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. 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, our solution 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 rate 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 [13], e-commerce [2] and network index [1]. Log structured merge trees (LSM-tree) [11] is a common way to implement KV stores, such as BigTable [1], LevelDB [3] and RocksDB [13]. 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 [8, 9, 12], 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 [6,10,15], which ehances the performance of storage system. PM technologies such as PCM [14], 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 [15], 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 recov-

ery 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 only need to validate the sequence number in cache in the validation phase. 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 rate.
- Our expected experimental results show that our solution provides higher read and write performance, faster recovery and lower transaction abort rate 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 [13], 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) [14], 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 [15], 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 [7]. 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) [5–7] 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 crashs 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 [6, 10, 15] that use PM in KV stores to optimize system performance, which requires to redesign the data structure, such as skiplist [5, 6]. In this work, in addition to redesigning RocksDB's [13] skiplist, we also introduce DRAM-based cache, which works with PM collaboratively to optimize the performance of the system.

2.2 RocksDB

RocksDB [13] is a persistent Key-Value store based on Log Structured Merge Tree (LSM-tree) [11]. 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-

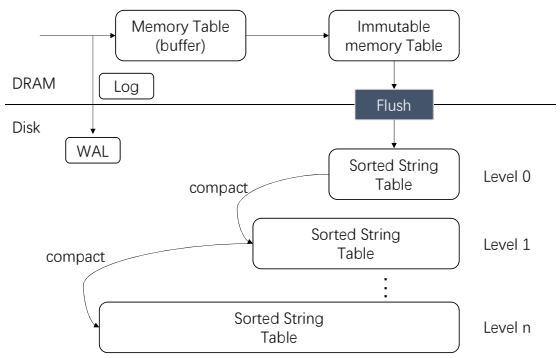

Figure 1: RocksDB architecture.

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 [5]. 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 [8, 9, 12]. 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 only need to validate the sequence number in cache in the validation phase.

3 Footnotes, Verbatim, and Citations

Footnotes should be places after punctuation characters, without any spaces between said characters and footnotes, like so.[1] And some embedded literal code may look as follows.

```
int main(int argc, char *argv[])
{
    return 0;
}
```

Now we're going to cite somebody. Watch for the cite tag. Here it comes. Arpachi-Dusseau and Arpachi-Dusseau coauthored an excellent OS book, which is also really funny [**?**], and Waldspurger got into the SIGOPS hall-of-fame due to his seminal paper about resource management in the ESX hypervisor [**?**, 5].

The tilde character (~) in the tex source means a non-breaking space. This way, your reference will always be attached to the word that preceded it, instead of going to the next line.

And the 'cite' package sorts your citations by their numerical order of the corresponding references at the end of the paper, ridding you from the need to notice that, e.g, "Waldspurger" appears after "Arpachi-Dusseau" when sorting references alphabetically [**?**, **?**].

[1] Remember that USENIX format stopped using endnotes and is now using regular footnotes.

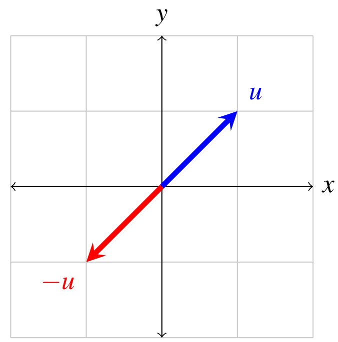

Figure 2: Text size inside figure should be as big as caption's text. Text size inside figure should be as big as caption's text. Text size inside figure should be as big as caption's text. Text size inside figure should be as big as caption's text. Text size inside figure should be as big as caption's text.

It'd be nice and thoughtful of you to include a suitable link in each and every bibtex entry that you use in your submission, to allow reviewers (and other readers) to easily get to the cited work, as is done in all entries found in the References section of this document.

Now we're going take a look at Section 4, but not before observing that refs to sections and citations and such are colored and clickable in the PDF because of the packages we've included.

4 Floating Figures and Lists

Here's a typical reference to a floating figure: Figure 2. Floats should usually be placed where latex wants then. Figure2 is centered, and has a caption that instructs you to make sure that the size of the text within the figures that you use is as big as (or bigger than) the size of the text in the caption of the figures. Please do. Really.

In our case, we've explicitly drawn the figure inlined in latex, to allow this tex file to cleanly compile. But usually, your figures will reside in some file.pdf, and you'd include them in your document with, say, \includegraphics.

Lists are sometimes quite handy. If you want to itemize things, feel free:

fread a function that reads from a `stream` into the array `ptr` at most `nobj` objects of size `size`, returning returns the number of objects read.

Fred a person's name, e.g., there once was a dude named Fred who separated usenix.sty from this file to allow for easy inclusion.

The noindent at the start of this paragraph in its tex version makes it clear that it's a continuation of the preceding paragraph, as opposed to a new paragraph in its own right.

4.1 LaTeX-ing Your TeX File

People often use `pdflatex` these days for creating pdf-s from tex files via the shell. And `bibtex`, of course. Works for us.

5 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 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. More fascinating text. Features galore, plethora of promises.

Acknowledgments

The USENIX latex style is old and very tired, which is why there's no \acks command for you to use when acknowledging. Sorry.

Availability

USENIX program committees give extra points to submissions that are backed by artifacts that are publicly available. If you made your code or data available, it's worth mentioning this fact in a dedicated section.

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] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.