LRU algorithm. The pages in the DRAM are divided into two kinds: the cold pages and the hot pages. The division is based on the recent operation frequency. When a new record needs to enter into the DRAM but there is no free space, PC-DB chooses a cold page and migrate the page that contains the new records to replace the cold page. In order to divide the pages in DRAM into cold pages and hot pages, LIRS maintains two sets: High Inter-reference Recency set and Low Inter-reference Recency set. Low Inter-reference Recency set contains the pages that are operated on (read and write) frequently within a period of time, While High Inter-reference Recency set contains the cold pages. In the LIRS algorithm, two parameters, IRR(inter-reference recency) and Recency, are used . IRR is the last two visit intervals of a page, and Recency is how many other pages have been visited since the last visit of the page. The IRR and Recency parameters do not contain the number of duplicate pages because the repeated calculation of the page does not have much effect on the priority of current page. The division of High Inter-reference Recency set and Low Inter-reference Recency set is based on the IRR, and if two pages have the same IRR, the page with the larger Recency is replaced. LIRS algorithm use stack S and list Q to manage the two set. Stack S is used to maintain the hot pages and the potential hot pages, and List Q is used to link all the cold pages. In this way, the three LRU issues, which are talked above, are addressed.

3.4 Cache Coherence

In a multi-core CPU, each core has its own cache. There may be copies of the same data in the cache of both cores, which may lead to data inconsistency when the two cores modify data alone. Therefore, we need to solve the cache consistency problem.In order to solve the problem of cache inconsistency, we need a mechanism to constrain each core, that is, cache consistency protocol.

The commonly used cache consistency protocols are all snooping protocols. Each core can monitor the status of itself and other cores at any time, so as to unify management and coordination. The idea of snooping is: each cache of CPU is independent, but the memory is shared. All cached data is finally written into the same memory through the bus. Therefore, each CPU core can "see" the bus, that is, each cache not only accesses the bus during memory data exchange, but also "snoops" the bus at all times to monitor what other caches are doing. Therefore, when a cache writes data to memory, other caches can also "snoop" to ensure the synchronization between caches according to the consistency protocol.In PC-DB,we simply use the most commonly used cache coherence protocol: MESI protocol

MESI protocol is a common cache consistency protocol, which guarantees cache consistency by defining a state machine. There are four states in the MESI protocol, which are all for cache lines (the cache consists of multiple cache lines, and the size unit of the cache line is related to the number of bits of the machine).

- (I) Invalid state: cache row invalid state. Either the cached row data is out of date, or the cached row data is no longer in the cache. For invalid state, it can be directly considered that the cache row is not loaded into the cache.
- (S) Shared status: cache row sharing status. The cache row data is consistent with the corresponding data in memory, and the corresponding cache rows in multiple caches are shared. In this state, the cache row can only be read and not written.
- (E) Exclusive state: the state unique to the cache row. The data in this cache row is consistent with the corresponding data in memory. When a cache row is in a unique state, the corresponding cache rows of other caches must be in an invalid state.
- (M) Modified status: the status of the cache row has been modified. The data in the cache row is dirty data, which is inconsistent with the corresponding data in memory. If a cache behavior has modified the state, the corresponding cache lines in other caches must be in an invalid state. In addition, if the cache row state in this state is modified to be invalid, the dirty segment must be written back to memory first.

The law of MESI protocol: after all cache rows (dirty data) in M state are written back, the data of cache rows in any cache level is consistent with memory. In addition, if a cache row is in the e state, it will not exist in other caches.

The MESI protocol guarantees the strong consistency of cache and provides the complete sequence consistency in principle. It can be said that under the memory model implemented by the MESI protocol, the cache is absolutely consistent, but this will also cause some efficiency problems.

4 Evaluation

A paragraph of text goes here. Lots of text. Plenty of interesting text. Text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text. More fascinating text. Features galore, plethora of promises.

5 Related Work

A paragraph of text goes here. Lots of text. Plenty of interesting text. Text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text. More fascinating text. Features galore, plethora of promises.

6 Conclusion

A paragraph of text goes here. Lots of text. Plenty of interesting text. Text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text. More fascinating text. Features galore, plethora of promises.

Acknowledgments

This work is the course assignment of Computer System Design and Implementation, which is completed under the guidance of instructor Zhaoguo Wang.

References

[1] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A distributed storage system for structured data (awarded best paper!). In Brian N. Bershad and Jeffrey C. Mogul, editors, *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 205–218. USENIX Association, 2006.

[2] 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 Thomas C. Bressoud and M. Frans Kaashoek, editors, *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 205–220. ACM, 2007.

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

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

[5] Intel. Intel optane persistent memory. https://software.intel.com/en-us/persistent-memory.

[6] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. Slm-db: Single-level key-value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 191–205, Boston, MA, February 2019. USENIX Association.

[7] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 993–1005. USENIX Association, 2018.

[8] 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 Geoff Kuenning and Carl A. Waldspurger, editors, *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 - March 2, 2017*, pages 257–270. USENIX Association, 2017.

[9] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. In Angela Demke Brown and Florentina I. Popovici, editors, *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*, pages 133–148. USENIX Association, 2016.

[10] Leonardo Mármol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. NVMKV: A scalable, lightweight, ftl-aware key-value store. In Shan Lu and Erik Riedel, editors, *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*, pages 207–219. USENIX Association, 2015.

[11] NVMRocks. Rocksdb on non-volatile memory systems. http://istc-bigdata.org/index.php/nvmrocks-rocksdb-on-non-volatilememory-systems/.

[12] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, 1996.

[13] 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, Shanghai, China, October 28-31, 2017*, pages 497–514. ACM, 2017.

[14] RocksDB. Rocksdb. https://rocksdb.org/.

[15] H.-S. Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P. Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E. Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, 2010.

[16] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In Dilma Da Silva and Bryan Ford, editors, *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, pages 349–362. USENIX Association, 2017.