M.S. THESIS

Write Behind Logging(WBL) : Optimizing LSM-Tree based Key-Value Store with Persistent Memory

Write Behind Logging(WBL) : 영구 메모리를 활용한 LSM 트리 기반의 키-밸류 저장 최적화

August 2022

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Yoonseo Choi

M.S. THESIS

Write Behind Logging(WBL) : Optimizing LSM-Tree based Key-Value Store with Persistent Memory

Write Behind Logging(WBL) : 영구 메모리를 활용한 LSM 트리 기반의 키-밸류 저장 최적화

August 2022

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Yoonseo Choi

Write Behind Logging(WBL) : Optimizing LSM-Tree based Key-Value Store with Persistent Memory

Write Behind Logging(WBL) : 영구 메모리를 활용한 LSM트리 기반의 키-밸류 저장 최적화

지도교수 염헌영

이 논문을 공학석사 학위논문으로 제출함

2022 년 7 월 5 일

서울대학교 대학원

전기·컴퓨터 공학부

최윤서

최윤서의 공학석사 학위논문을 인준함

2022 년 7 월 1 일

위 원 장	이 재 진	(인)
부위원장	염 헌 영	(인)
위 원	김 진 수	(인)

Abstract

Current LSM-tree based key-value database systems are optimized for the DRAM-SSD memory architecture. With the emergence of persistent memory(PMEM), which is a non-volatile storage device with performance close to DRAM, memory architecture have changed. In this new hybrid memory architecture, we need to optimize database system to exploit the benefits given by persistent memory. LSM-tree based key-value database systems suffer from (1)Write Ahead Logging and (2)Write stalls from compaction jobs. In this paper, we separate read/write path of the system. First, we write data on persistent memory directly, eliminating the need of write ahead logging. Since PMEM provides larger space than DRAM, concerns for compaction stalls diminish as well. Data written on persistent memory is then forwarded to DRAM. Since DRAM is free of compaction jobs and write path, there are some flexibility of customization to improve read performance. Our work was made above

vanilla RocksDB. We evaluate our work by using YCSB workloads, comparing with original version of RocksDB system. The result showed that our system works well on write-intensive workloads, and showed little improvement on read-intensive workloads.

Keywords: Persistent Memory, LSM-Tree, Key-Value Database, Write Stalls, Reducing Log Overhead, Separate Read/Write Path

Student Number: 2019-20797

Contents

List of Figures

Chapter 1

Introduction

Current database systems are well developed for DRAM-SSD memory architecture, and it is optimized in many ways under this hardware layout. With the emergence of Intel Optane Persistent Memory(PMEM) [1] however, memory hierarchy have changed. Since PMEM have middle performance level in between DRAM and SSD, it is desirable to use PMEM as middle layer between DRAM and SSD rather than replacing either one of them, as shown in Figure 1.1. With persistent memory adopted, new opportunities for optimizations have appeared. While there are various kinds of database systems, our work focus on LSM-Tree based key-value database system, such as RocksDB, LevelDB and Cassandra.

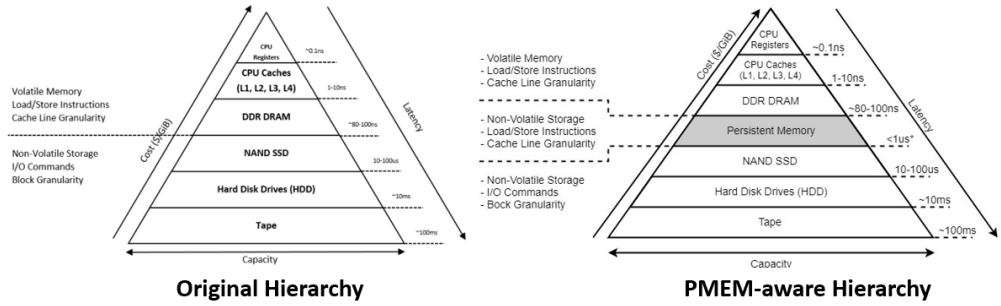


Figure 1.1 Memory Hierarchy

In this paper, we analyze the details of performance characteristics of PMEM and workflow of LSM-tree to find some way to exploit them.

1.1 Motivation

LSM-tree is a data structure that performs well on write-intensive workloads. It was originally designed for DRAM-SSD like architecture, which have fast volatile memory and slow stable storage. Insertion of a data in LSM-tree works in append-only way, which means that write can be done in average O(1) time. To provide decent read latency, LSM-tree maintains each nodes as sorted run which is merged and flushed down to bottom level as each node grows to full size. This work of maintaining the structure creates an overhead. Also, writing

data down to non-volatile memory means that there is a need for some methods to provide atomicity and consistency of data. This is handled by Write Ahead Logging(WAL) policy, which is also an overhead, since writing actual key-value pair can't be done until log is written to slow and stable storage.

Since the commercial release of Intel DC Persistent Memory Module in 2019 [1], there were many studies to reduce down the overheads explained above by utilizing the benefits of persistent memory. Some of them used persistent memory as a log storage. This strategy provides faster write by reducing WAL overhead. [2] Other works have used PMEM as container for data along with DRAM. [3] With this strategy, database system have less threats of write stalls because there is more likely to have additional room to write. Our work adopted an idea of using PMEM as an additional container for data and expanded it. In our work, PMEM works alone as a write buffer to disk without DRAM. By avoiding writes to volatile memory, logging won't be necessary. Also, we use DRAM as a read buffer, which makes an opportunity to optimize for faster read.

1.2 Contribution

Our work was implemented on vanilla version of RocksDB [4]. Our design goal was to successfully separate read and write path of data. In this design, PMEM carries data within a instance called MemTable. Immutable Memtables, which is an instance copied from MemTable when it becomes full, also resides in PMEM. Compaction and flush jobs operate within PMEM and SSD, not involving DRAM. Data in PMEM is copied to DRAM, and DRAM behaves as a read buffer. When read request is handled, DBMS looks at DRAM first, and then PMEM and disks. This separation of read and write path gives an opportunity to customize the read workflow. Which means that it is easier to adopt new ideas, such as changing data structure that is read-favorable, such as B-Tree.

We compared our modified design of RocksDB with the vanilla version with YCSB workloads. [5] The results showed that it performed 8% - 12% better on write-intensive workloads, and about 1% - 3% better on read-intensive workloads. Also, we implemented our idea on LevelDB, which is premature version,

to compare with NoveLSM [3]. Our result in levelDB showed about 30% better performance on write-intensive workloads.

1.3 Outline

This paper have structures like below :

- **Chapter 2** summarizes the background about the performance characteristics of Intel Optane DC Persistent Memory Module [1], how LSM tree based key-value database storage engine works, and related works that optimized LSM-tree with persistent memory.

- **Chapter 3** provides the reasons why we designed the system in this way and how we implemented them in detail.

- **Chapter 4** introduces the experimental setup and analyze the result of experiments.

- **Chapter 5** summarizes and concludes our work. It also points out the directions for future work.

Chapter 2

Background

This research is about optimizing LSM tree based key-value database under an assumption that memory hierarchy will be changed with persistent memory device. We used Intel Optane DC Persistent Memory Module device for an experimental setup. In this chapter, we explain (1)the performance characteristics of Intel Optane Persistent Memory device, (2)the workflow of RocksDB's LSM-tree based database systems, and (3) related works that our work referred to.

2.1 Intel Optane Persistent Memory

Intel DC Persistent memory is a persistent storage device that provides read and write latency close to DRAM. Many studies have shown the details of performance characteristics of PMEM since 2019, when it was commercially released. There are 4 major features of PMEM : 1.Non-volatility; which means that the data is not lost on power crash. 2. Faster than SSD, slower than DRAM; PMEM is approximately 20-40 times faster than SSD and 5-10 times slower than DRAM. 3.Byte Addressablilty : the data moves with byte unit, unlike solid state disk or hard disk drive where data moves with a block unit. However as shown in figure 2.1 PMEM performs best at 256B granularity, so programmers should be aware of this. 4. Higher capacity than DRAM; PMEM offers greater memory capacity per socket than DRAM, which of course is in lower cost.

Granularity	Workload traffic	Bandwith
256GB	Read 100%	8.3GB/s
256GB	Write 100%	3.0GB/s
256GB	Read 67% Write 33%	5.4GB/s
64GB	Read 100%	2.13GB/s
64GB	Write 100%	0.73GB/s
64GB	Read 67% Write 33%	1.35GB/s

Figure 2.1 Performance of Intel Optane DC 200 Persisent Memory Module(256GB)

2.2 LSM Tree based Key-Value Database

In modern database systems, B-Trees and LSM-Trees are data structures most commonly used for key-value storage engines. These two data structures have different optimal characteristics; While B-Trees work better on read-intensive workloads, such as short-range queries, LSM-Trees, on the other hand, work better on write-intensive workloads, such as update queries. This different behavior comes from the difference that LSM-Trees have exponentially growing size of node on bottom level, like in Figure 2.2. B-Trees, however, have same bounded size node at every level(height) of tree. In B-Trees, each node have a

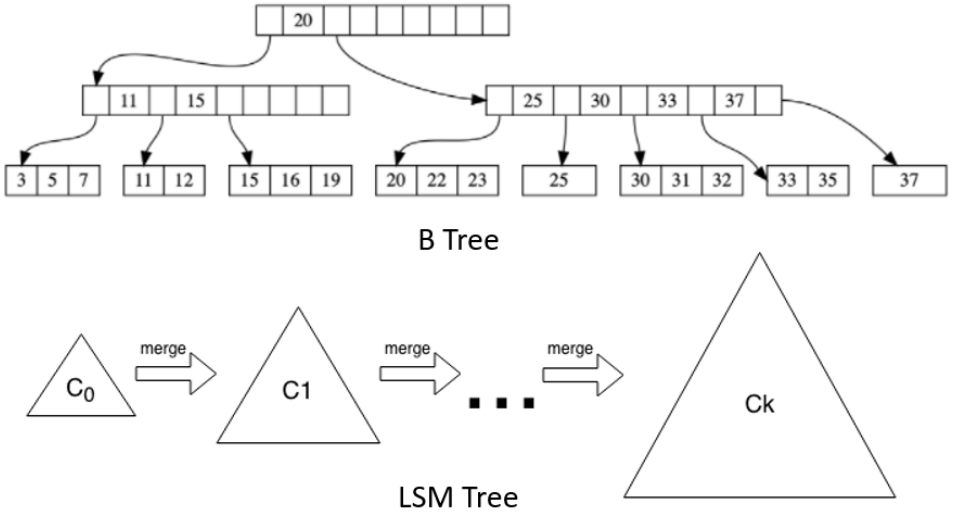


Figure 2.2 structure of B-Tree and LSM-Tree

fixed size, which is 4 in Figure 2.2. This structural difference results in different read and write latency. B-tree's average insert and search time takes O(logn) time, because height of the tree is dominant factor. LSM Tree keeps a append-only structure on top level, which makes O(1) average write time. As shown in Figure 2.2, LSM-Tree merges down to bottom level as they grow in size. Since bottom level keeps larger data, read takes average O(n) time to finish. There are some methods to overcome poor read performance such as bloom filters and fence pointers, yet these are not our concern.

Whenever an insertion query arrives, database system handles it by stor-

ing a key-value pair in a in-memory write buffer called "MemTable". When MemTable size grows and becomes full, with size is predetermined by user or default setting, it is turned to an read-only in-memory buffer called "Immutable MemTable. Several Immutable Memtables, which number is also predetermined by user or default setting, are merged and flushed down to disk/SSD as a "Sorted Strings Table"(SST) data file. As shown in figure, SSTs are organized as a sorted run and compose a level. When level n SSTs become full, they are merged and flushed to level n+1.

The major performance bottleneck in LSM tree based database systems is a write stall due to compaction and flush jobs. This happens when the rate of write request overwhelms the rate of compaction and flush jobs.It is known that most of write stall comes from compaction in between L0 and L1. [6]

2.3 Related Works

There were many studies to improve database systems by utilizing persistent memory. "Managing Non-Volatile Memory in Database Systems" [7] separated

cold/hot data and used PMEM as a additional MemTable buffer to store them. NoveLSM [3] also used PMEM as an additional MemTable buffer, but used it mainly for storing immutable Memtables, and secondly for storing MemTables under write stall conditions. MatrixKV [6] used PMEM as a matrix container for handling compaction jobs for L0 tree. This matrix container is a compaction optimized architecture, and the idea comes from the analysis that L0,L1 compaction jobs are most critical to write stalls. ChameleonDB [8] and SpanDB [2] used PMEM as a storage for WAL, to reduce overheads of writing logs to Disks before storing key-value pairs.

Our work was largely motivated by work of NoveLSM [3], and it's design is shown in Figure. While RocksDB keeps both MemTable and immutable MemTables in DRAM, NoveLSM copies immutables to PMEM. Additional MemTables can be also created upon write stalls in DRAM. When writing data to MemTable in PMEM, logging doesn't take place because PMEM is a stable storage. We focused on this idea and expanded by moving every MemTables to PMEM, removing the overhead of logging. There are some works to adopt new

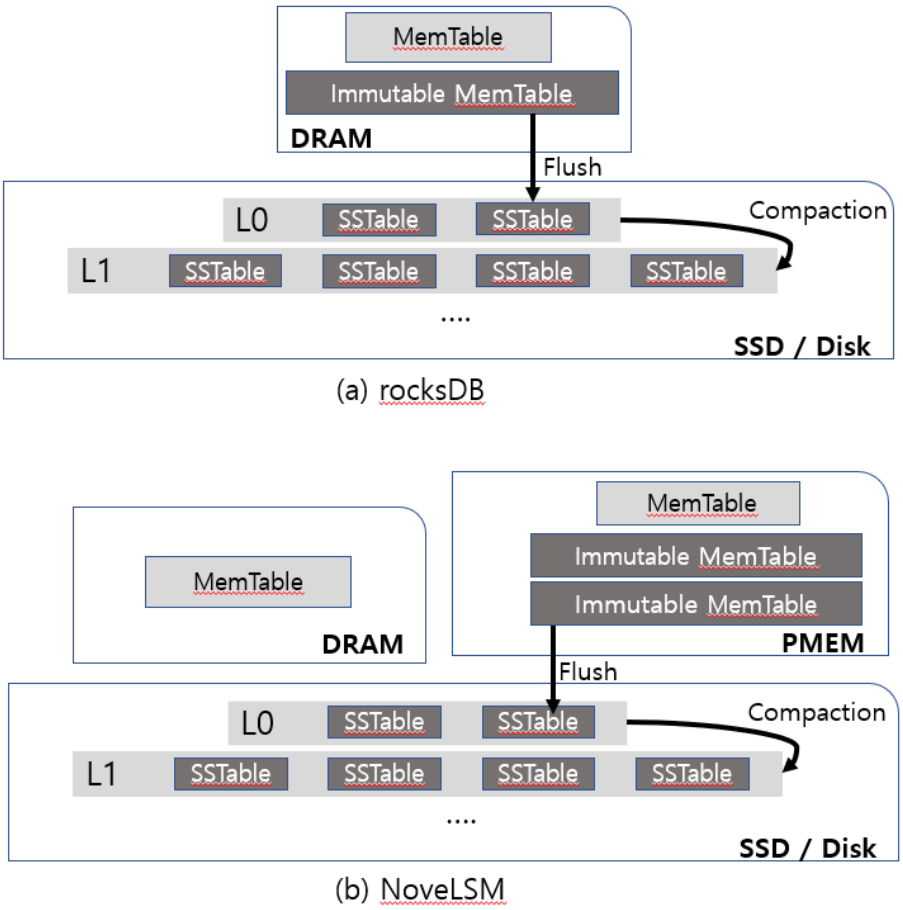


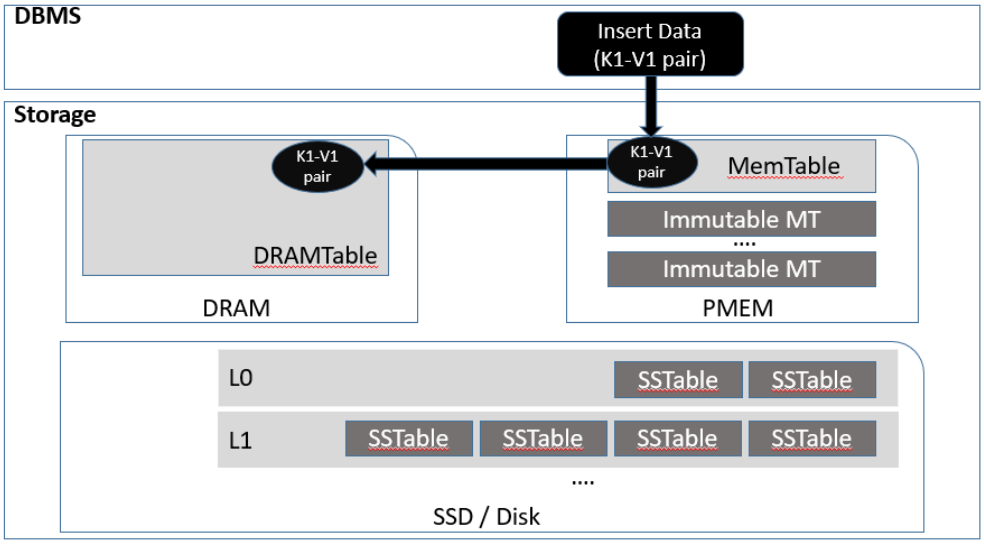
Figure 2.3 Structure of RocksDB and NoveLSM

data structures to enhance performance. HiKV [9] used hybrid index key-value store, and "LSM-trees and B-trees: The best of both worlds" [10] tried to use both LSM-trees and B-trees to utilize their advantages. In our work, we separated read and write path to make easier to adopt these ideas, by creating a room for flexible customization.

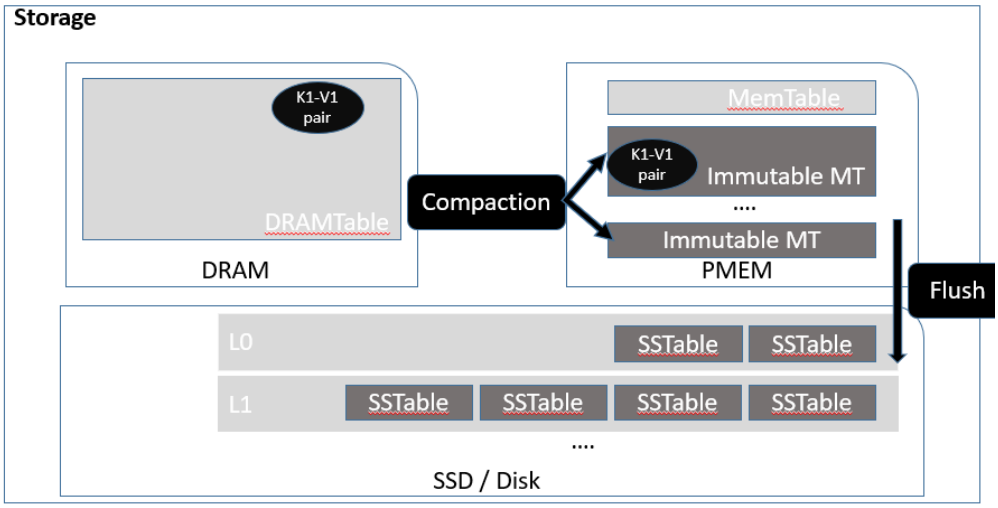
Chapter 3

Implementation and Design

Our design policy is based on the goal to exploit the benefits of PMEM, to overcome the overhead of logging and compaction jobs. Our main idea is to write data on PMEM without logging to enhance write performance and to forward data to DRAM to maintain the read performance. This is implemented by separating read and write path. In this design, DRAM behaves like a buffer carrying latest written data. In this chapter, we show the details of the design and implementation of our work.



(a) Write path



(b) Compaction and flush

Figure 3.1 Insert workflow

3.1 Write Path

Figure 3.1(a) shows the write path of our design. When insert query is handled, key-value pair is directly stored in PMEM, which is implemented by storing them in MemTable. After safely storing key-value pair in stable storage(PMEM), it is then copied and forwarded to DRAMTable. DRAMTable is a MemTable-like instance that resides in DRAM, and works as a container for faster read. If the size of MemTable grows and becomes full, it is turned to an Immutable MemTable. Several Immutable MemTables are wating for background compaction jobs. As shown in Figure 3.1(b), Immutable MemTables are also stored in PMEM. Accessing persistent memory was implemented with the help of PMDK libraries.

In this design, we can point out 3 advantages compared to the original version; (1)This design reduces overhead of logging by writing data to stable storage, which is PMEM. (2) We can avoid write stalls due to compaction by keeping MemTable and Immutable Memtables in larger container, which is provided by PMEM. (3) Since both compaction and flush jobs are done without

involving DRAM, we can utilize DRAM space better by keeping MemTable and Immutable MemTables in PMEM. This will be a cornerstone for optimizing read path by using read-favorable data structures.

3.2 Read Path

Figure 3.2 shows the read path of our design. For reading the data, DBMS looks for the key in DRAM first. If the key is not found in DRAM, the process will go through PMEM and SSD. This process of reading can have a synchronization issues upon updates on existing data. It means that when a data is updated, the same data that resides in DRAM must be updated too, in atomic way. Since read path first looks for DRAM, there is a chance of reading invalid old data. We handle this problem by logically deleting the data in DRAM before an actual update. In this way, read request will not be able to read an old data in DRAM, which data is updated in PMEM. However, if there is a lot of update request, many data in DRAM would be logically deleted. This leads to write amplification. Biggest problem about write amplification in original LSM-tree

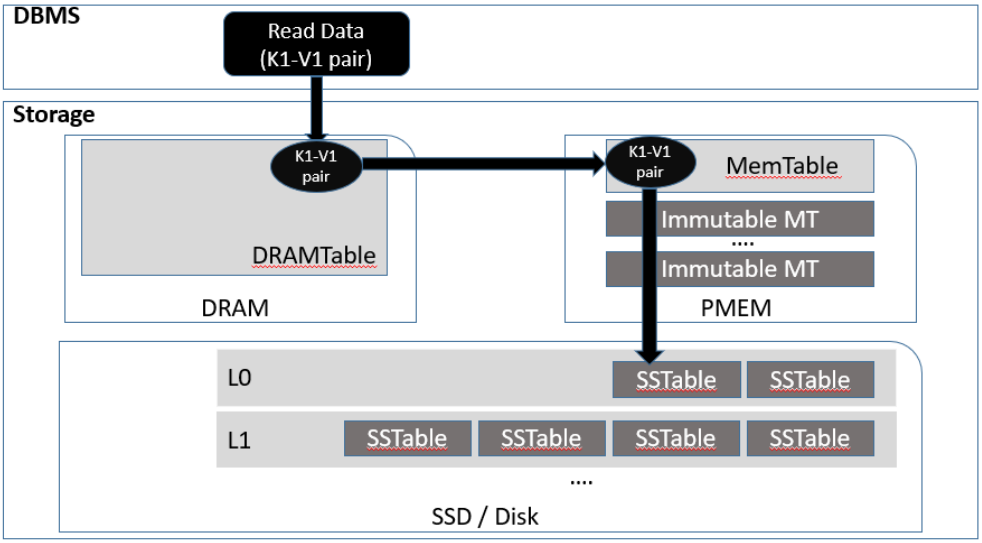


Figure 3.2 Read path

based database system was that it caused too many unnecessary compaction jobs. In our design this is not a problem becasuse DRAM doesn't take part in compaction.

Chapter 4

Evaluation

4.1 Experimental Setup

Hardware Platform : As shown in Figure 4.1.(a), We used a server machine with 2-way 16 core processor, 32GB DRAM, 2 128GB Intel DC Optane Persistent Memory Module, and 1TB hard disk drive.

System Configuration : The baseline of the experiment is RocksDB with version 6.24.0. Our modification was done on the vanilla RocksDB. We used default RocksDB settings for the MemTable, using 3 MemTables with 16MB maximum size.

Component	Specification
CPU	2-way E5-2650(2.2GHz)
DRAM	32GB / Samsung DRAM Module(DDR4)
PMEM	256GB / Intel DC Optane Persistent Memory
HDD	1TB / Western Digital Caviar Blue

(a) Server HW Specification

Type	Characteristics
A	Read 50%, Update 50%
B	Read 95%, Update 5%
C	Read 100%
D	Read latest
F	Read-Modify-Write 100%
G	Insert 100%
H	Insert 50%, Read 50%

(b) YCSB workloads

Figure 4.1 Experimental Setup

Workload and Dataset : We used YCSB benchmark, which is the most popular key-value database workload. The workload's dataset size is set to 40GB, with 1 record having 1KB size. Each workload runs 4 million operations. In addition to 5 YCSB workloads, we customized two workloads more. Like shown in Figure 4.1.(b), workload G and H are our customized version of workload. We added them to test the write intensive workloads, because our work is focused on improving write performance just yet. We also implemented our work in levelDB, to compare the performance against NoveLSM. Again, we used YCSB benchmark with workload size varying from 1 to 10GB(1KB record).

4.2 Experiment Result

Figure 4.2 shows the throughput of baseline and modified version of RocksDB in 7 YCSB workloads. In workloads G and H, we achieved 13.83% and 9.97% of performance gain. This gain comes from reducing logging overhead of insert operations. Workload A is also write-intensive, but showed 2.82% performance gain, which is less than workload g and h. The difference between them is that queries consist of update operations, not insert operations.

Read-intensive workloads showed about 1-3% performance gain, except for workload B. It seems that updates cause some overhead in our policy. We have talked earlier in Chapter 3.2 that update queries results in logical deletes in DRAM. This can be a problem by reducing hit ratio in DRAM. However, figure 4.4 shows that DRAM hit ratio have decreased by small amount in 3 workloads, A, B and C. So it seems that our update policy doesn't create much overhead.

Figure 4.5 shows the throughput of baseline and our modified version(WBL) of LevelDB in 2 YCSB workloads. Here, we used NoveLSM [3] as baseline to

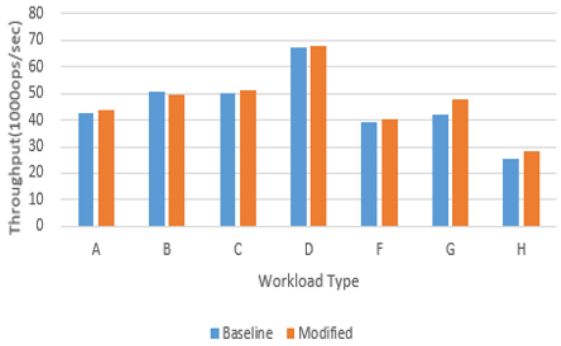


Figure 4.2 YCSB Result in RocksDB

Type	Performance Gain
A	2.82%
B	-2.45%
C	2.28%
D	0.86%
F	3.34%
G	13.83%
H	9.97%

Figure 4.3 Performance Gain

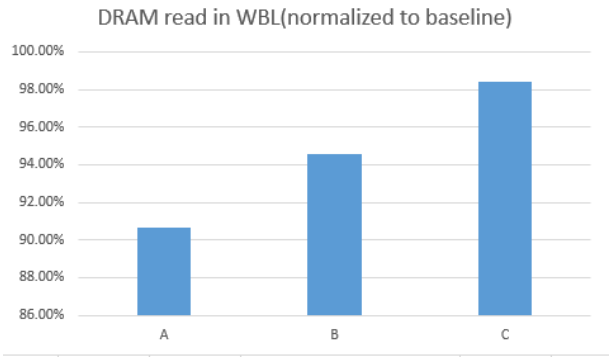


Figure 4.4 DRAM hit compare with baseline

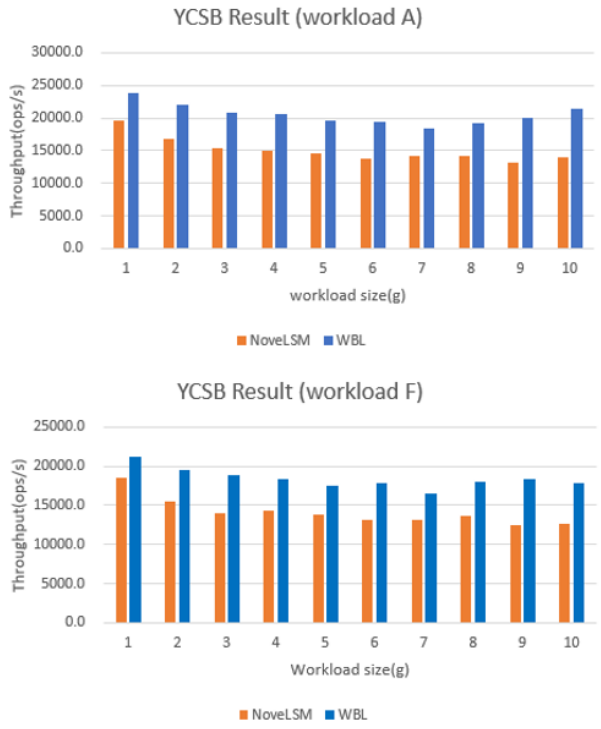


Figure 4.5 YCSB Result in LevelDB

compare with our modified version. WBL showed average of about 20000(ops/s) throughput, which is 30% better than NoveLSM. This seems very impressive, but we have to again, note that our work in LevelDB is very premature. Our work in levelDB was done on base of NoveLSM [3]. We simply forced every key-value store to be stored in NVMTable, and forwarded them to DRAM.

Chapter 5

Conclusion

With the arrival of persistent memory device in data centers, database systems gained a significant room for improvement. It is inevitable to change our system to better fit into new hybrid memory architecture with persistent memory. LSM-tree based key-value database systems, such as RocksDB [4], LevelDB [11], and Cassandra [12] are also optimized for original DRAM-SSD based hierarchy.

Major problems for LSM-tree based database system was (1)overhead of Write Ahead Logging, which sits on top of critical path in writing data, and (2)write stalls induced by waiting for compaction and flush jobs. Computer scientists in other work have tried to to handle both problems by utilizing

PMEM as log or data storage. We propose a idea of separating read and write path, using PMEM as an initial write buffer instead of DRAM. By writing down to stable storage, we removed the overhead of WAL. We use DRAM as a read buffer which is filled by forwarded data from PMEM.

Our experiment results showed that our work performed well on insert queries and write-intensive workloads. This is due to removing overhead of writing logs. On updates queries and read-intensive workloads, however, it showed little performance gain. Updates creates write amplification in our system, but there were little difference in DRAM hit ratio between baseline and our work.

Our plans for future work take place in read path. In our design, space in DRAM can be customized to best provide read performance only. First, we will adopt some garbage policy to handle read performance loss generated by updates. Also, we will test some eviction policy for DRAM. Finally, we plan to change the data structure in DRAM, to provide fast read.

Bibliography

[1] "Intel Optane Persistent Memory." `https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html`.

[2] H. Chen, C. Ruan, C. Li, X. Ma, and Y. Xu, "Spandb: A fast, cost-effective lsm-tree based {KV} store on hybrid storage," in *19th {USENIX} Conference on File and Storage Technologies ({FAST} 21)*, pp. 17–32, 2021.

[3] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Redesigning lsms for nonvolatile memory with novelsm," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pp. 993–1005, 2018.

[4] "A Persistent Key-Value Store for Fast Storage Environments." `https://rocksdb.org/`.

[5] "YCSB: Yahoo! Cloud Serving Benchmark." `https://github.com/brianfrankcooper/YCSB`.

[6] T. Yao, Y. Zhang, J. Wan, Q. Cui, L. Tang, H. Jiang, C. Xie, and X. He, "Matrixkv: Reducing write stalls and write amplification in lsm-tree based {KV} stores with matrix container in {NVM}," in *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pp. 17–31, 2020.

[7] A. van Renen, V. Leis, A. Kemper, T. Neumann, T. Hashida, K. Oe, Y. Doi, L. Harada, and M. Sato, "Managing non-volatile memory in database systems," in *Proceedings of the 2018 International Conference on Management of Data*, pp. 1541–1555, 2018.

[8] W. Zhang, X. Zhao, S. Jiang, and H. Jiang, "Chameleondb: a key-value store for optane persistent memory," in *Proceedings of the Sixteenth European Conference on Computer Systems*, pp. 194–209, 2021.

[9] F. Xia, D. Jiang, J. Xiong, and N. Sun, "Hikv: A hybrid index key-value store for dram-nvm memory systems," in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pp. 349–362, 2017.

[10] V. Jain, J. Lennon, and H. Gupta, "Lsm-trees and b-trees: The best of both worlds," in *Proceedings of the 2019 International Conference on Management of Data*, pp. 1829–1831, 2019.

[11] S. Ghemawat and J. Dean, "LevelDB, A Fast and Lightweight Key/Value Database Library by Google." `https://github.com/google/leveldb`.

[12] "Facebook. Cassandra on RocksDB at Instagram." `https://developers.facebook.com/videos/f8-2018/cassandra-on-rocksdb-at-instagram`.

[13] "Persistent Memory Development Kit." `https://github.com/pmem/pmdk`.

요약

현재 LSM 트리 기반의 키-밸류 데이터베이스 시스템은 DRAM-SSD 메모리 구조를 기반으로 최적화되어 있다. 하지만 영구 메모리가 등장함으로서, 메모리 계층구조에 변화가 생겼다. 영구 메모리는 DRAM에 근접한 수준의 성능을 가진 비휘발성 메모리이다. 우리는 제 이 하이브리드 메모리 계층구조에서 영구 메모리를 잘 활용해서 데이타베이스 시스템을 새롭게 최적화할 필요가 있다. 보통 LSM 트리 기반의 키-밸류 데이터베이스 시스템은 Write-Ahead-Logging(WAL)과 (2)Compaction으로 인한 Write Stall에 의해 성능이 저하되는 문제점을 가지고 있다. 이 연구에서는 영구메모리가 도입된 메모리 계층구조에서 읽기/쓰기 경로를 분리함으로서 이 문제를 해결하고자 한다. 먼저 쓰기 경로는 DRAM을 거치지 않고 바로 PMEM에 씀으로서 WAL의 필요성을 제거한다. 영구 메모리에 적은 데이터는 후에 DRAM으로 이동한다. DRAM은 compaction이나 쓰기 경로 위에 존재하지 않기 때문에 오직 읽기만을 위해 최적화 되기 편해진다. 이 연구는 rocksDB 버전 위에서 코드 수정을 통해 진행되었다. 그리고 기존의 rocksDB와의 성능 평가를 위해 YCSB 워크로드를 사용했다. 실험 결과를 통해 우리의 연구가 write가 많은 워크로드에

서는 주목할만한 성능 개선이 있었고, read가 많은 워크로드에서는 약간의 성능 개선이 있다는 것을 알게 되었다.

주요어: 영구 메모리, LSM 트리, 키-밸류 데이터베이스, Write Stall, WAL, 읽기/쓰기 경로 분리

학번: 2019-20797