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. Our expected evaluation results show that PC-DB provides higher read and write throughput, 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 [?], e-commerce [?] and network index [?]. Log structured merge trees (LSM-tree) [?] is a common way to implement KV stores, such as BigTable [?], LevelDB [?] and RocksDB [?]. 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 guarantee 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 [?, ?, ?], 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 [?, ?, ?], which ehances the performance of storage system. PM technologies such as PCM [?], memristors and 3D XPoint [?] 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 [?], 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 ratio 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 evaluation results show that PC-DB provides higher read and write performance, faster recovery and lower transaction abort ratio compared with RocksDB.

The rest 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 and presents the design details of DRAM-based cache. Section 4 shows the design of experiments and the expected experimental results. At last, Section 5 discusses the related work and Section 6 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 [**?**], 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) [**?**], memristors and 3D XPoint [**?**].

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 [**?**], 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 [**?**]. Compared with the traditional storage devices using block access, PM supports more fine-grained writes. When writing persistent data, we need to guarantee 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) [**?**, **?**, **?**] to make memory writes ordered and consistent. In addition, in the case where 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 [**?**, **?**, **?**] that use PM in KV stores to optimize system performance, which requires to redesign the data structure, such as skiplist [**?**, **?**]. In this work, in addition to redesigning RocksDB's [**?**] skiplist, we also introduce DRAM-based cache, which works with PM collaboratively to optimize the performance of the system.

2.2 RocksDB

RocksDB [**?**] is a persistent Key-Value store based on Log Structured Merge Tree (LSM-tree) [**?**]. The architecture is shown in Figure **??**. 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. If the data contained in memtable reaches the threshold value, 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

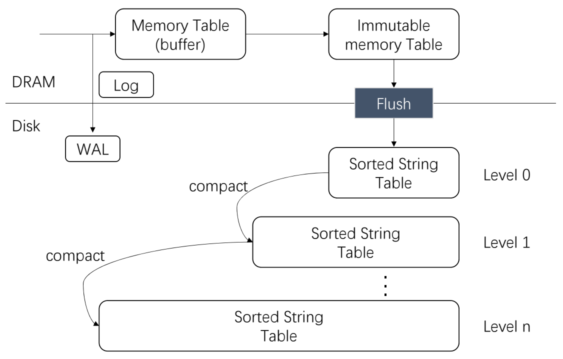


Figure 1: RocksDB architecture.

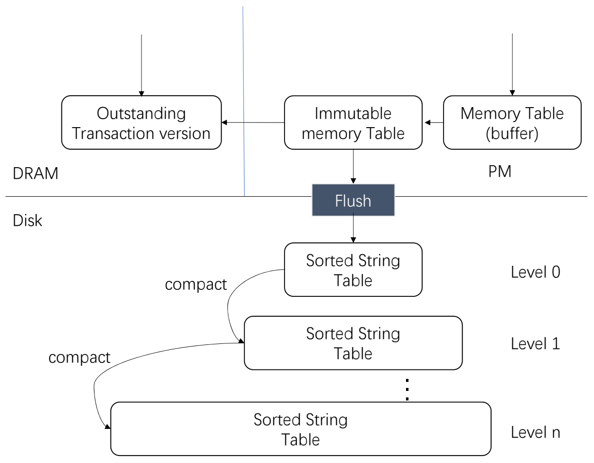


Figure 2: PC-DB architecture.

through background thread. The disk component consists of multi-level sorted string tables (SSTable), from lowest L_0 to highest L_n. Except L_0, there are one or more SSTable files at each level, 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 typically about 10 times larger than the previous level. In order to maintain such hierarchical level and data order, when the size of a level reaches 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 [**?**]. 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 [**?**, **?**, **?**]. 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 **??** 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 [?] 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 PM and DRAM in the validation phase instead of serching disk.

3.1 Persistent MemTable

In RocksDB, the 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 ,version and 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 since it scans long WAL for recovery.

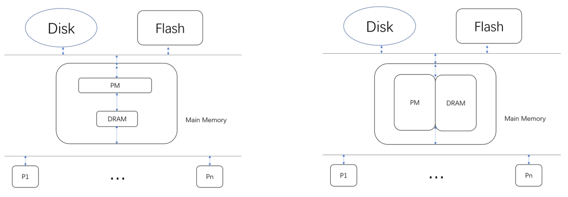
Based on the above problems, we adopt persistent memory for optimization. The Memtable is stored in persistent memory ,which avoids the overheads of WAL for consistency and durability. The MemTable is a persistent skiplist, which is the same data structure in RocksDB. As we have talked before, the PM support atomic writes of 8 bytes, and in the skiplist, operations like inseration, update, and deletion can be done with a logical inseration 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.

To put a KV pair into the PC-DB, PC-DB inserts the KV pair along with its version into the MemTable. When the MemTable is full,PC-DB will make it immutable, and create a new MemTable. The KV pairs in the Immutable MemTable will be flushed to the SSTable in L_0 and compacted to the SSTable in the lower level , which is the same as RocksDB.

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 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.

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();
```



(a) DRAM is the cache of PM

(b) Use DRAM and PM in parallel

Figure 3: Structure of two PM and DRAM mixing methods

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 [?] 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 determine which type of data read / write is suitable . Using App Direct Mode, we can store the outstanding transaction version in the DRAM and store the MemTable and Immutable MemTable in PM.

Figure 4 shows the algorithm of the Optimistic Concurrency Control of RocksDB. During the commit phase, RocksDB checks 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 be aborted. 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 store the location of each record flushed to the disk in memory, but it will occupy much memory space, which DRAM can't afford. In the OCC optimization phase, in order to reduce the memory consumption, 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 recordsversions in the DRAM. We use DRAM to store the outstanding transaction versions, but there is another problem when validating the version in the validation phase of OCC: when validating, should PC-DB first search the DRAM for the latest version of records, or should it first search the PM for the version. If the PC-DB first search the DRAM for the latest version, we should make sure the version of the records in the DRAM must be the latest all the time. In such situation, when a transaction wants to commit, it should first insert or update the version of the records in the DRAM, and then commit .In this period, it should get a lock to avoid other transactions update the versions of the record at the same time. Also, to avoid the situations that if a transaction is aborted after it update the versions of the records and before it commits, PC-DB should keep the log, and recover the version in the DRAM based on the log when the transaction is aborted at this time. The lock and the log will be the performance bottleneck if there are many transactions racing in the system. Thus, the validation strategy of the OCC in PC-DB is: 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 the function *commit*() 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.

```
globalSeq (+1 once a transaction commits)

Record: <key, value, seq>              Commit()
Txn: <tid, rset, wset>                   Sort & Lock records in rs & ws
                                         for each <r, seq> in rset
Get(key_t k)                               if r.seq != seq
  r <- store.get(k)                          Abort()
  put <r, r.seq> into rset
                                         tid = globalSeq;
                                         globalSeq = tid + 1;
Write(key_t k, val_t v)
    put <k, v> into wset                 for each <k, v> in wset
                                           store.put(k, v, tid);
                                         Unlock all
```

Figure 4: Algorithm of the Optimistic Concurrency Control

3.3 DRAM Buffer Management

If PC-DB commits a transaction, it will add the new records' version in the DRAM. Once the DRAM is full, there are several existing strategies to evict the records in the DRAM. One very straight way is to evict the records as soon as the accessing 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 evict 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 evict some records, we choose these records who are time out.

Our second approach to evict record is based on the the page swap policy. To manage data buffer, there are many algorithms. The most commonly used algorithm is the LRU algorithm that is based on the locality for all the data accesses: if some data is accessed once, it will be accessed again with high possibility. We can use the LRU stack to store the information of locality, which is also called "recency". When some data is accessed, the LRU stack will record the data access information. In the LRU stack, the entry in the top is the most recent accessed and the entry in the bottom is the least recent accessed (LRU). Thus if we want to evit some entries, the entries in the bottom of the LRU stack will be evicted. If we use this data access pattern, the locality will be made good use of. However, the LRU page swap policy also has some shortcommings: (1)when we make a sequential scans, every record will be accessed only once. In such a situation, there is no locality, because we won‘t access a record again after we access it. (2) Sometimes we can access some blocks of data again and again. In this data access pattern, where the data we access in one route is larger than the buffer size, the LRU page swap policy will always evicts the blocks that will be accessed soon in the next loop. (3) There can be multiple processes in the machine, and each of them has its own probability for data re-accesses, LRU can not know the possibility of their data access pattern.

Our second approach to evict records in the DRAM uses 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.

4 Evaluation

4.1 Methodology

In evaluation, we use a Dell r740 server with two Intel Xeon E5-2640V3 processors(2.6Ghz), 64GB DRAM, and Intel SSD of 480GB. For PM, we use Intel Optane DC 256 GB Persistent Memory Module, and for the operation system, we use Ubuntu 18.04LTS with Linux kernel version 4.15 is used. We run PC-DB and RocksDB in this platform by contrast. In PC-DB, we use the APP Direct Mode of Intel Optane DC Persistent Memory Module. In RocksDB and our PC-DB, the MemTable size in RocksDB and PC-DB is set to 64MB, and the key size is set to a fixed size: 20 KB, and all SSTable files are stored in the SSD in both database. To evaluate the performance of the two database, we use the db_bench benchmarks as microbenchmark and the YCSB as real world workload benchmarks.

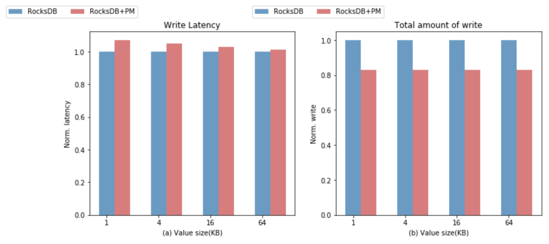


Figure 5: Random write performance comparison

4.2 Using a Persistent MemTable

Generally our design can be divided into two parts: to get better consistency in exchange of performance, we use PM to replace DRAM; and to improve the read amplification in the validation phase in OCC, we use DRAM to store the outstanding transaction version. In order to better understand what is the effect of the two strategy, we first evaluate the performance of RocksDB, a modified version of RocksDB by replacing the DRAM using PM, and the PC-DB. Figure 5 shows the expected performance of this evaluation for the random write workload form db_bench over various value sizes. The write latency and total amount of data written to disk is normalized to these of RocksDB. In the figure of write latency, the write latency of RocksDB is similar to that of RocksDB+PM. This is because 1)the write and read latency of PM is higher than the DRAM. 2) RocksDB have to flush WAL to the disk constantly, reducing the performance of RocksDB. However, RocksDB+PM can persistent the inserted data as soon as it is inserted to MemTable, and it can also provide stronger durability. As for the figure of total amount of write, RocksDB+PM can reduce the amount of write data by about 16% compared to the result of RocksDB. This is because RocksDB+PM don‘t have to write the WAL.

4.3 Result with Microbenchmarks

Figure 6 shows the operation throughputs of random write and random read workloads of PC-DB, and all of them are normalized to these of RocksDB. For the random write operations, PC-DB provides almost the same throughput as the RocksDB over all the value sizes. As we have talked before, the write performance of RocksDB is almost the same as RocksDB+PM, and the introducing of DRAM has almost no effect of write performance, because we just need to insert or update the version of the records when a transaction is committed. This overhead of this operation is small. For the random operations, PC-DB also shows a similar performance as RocksDB. This because the read latency of PM is almost the same as that of DRAM, and if PC-DB can not get the key in PM, it will search the disk for the recoed, just like the RocksDB.

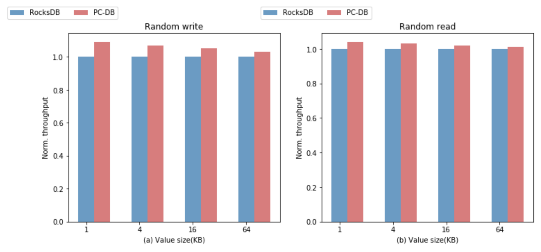


Figure 6: Throughput of PC-DB normalized to RocksDB

4.4 Results with YCSB

YCSB consists of six workloads. Note that in workload A E, write operation has a more share in the total operations. In figure $7(a)$ and $7(b)$, the throughput of PC-DB are higher than those of RocksDB and the latency of PC-DB is smaller than the RocksDB for all workloads, which is because of our using of DRAM as the cache. In the validation phase of OCC, we first search the PM for the latest version of a record, and if it is not found, we search it in the DRAM. While in RocksDB, we have to search the disk for the version if it is not found in the DRAM, which will cost a lot. We can also notice that if the write operations have more shares in the total operations, the PC-DB performs better. In figure $7(c)$, the total amount of write in PC-DB is much smaller than that of RocksDB for all workloads. this is the same reason as the 5.2: we don‘t have to flush WAL in PC-DB.

5 Related Work

Previous work, such as skimpystash [**?**] and SILT [**?**] ,design KV storage for SSD. Skimpystas uses flash as the intermediate cache to store append-only logs, so as to benefit from the high sequential write performance of SSD. SILT reduces the use of DRAM and maintains a sorted log index in memory by splitting logs across DRAM. To conclude, these work has increased throughput through batch processing and adding software layers, thus enhancing sequencing.

Some previous work has redesigned LSMs for SSD. Wisckey [**?**] redesigned LSMs to reduce read / write amplification and use SSD bandwidth. VT-tree [**?**] tree design proposes a file system and user level key value storage, which is suitable for workload independent storage. Wang et al. [**?**] exposed I / O channel information of SSDs to leveldb to utilize parallel bandwidth.

NoveLSM [**?**] is a persistent KV storage system based on LSM, which aims to utilize non-volatile memory and provide low latency and high throughput for applications. It stores the the MemTable and Immutable MemTable in both DRAM and PM for performance and better recovery.

performance.png performance.bb

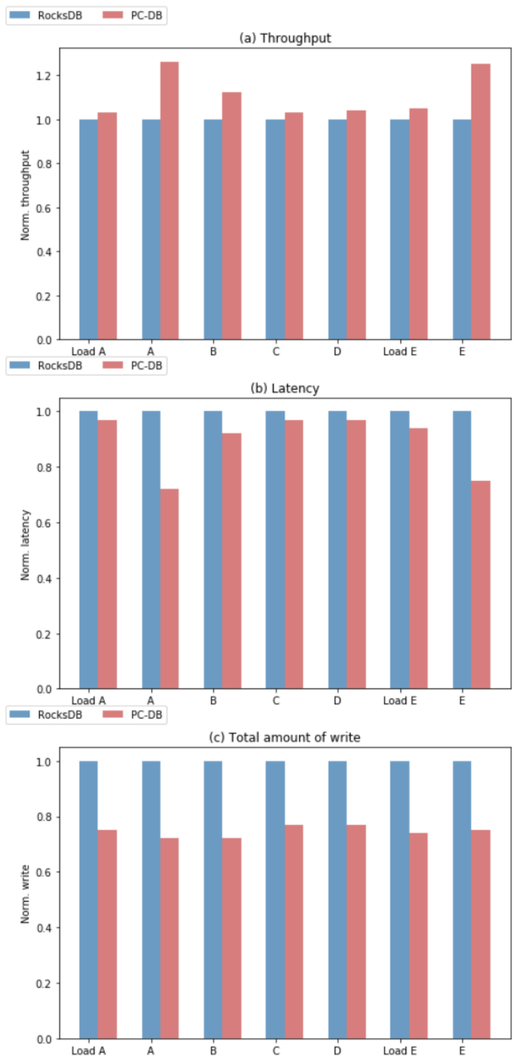


Figure 7: YCSB performance

6 Conclusion

In this paper, we presented the Persistent-Consistent DB(PC-DB) that use PM to store the MemTable and Immutable MemTable, thus providing faster recovery and better consistency. At the same time, PC-DB make use of the DRAM as a cache for outstanding transaction version, solving the problem of the read amplification in the validation phase of OCC. Our experiments shows that PC-DB performs well in the YCSB, which are real world workload benchmarks.

Acknowledgments

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