DESIGNING LARGE-SCALE KEY-VALUE SYSTEMS ON HIGH-SPEED STORAGE DEVICES

by
XINGSHENG ZHAO

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON
May 2022

DESIGNING LARGE-SCALE KEY-VALUE SYSTEMS ON HIGH-SPEED STORAGE DEVICES

The members of the Committee approve the doctoral dissertation of Xingsheng Zhao

Song Jiang ______________________________
Supervising Professor

Hong Jiang ______________________________

Hao Che ______________________________

Jia Rao ______________________________

Dean of the Graduate School ______________________________

Copyright © by Xingsheng Zhao 2022
All Rights Reserved

To my family,
for their constant support and unconditional love.

ACKNOWLEDGEMENTS

The end of one journey is the beginning of the next. Before the new journey begins, I would like to thank a great many people who have greatly supported me during my PhD study at UT Arlington and the writing of this thesis.

First and foremost, I would like to express my deepest gratitude to my erudite supervisor, Dr. Song Jiang for his continuous support, invaluable guidance and generous help during my PhD study. I am so glad to be part of our group. And I still remember the day when I was on the same flight to the United State with Dr. Jiang as the beginning of my PhD journey. What a coincidence! I will also not forget all those illuminating discussions with him at each stage of my research and my writing, which could turn out to be a whole day long, led me to this final completion of this thesis. I learned a lot from his inspiring ideas and enthusiasm in scientific researches. Those experiences trained me to be a person with independent thought and an open mind, which will be very helpful for my future study and work.

I would also like to extend my deepest appreciation to my committee members, Dr. Hong Jiang, Dr. Hao Che, and Dr. Jia Rao, for their constructive comments, suggestions, and supports throughout this work.

I am grateful to my industry colleagues. I would like to thank Qingda Lu, Zhu Pang, Shuo Chen, Qingqing Zhou and Kuang He, for their supports and suggestions. I am delighted to have worked with them.

I also had great pleasure of working with my labmates, who shared their best knowledge with me. I thank Yuehai Xu, Xingbo Wu, Fan Ni, Leijie Zeng, Haitao Wang, Zhuo Huang, Chen Zhong, Venkata Naga Prajwal Challa, Kun suo, Yong Zhao, Xiaofeng Wu, Lingfeng Xiang, Hang Huang, for your enthusiastic support, timely help and encouragement. I also want to thank my friends at UTA, Chaochao Yang, Huiyang Li, Jian Li, Jiayi Wang, Lin Sun, for your accompany.

This five years PhD journey would not have been possible without the support of my family. I would not have made it this far without them. Mom, thank you for everything you have done for me. Thanks for being always present and doing the (im)possible to keep me going further. Yingge, my dear wife and best friend. I am so luck to meet you at college. Without you, I will never have the courage to pursue the PhD journey in a foreign country. Your kindness, patience, tolerance and understanding helped me be a better person.

May 1, 2022

ABSTRACT

DESIGNING LARGE-SCALE KEY-VALUE SYSTEMS ON HIGH-SPEED STORAGE DEVICES

Xingsheng Zhao, Ph.D.
The University of Texas at Arlington, 2022

Supervising Professor: Song Jiang

With the evolution of new technologies, such as edge computing, full self-driving, virtual reality, and multi-media streaming, the volume of data is growing at an accelerated speed. The global data volume could achieve 175 zettabytes by 2025. With this huge amount of data, the focus of data management has been shifted from traditional SQL databases to NoSQL databases, which provide higher performance and better scalability. Key-value (KV) stores are a common type of NoSQL database and are becoming a major storage infrastructure in various application domains. With the development of high-speed storage devices, such as NVMe SSD, Open-channel SSD, and non-volatile memory, new challenges and opportunities have appeared for KV stores. Traditional KV stores suffer from large write amplification and non-trivial indexing overhead on the high-speed storage devices. And the performance bottleneck gradually shifts from the storage side to the software side in modern database designs.

In this dissertation, we propose solutions to overcome the obstacles in the KV store design. To support efficient indexing and range search, key-value items must be sorted. However, the sorting process can be excessively expensive. In the KV systems adopting the Log-Structured Merge Tree (LSM) structure, the write amplification can be very large due to its repeated internal merge-sorting operation. We propose WipDB, which leverages relatively stable key distributions to bound the write amplification at a small number. Meanwhile, to improve the efficiency for large-scale KV stores, we need persistent indexes on the Non-volatile memory (PMEM) to provide instant-recovery ability for the database. However, existing designs of the persistent index, especially the persistent hash table, always make trade-offs between performance, consistency, and persistency. To meet all three requirements, we propose TurboHash for building a high-performance KV store.

Though PMEM provides persistency of data, compared to the DRAM, the performance of PMEM is still much worse than DRAM in terms of either latency and throughput

(by around 3X or more). While using DRAM as a read cache and/or write-back buffer seems to be a plausible remedy to close the performance gap, traditional cache designs are not effective or even not functional for data of either weak temporal or spatial access locality and requiring persistency and crash consistency. To address these issue, we propose a framework, that turns an index structure designed for persistent memory into a much faster one with application-managed caching and buffering.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS v
ABSTRACT vi
Chapter Page
1. INTRODUCTION 1
1.1 Reducing the Write Amplification for Key-Value Stores 1
1.2 Reducing Overhead for Crash-consistent Indexes on NVMs 2
1.3 Using Hybrid DRAM-PMEM architecture to improve indexes performance for persistent memory 3
1.4 Organization 4
2. WIPDB: A WRITE-IN-PLACE KEY-VALUE STORE THAT MIMICS BUCKET SORT 5
2.1 Introduction 5
2.2 Background 8
2.2.1 Why LSM-tree? 8
2.2.2 Why Approximate Sorting? 9
2.2.3 Overcoming Variation of Key Density 11
2.2.4 The Write-in-place Approach 12
2.3 The WipDB Design 13
2.3.1 The WipDB Architecture 13
2.3.2 The Operations 15
2.3.3 Efficiency and Persistence of MemTables 15
2.3.4 Support of Range Search 16
2.3.5 Bucket Splitting and Merging 17
2.3.6 Use of Write Ahead Log for DRAM resident MemTable 18
2.3.7 Read-aware Compaction Scheduling 19
2.4 Evaluation 21
2.4.1 Experiment Setup 21
2.4.2 Write Performance 22
2.4.3 Read Performance 24
2.4.4 Impact of WAL on Restart Time 26
2.4.5 Results of the YCSB Benchmarks 26
2.5 Related Work 28
2.5.1 Optimizations for efficient compaction. 29

2.5.2 The tiering merge scheme. 29
2.5.3 Key-value separation. 30
2.5.4 In-memory key-value stores 30
2.6 Summary 31
3. TURBOHASH: A HASH TABLE FOR KEY-VALUE STORE ON PERSISTENT MEMORY 32
3.1 Introduction 32
3.2 Motivations 36
3.2.1 Probing Scope and Distance. 36
3.2.2 Sequential and Random Accesses. 37
3.3 The Design of TurboHash 38
3.3.1 The Architecture 38
3.3.2 Establishing the Search Path 39
3.3.3 A Bucket's Data Structure 41
3.3.4 Insert, Update, Delete, and Read 42
3.3.5 Shard Resizing & Failure Recovery 45
3.4 Evaluation 45
3.4.1 Experiment Setup 46
3.4.2 Overall Performance 47
3.4.3 Latency Comparison 49
3.4.4 Results of YCSB Benchmarks 51
3.4.5 Load Factor 52
3.4.6 Probing Distance 53
3.4.7 Shard Rehashing 54
3.5 Related Work 55
3.5.1 Hash Table for Persistent Memory 55
3.5.2 Hash Table Concurrency Control 56
3.6 Summary 56
3.7 Appendix 57
4. Spot-On: Optimizing Use of DRAM to Improve Performance of Index Structures on Intel Optane DC Persistent Memory 61
4.1 Introduction 61
4.1.1 DRAM as a Cache of PMEM 62
4.1.2 DRAM as a Write Buffer of PMEM 62
4.2 Background 63
4.2.1 Non-Volatile Memory 63
4.2.2 Persistent Indexes 64

4.3 Case Study 65
4.3.1 Case Study: Buffering 65
4.3.2 Case Study: Out-place-update 66
4.3.3 Case Study: Caching 67
4.3.4 Combine Together 67
4.4 Design 69
4.4.1 Three layers in SPTree 70
4.4.2 Concurrent Control 71
4.4.3 Search Operation 71
4.4.4 Insert/Update/Delete/Lookup/scan 72
4.4.5 Split and Merge 73
4.4.6 Recovery and Crash Consistency 73
4.5 Evaluation 74
4.5.1 Experiment Setup 75
4.5.2 Overall Performance 75
4.5.3 Latency Comparison 77
4.5.4 Recovery 77
4.6 Summary 78
5. CONCLUSIONS AND FUTURE WORK 79
5.1 Contributions 79
5.2 Future Work: Data rearrangement supporting log-structured storage 79
REFERENCES 81
BIOGRAPHICAL STATEMENT 87

CHAPTER 1
INTRODUCTION

We are in the era of big data. Large volumes of data are being generated, stored, and analyzed every second. This happens not only in data centers hosting thousands of servers, but also in everyone's computers and smartphones. While the data volume is quickly growing, the tasks on organizing, retrieving, and processing the data become ever challenging. In the recent years the focus of data management has been largely shifted from traditional SQL databases to NoSQL (Not-only-SQL) databases, such as key-value (KV) stores and key-value caches, which provides essential functionalities and much higher performance for storing and retrieving data. Unfortunately, the design of these systems are often ineffective, sometimes incapable of tackling challenging tasks involving massive amount of data. Even running on servers with powerful CPUs, large DRAM, fast storage devices of large capacity, and low-latency network, the applications still have a hard time achieving their expected performance.

This dissertation investigates the causes of the issues in KV systems and identifies opportunities to enable efficient data accessing for big-data applications.

1.1 Reducing the Write Amplification for Key-Value Stores

Key-value (KV) stores have become a major data management component in a storage system. By providing a KV API for writing, reading, and updating data items, the store makes it possible for the upper-level software to access the storage in its own defined key space for values of any sizes. In contrast to the rigid block interface provided by the block storage subsystems, such as physical/virtual disks and storage volumes, a KV interface makes user-facing software, much easier to develop, as it can leave chores, such as address conversion between keys and block addresses and storage space management, to the KV system. More importantly, in a distributed system the key space can be conveniently partitioned into multiple shards, enabling a shared-nothing architecture to achieve linear horizontal scalability. However, to realize its full potential a KV store must be well designed to simultaneously meet a number of goals, which are (1) good performance with small KV items in a storage system of large capacity, (2) support of range search, (3) low read amplification, and (4) low write amplification. It is a challenge to simultaneously

achieve all these goals in a KV system. To achieve the first three goals, the KV items have to be sorted by their keys. If sorted, the store can index disk blocks, instead of individual KV items, to significantly reduce index size and use only one disk access to service a read request. When the items are sorted, range search (e.g., for all items between two keys, or for items of a common prefix) can be well supported. Unfortunately, maintaining an always and fully sorted list on the disk is a prohibitive task. It usually leads to exorbitant write amplification.

To address these issues, we propose a new KV store (WipDB). Accordingly, the write amplification can be effectively reduced and most sorting operations can be moved off the read service's critical path. WipDB dynamically partitions the entire space into a large number of buckets with the knowledge of long-term key distribution, and placing keys directly in the corresponding buckets. Meanwhile, expecting existence of read locality in the key space, WipDB introduces a locality-aware sorting scheme. Sorting indifferent buckets can be scheduled so that sorting of KV items in buckets that do not actively service read requests can be postponed. Experiment results show that WipDB improves write throughput by 3 to 8×(to around 1 Mops/s on one Intel PCIe SSD) over state-of-the-art KV stores.

1.2 Reducing Overhead for Crash-consistent Indexes on NVMs

A hash table is a fundamental data structure for efficient organization of key-value (KV) data in the memory. It allows data to be quickly located with few intermediate index search. This is especially important for organizing a very large number of small KV items, where often it may take only one cache-line memory access to retrieve a data item. Had many non-sequential memory accesses been required on an index structure, such as B+ tree or skip list, the actual cost of reading a small piece of data would be amplified by multiple times [72, 53]. Accordingly, hash tables have been employed to manage key-value cache in the DRAM, such as MemCached [55, 28] and Redis. With emergence of byte-addressable non-volatile memory, efforts have been made to design persistent hash tables on the memory [84, 52, 15, 44, 83, 48, 86, 43, 71, 85, 32], as well as using persistent memory to build KV stores [76, 24, 38, 37, 78, 81]. One of the major issues and challenges on designing a high-performance persistent hash table is on its efficient support of crash consistency and atomic update. With crash consistency, a data structure can stay in a consistent state, or be restored to a consistent state after an unexpected crash. To assure the consistency, one has to enforce a particular order on a sequence of actions. A simple example is that a hash table's bucket has to be allocated and initialized before its address can be assigned to a pointer in the table's directory. To enforce the order, fence/flush primitives, which are

expensive, have to be used between the operations. More extensive use of fence/flush is required for updating a directory during a rehashing operation. For data integrity, update of a piece of data, such as a key or a value, must be atomic. After a recovery from an unexpected crash, either a version of the data before the update or the one after the update must be recovered. This means that the data cannot be modified in place if it is larger than an atomic write unit (8 bytes). Otherwise, it may destroy the old version without making the new version established at the time of a crash.

We propose a persistent hash table design, named TurboHash, for a high-performance key-value store on the non-volatile byte-addressable memory (Intel Optane DC), by addressing all of the aforementioned issues. TurboHash first hashes keys into multiple shards. Each shard is a small hash table. TurboHash is designed to support a KV store whose capacity, in terms of number of KV items, can be well planned according to known memory size and expected KV sizes. Accordingly, we can pre-determine number of shards according to expected shard capacity. By using a well-randomized hashing function, such as MurMurHash, and allowing a sufficiently high shard capacity, we can avoid an expensive re-sharding operation on the entire table. Within each shard, buckets of 256 bytes are physically contiguous. Linear probing is used to resolve hash collisions within a shard. When a collision cannot be resolved in the existing shard, the shard itself is resized. TurboHash uses linear probing in a shard and supports out-of-place updates. Specifically, it proposes near-place update (in the same bucket where the old version stays), and uses one atomic write to efficiently make old/new versions of a key invisible/visible, respectively. It supports lock-free reads. It develops well-defined search paths so that a search does not have to always cover the entire probing scope for a negative search. Experiment results demonstrate that TurboHash improves state-of-art designs by 2×-8× in terms of throughput and latency.

1.3 Using Hybrid DRAM-PMEM architecture to improve indexes performance for persistent memory

The memory/storage hierarchy, which consists of multiple levels including CPU cache, DRAM, and block devices such as SSDs and HDDs, has been stable for decades. Accordingly, the principal management designs for data across its levels, such as set-associative CPU caches, page-based virtual memory and block-based read cache and write-back buffer, are well established by carefully considering individual devices' performance characteristics to maximize the hierarchy's performance. However, with emergence of Intel Optane DC persistent memory (Optane PMEM for short), the first commercially available persistent byte-addressable memory, a new level is introduced into the hierarchy. We con-

tend that it is necessary for the DRAM to serve as a cache level for the PMEM to boost its effective performance.

Like DRAM, the Optane PMEM is a byte-addressable memory device that can be directly accessed via load and store instructions. However, its performance gap with the DRAM is still significant (around 2.5X-3X worse than that of DRAM in terms of its latency and throughput). Therefore, placing the PMEM underneath the DRAM in the hierarchy has the potential of taking advantage of DRAM's high performance. While DRAM is used as PMEM's cache, it not only should be used as a read cache, but also must be used as a write buffer to enable the write-back policy. Optane PMEM has an access unit of 256 bytes to the memory's media, any write smaller than the size leads to a write amplification and reduction of effective throughput. To address these issues, we designed Spot-on, a framework which turns an index structure designed for persistent memory into a much faster one with application-managed caching and buffering. We develop SPTree based on Spot-on. SPTree buffers the internal nodes in DRAM and existing keys in bloom filters, while asynchronously updating internal nodes in PMEM for crash consistency. Meanwhile, it assigns write buffer for write-intensive workloads to reduce write amplification. Experiments show that SPTree provides higher write and read throughput up to 2X - 4X respectively compared with the state-of-the-art persistent B+-Tree designs.

1.4 Organization

The rest of the dissertation is organized as follows. In Chapter 2, we introduce the root causes of write-amplification (WA) for LSM-tree based Key-Value stores, and how can we reduce the WA with an upper bound by extending the database horizontally with the knowledge of key distribution. In Chapter 3, we present TurboHash, a persistent hash table providing both high-performance and crash consistency for large scale Key-Value Stores. In Chapter 4, we propose Spot-on, a framework that improves the PMEM index performance by caching and buffering. We also develop SPTree, a high performance persistent B+-Tree. In Chapter 5, we summarize our contributions and discuss the future work.

CHAPTER 2

WIPDB: A WRITE-IN-PLACE KEY-VALUE STORE THAT MIMICS BUCKET SORT

Key-value (KV) stores have become a major storage infrastructure on which databases, file systems, and other data management systems are built. To support efficient indexing and range search, the key-value items must be sorted. However, this sorting process can be excessively expensive. In the KV systems adopting the popular Log-Structured Merge Tree (LSM) structure or its variants, the write volume can be amplified by tens of times due to its repeated internal merge-sorting operation.

In this paper we propose a KV store design that leverages relatively stable key distributions to bound the write amplification by a number as low as 4.15 in practice. The key idea is, instead of incrementally sorting KV items in the LSM's hierarchical structure, it writes KV items right in place in an approximately sorted list, much like a bucket sort algorithm does. The design also makes it possible to keep most internal data reorganization operations off the critical path of read service. The so-called Write-in-place (Wip) scheme has been implemented with its source code publicly available. Experiment results show that WipDB improves write throughput by 3 to 8× (to around 1 Mops/s on one Intel PCIe SSD) over state-of-the-art KV stores.

2.1 Introduction

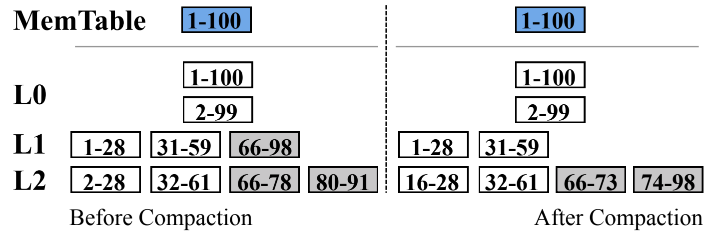
Key-value (KV) stores have become a major data management component in a storage system. By providing a KV API for writing, reading, and updating data items, the store makes it possible for the upper-level software to access the storage in its own defined key space for values of any sizes. In contrast to the rigid block interface provided by the block storage subsystems, such as physical/virtual disks and storage volumes, a KV interface makes user-facing software, such as file systems [36, 64] and database systems [25, 12], much easier to develop, as it can leave chores, such as address conversion between keys and block addresses and storage space management, to the KV system. More importantly, in a distributed system the key space can be conveniently partitioned into multiple shards, enabling a shared-nothing architecture to achieve linear horizontal scalability.

However, to realize its full potential a KV store must be well designed to simultaneously meet a number of goals, which are (1) good performance with small KV items in a storage system of large capacity, (2) support of range search, (3) low read amplification, and (4) low write amplification.

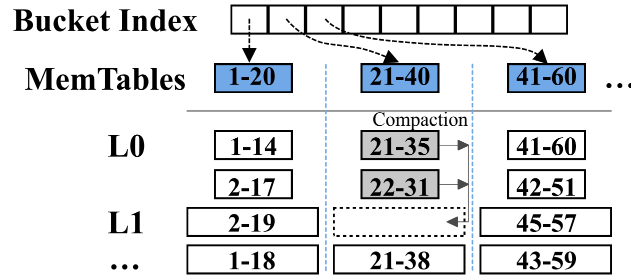
It is a challenge to simultaneously achieve all these goals in a KV system. It has been noted that in many real-world KV workloads small keys and values are common [4, 54]. A KV item can be tens of bytes. Each (4KB) disk block may contain tens of such items. In a server of multi-terabyte storage space there can be tens of billions of such items. If such a huge number of data items are individually indexed, the index size (tens of, and even hundreds of gigabytes) can be too large to be fully held in the memory. If (some) index data are only stored on the disk, it needs multiple disk reads (for index data and then KV data) to service a read request, leading to high read amplification. To achieve the first three goals, the KV items have to be sorted by their keys. If sorted, the store can index disk blocks, instead of individual KV items, to significantly reduce index size and use only one disk access to service a read request. When the items are sorted, range search (e.g., for all items between two keys, or for items of a common prefix) can be well supported.

Unfortunately, maintaining an always and fully sorted list on the disk is a prohibitive task. Analogous to expensive data shifting operations for maintaining a sorted array in the memory, writing new KV items into an on-disk sorted list, which is stored in one or multiple files, involves re-writing of file(s). This usually leads to exorbitant write amplification. High write amplification, on the one hand, compromises foreground throughput as more bandwidth is consumed by background disk I/O. On the other hand, it reduces lifetime of SSD disks. Unsurprisingly, major efforts on improvement of KV stores have been on the amelioration of the write-amplification issue.

Currently the most successful effort is to employ the log structured merge tree (LSM-tree) technique to maintain multiple and increasingly large levels of sorted lists [57] (See Figure 2.1a). Example KV systems adopting the technique include Google's LevelDB [30] and Facebook's RocksDB [26]. In these systems new KV items in the incoming write requests are progressively merge-sorted on different levels across the hierarchical structure and finally enter the last and the largest level of a sorted list. Though LSM-tree can significantly reduce write amplification, the amplification can still be as high as 20∼70X depending on store size [73, 62, 50, 10]. To reduce the high write cost, researchers adopt the approximate sorting technique, attempting to avoid rewriting data in the same level and reduce the amplification to as low as number of levels (e.g., 5∼7X) [62, 73]. In such a sorting method, the *key space* is partitioned into multiple buckets (each for a given key range), and KV items are globally sorted (across the buckets) but locally unsorted (within a bucket). While the level size grows exponentially in the LSM-tree hierarchy, the store usu-



(a) Compaction in LSM-tree



(b) WipDB

Figure 2.1: *Architectures of LSM tree and WipDB*

ally has seven or fewer levels. If successful, the technique can substantially reduce write amplification.

However, current approximate sorting technique has two critical issues in its effort on achieving low write amplification. The first issue is that the approximate sorting technique cannot be effectively applied on the LSM-tree structure due to its demand on an appropriate key partitioning. As a consequence, the stores adopting this technique either do not support range search (e.g., LSM-trie [73]) or consume a considerable amount of memory in guards to reduce the write amplification. (e.g., PebblesDB [62]). The second issue is that the expensive across-level progressive sorting operations compromise performance of read requests. In terms of performance impact, the high write amplification is less of an issue on the write performance than on the read performance. If the sorting operations are not immediately conducted (e.g., postponed to less busy periods), writes can be quickly done. However, the read performance will be seriously degraded if the items are not adequately sorted (more explanation in Section 2.2). However, if the sorting is carried out immediately, expensive across-level sorting would consume substantial bandwidth, also degrading the read performance.

In this paper we propose a new KV store architecture to fundamentally address the two issues. Accordingly, the write amplification can be effectively reduced and most sorting operations can be moved off the read service's critical path. In the way, all the four

goals can be simultaneously achieved. The proposed KV store is named WipDB (Write-in-place). It mimics the bucket sort by partitioning the entire key space into a large number of buckets, and placing keys directly in the corresponding buckets. In contrast, existing LSM-tree-based KV stores, including LevelDB and the optimized ones using the approximate sorting technique, place KV items into a level's corresponding buckets (e.g., represented by SSTables in LevelDB) and move them down to next level's buckets using merge sort in a level-by-level manner, as shown in Figure 2.1a. However, similar to that in a bucket sort a key has only one bucket in WipDB holding it and KV items are not moved across buckets. KV items in a bucket are managed with a miniature LSM tree (see Figure 2.1b).

In this WipDB architecture, both of the aforementioned issues can be well addressed. First, the approximate sorting can be used in each miniature LSM tree of a bucket without key partitioning to reap its full performance benefit. Second, expecting existence of read locality in the key space, WipDB introduces a locality-aware sorting scheme. Sorting in different buckets can be scheduled so that sorting of KV items in buckets that do not actively service read requests can be postponed. This benefit for hiding internal data sorting cost is not possible in existing LSM-tree-based KV stores.

In summary, in the paper we make three major contributions: (1) we propose a new LSM-tree-based KV store architecture that allows new keys to be written in their right places (buckets) in a sorted list at the outset by utilizing long-term key distribution; (2) by improving and hiding write-related sorting cost, WipDB can make both writes and reads efficient; and (3) we have implemented WipDB and extensively evaluate its efficiency by comparing it with state-of-the-art KV stores.

2.2 Background

To motivate the design of the proposed WipDB, in this section we will discuss the design rationale of the LSM-tree architecture and its weaknesses, as well as existing efforts attempting to address the weaknesses and their inadequacies.

2.2.1 Why LSM-tree?

As we have indicated, KV items must be sorted in the storage for reduced index size (so that it can be all cached in the memory) and support of range search. This is especially the case for a KV store hosting a large number of small KV items. The challenge is to keep the items sorted when new items keep streaming into the store, as the cost can be huge. Assuming that the store currently has M KV items (of the same size) that have been fully sorted on the disk. A list of N (sorted) new items currently in the memory are to be written

to the store. To maintain a fully sorted store, one has to write $M + N$ items, rather than just the N new items to the store. Specifically, one needs to first read the M items off the disk, merge-sort the new list of N items with the existing list of M items into one list, and write the resulting sorted list back to the disk. The operation is often named *compaction* [12]. The write amplification is $(M + N)/N$, which can be huge when the store size (M) and the ratio of the two lists (M/N), named *compaction ratio*, become increasingly large. To address the issue there are two potential ways. One is to reduce the ratio (M/N). LSM-tree takes this way. The other is to increase size of the in-memory list (N). This is the way taken by the proposed WipDB. However, there are significant hurdles to overcome on the way towards its effectiveness (see Section 2.2.3).

To limit the ratio and allow the store to grow to a very large size, LSM-tree introduces multiple sorted lists to form a multi-level hierarchy. As illustrated in Figure 2.1a, these levels are named Levels 0, 1, ..., and $n - 1$ in a n-level hierarchy. Except for L_0, which sits at the top of the hierarchy and may have multiple sub-levels of similar size, every other level is up to 10X as large as the level immediately above it.

Because compaction is conducted only between two adjacent levels, the compaction ratio is bounded by 10, or the write amplification for moving KV items from one level to its next lower level is bounded by 11. For example, in a 5-level LSM-tree store, starting at Level 0 KV items move down the hierarchy level-by-level in a sequence of compactions and produce a write amplification of up to 44 when they arrive at Level 4. Though LSM-tree's write amplification is way much smaller than the store that inserts items directly into a single sorted list, its amplification is still significant. High write amplification consumes much of the I/O bandwidth and makes both write and read requests slow.

2.2.2 Why Approximate Sorting?

Apparently, LSM-tree's 20-70X write amplification is still a major performance concern. Recently, significant efforts have been made to reduce the cost by using *approximate sorting* to avoid rewriting data in the same level [73, 62]. As we know, to reduce write amplification LSM-tree does not directly sort KV items into a single list. Instead, it only sorts the items in the same level (horizontally sorted) and leaves items in different levels unsorted (vertically unsorted). In the approximate sorting, the key space in each level is partitioned into multiple segments, named buckets. Keys between the buckets are sorted. Keys within a bucket are not fully sorted. There can be multiple overlapping small sorted lists in a bucket. Each level is said to be approximately sorted.

By allowing each level to be horizontally unsorted, the key-value architectures, represented by LSM-trie [73] and PebblesDB [62], can potentially reduce a compaction's write

amplification to 1, and possibly make the store's write amplification as low as its level count. Here is the reason. During a compaction, to move items in a bucket at a level to its next level, the store first merge-sorts the small lists in a bucket into one sorted list. It then uses the boundaries of buckets at the next level to segment the list and simply writes each segment to its corresponding bucket in the next level. Allowing KV items to be partially (either horizontally or vertically) unsorted, a read operation needs to search more sorted lists, such as levels in a LSM-tree and small lists in a bucket of an approximately sorted list. By using stronger bloom filters this may not be a performance concern. The real concern is on use of the approximate sorting in different levels.

For the approximate sorting approach to be effective, one has to segment the key space appropriately so that each bucket has about equal number of KV items [1]. Otherwise, some buckets may receive new items at a (much) higher rate than others, making them and their downstream buckets become very large, and their compactions expensive. The hierarchy would also grow in an unbalanced manner. This is why LevelDB introduces constant-size SSTable. It is also reminiscent of the tree balancing issue addressed in the B+ tree.

Currently, the issue is not well addressed in the use of approximate sorting, leaving its performance promises unfilled. To ensure a balanced LSM-tree structure, LSM-trie hashes user-supplied keys with a cryptographic hash function like SHA-1, and uses the hashed keys for sorting in the tree. A consequence of this approach is that range search is not supported. In contrast, PebblesDB attempts to preserve the support of range search by keeping user keys sorted. To this end, it uses a probabilistic function to select some keys as "guards" from all keys entering a level to form buckets (between two adjacent guards). Number of guards, or buckets, in a region of the key space in a level is roughly proportional to the region's key density. Intuitively, each bucket has about the same number of KV items. But positions of guards may keep changing, especially for guards in the higher levels (Level i where i is small), in response to variation of key density in the corresponding key regions. In PebblesDB, a guard in a higher level must also be a guard in all of its lower levels [62]. Adding a new guard into a level requires splitting of a bucket. This will either cause rewriting data in the same level, which defeats the very purpose of using approximate sorting, or requires an immediate premature compaction of the bucket to the next level, leading to a cascade of downward compactions. Both can significantly offset the benefit of the approximate sorting.

[1] This is a requirement similar to that on an efficient bucket sort.

2.2.3 Overcoming Variation of Key Density

The key to success of the approximate sorting approach is stability of the key distribution in the key space. In other words, the key densities in different regions of the key space need to remain roughly unchanged (at least for an extended period of time), so as to keep the partitioning (into the buckets) stable. We assume user keys generated by applications have a long-term relatively stable distribution (e.g., for weeks, months, or even years). For real-world services, keys are often constructed by sequencing some descriptors of an object. For example, a key about an Amazon's product can be generated as *Grocery* $\rightarrow$ *Snack Foods* $\rightarrow$ *Cookies* $\rightarrow$ *Chocolate* $\rightarrow$ *Oreo Mini Chocolate Sandwich Cookies*. In the example of Google's BigTable [12], which decomposes a database table into KV items for storage in a distributed KV store, a key is generated by concatenating row name, column name, timestamp, such as *com.google.maps/index.html+Spanish+04/18/2019,12:00am*. In the real world the number of products or news articles under a certain category, or popularity of a category, can be expected to be stable within a relatively long time period. So is the long-term key-distribution stability in many significant application scenarios [2].

The issue is that existing use of the approximate sorting approach cannot exploit the long-term stability, as such stability may only exist in the last one or two levels, which store majority of a store's KV items inserted over weeks, months, or even years reflecting the user keys' inherent and stable distribution. In contrast, KV items in the top levels are inserted in a short time period and may exhibit a key distribution that's different from the long-term stable one and can change quickly from time to time. For example, Level 0 has a capacity of tens of megabytes, and may only store KV items inserted in the last few minutes. However, when stores such as PebblesDB [62] applies the approximate sorting approach at every level, dramatic and expensive bucket adjustment is expected in the most (top) levels, which cascades to the lower levels.

To illustrate the situation, we continuously write 100-byte KV items, whose keys are generated using the `db_bench` tool in the LevelDB code release. In each level we contiguously place a hypothetical guard for every 50K keys, or a bucket between two adjacent guards holding 50K KV items. After 1 billion items have been inserted, we track variation of guard positions in Levels 1, 2, and 3 after every compaction. As Figure 2.2 shows, in all the three levels the guard positions vary but at different intensity. In other words, if we had fixed the guard positions the number of KV items in a bucket can be highly variable in Levels 1 and 2. However, in a lower level (Levels 3), where much more items are stored, spontaneous variation of key distribution can be smoothed out.

[2]In Section 2.4.2, we experimentally show that WipDB's performance advantage is not contingent on existence of a strong stability at all.

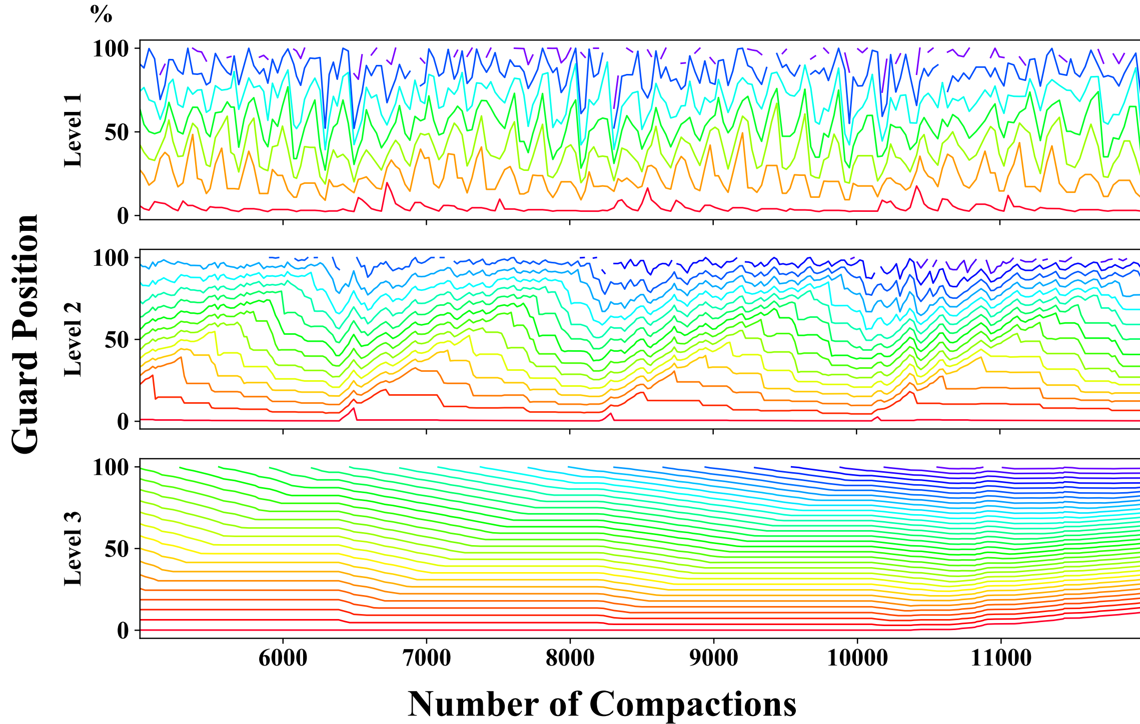


Figure 2.2: *Guard positions in different levels in LevelDB (L1, L2, and L3) after certain number of compactions in the system. The position is expressed as a percentage of the guard key in the entire key space* $(0..10^9)$*. A workload with the uniform distribution is used here.*

2.2.4 The Write-in-place Approach

To leverage the stable key space distribution, we propose to eliminate the top levels in the LSM-tree structure and partition the key space of the last level into equal-capacity buckets according to the workload's long-term key distribution. Within each bucket of limited capacity KV items are organized as a miniature LSM-tree, as illustrated in Figure 2.1b. While the proposed write-in-place (WipDB) approach similarly allows partially sorting in both horizontal and vertical dimensions, the novelty is to switch the order of the dimensions where the partially sorting technique is applied to make exploitation of the long-term stable key distribution possible.

LevelDB introduces the partially sorted structure on the vertical dimension (multiple overlapping sorted levels). PebblesDB further allows partial sorting on the horizontal dimension (multiple partially sorted buckets in each level). In contrast, WipDB first applies the partially sorting technique on the horizontal dimension by directly placing keys in the right buckets. It then uses the technique on the vertical dimension by using LSM-tree in

each bucket. The new architecture has two benefits. First, because the LSM-tree in a bucket is of limited size, WipDB can apply the approximate sorting technique to avoid rewriting in the same level without further partitioning any level of the tree into smaller buckets. Second, the sorting operation in a bucket can be scheduled according to the read request pattern to potentially move it out of read operations' critical path.

2.3 The WipDB Design

By leveraging the approximate sorting technique at only one level that contains a long list of data items and exhibits stability of a key distribution, WipDB can pre-define buckets in the key space and write incoming KV items into the buckets they belong to (write-in-place). Conceptually, WipDB mimics the bucket sort with KV items sorted across the buckets. Within a bucket, the items are managed within an LSM-tree. The design of WipDB addresses a number of critical issues, including how to partition the key space into buckets, how to efficiently write KV items into the buckets on the disk, how to prevent new KV items in the memory from being lost upon power failure or system crash, how to minimize the performance impact of in-bucket sorting, and how to adapt the buckets to change of key distribution.

2.3.1 The WipDB Architecture

In the WipDB architecture (Figure 2.1b), the key space is partitioned into a certain number of buckets so that each bucket is supposed to contain about the same number of KV items according to the observed key distribution. Each of the buckets admits new KV items directly from a buffer in the memory that corresponds to the bucket. The buffer is responsible for receiving new KV items whose keys are in same range of its corresponding bucket. As the buffer plays a role similar to MemTable in LevelDB [30], we name it *MemTable* too. The difference is that there are multiple MemTables, each for a bucket on the disk. When a MemTable is full, it's written to its corresponding bucket on the disk as a file containing a sorted list of KV items. This process is similar to the minor-compaction operation in LevelDB, where a MemTable becomes an SSTable in Level 0 of the LSM-tree on the disk.

As we mentioned, we use the LSM-tree structure within each bucket. To reduce write amplification due to compaction within the LSM-tree, we adopt the technique used in LSM-trie and PebblesDB, where a level consists of multiple overlapping sublevels, to avoid rewriting in the same level (see Figure 2.1b). Specifically, in a compaction operation KV items in the sublevels of Level i ($i = 0, 1, 2, \dots$) are merge-sorted into one list, which

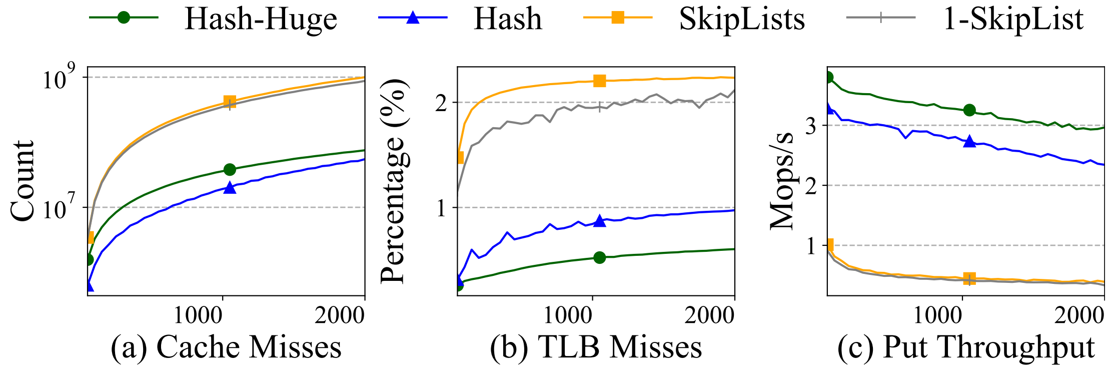


Figure 2.3: *Performance comparison of skip list and hash table.*

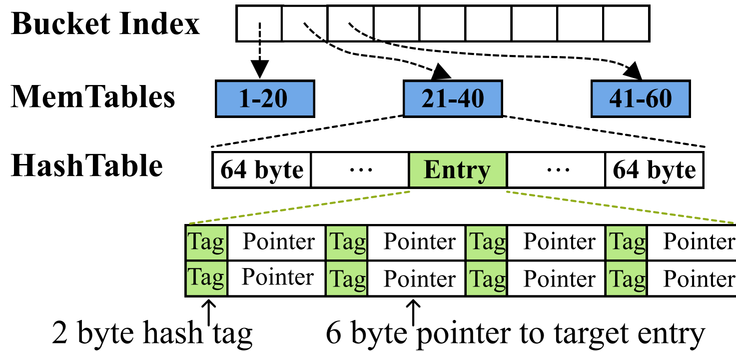


Figure 2.4: *MemTable Design*

is then written back as a new sublevel of Level $i + 1$ with a write amplification of one. Interestingly, though the three KV stores (WipDB, LSM-Trie, and PebblesDB) use the same technique for significantly reduced write amplification, only WipDB can take its full advantage. As we mentioned, when one big LSM-tree is used to manage a KV store, each level has to be partitioned into multiple segments, such as SSTables in LevelDB. The store conducts compactions on a few selected segments once at a time to cap the time of a compaction operation. To make the tree grow in a balanced manner, both LSM-trie and PebblesDB make a major effort attempting to maintain about the same number of items in each segment. This is challenging as short-term key distribution keeps changing. To this end, LSM-trie gives up support of range search by using SHA-1 hashed keys in the sorting. PebblesDB constantly adds guards in each level, incurring SSTable splitting operations and rewriting in the same level. WipDB addresses this issue by removing the need of introducing segments into a level. As a WipDB store can have a sufficient number of buckets so that each bucket won't grow very large (e.g, up to a 1GB). Each level is limited at a relatively small size (e.g., tens of Megabytes). This makes partitioning within

a WipDB's level unnecessary. Therefore, all sorted KV items in a sublevel of a bucket's LSM-tree are stored in one file, which is named *LevelTable*. A compaction is applied on multiple LevelTables of a level. In this way, the write amplification can be as low as the number of levels.

2.3.2 The Operations

WipDB supports all basic KV store operations, including write, deletion, modification, and read. Like most other KV stores, WipDB executes write, delete, and modification operations, collectively named *update operations* as writing new KV items to the store. In particular, for deletion operation a special KV item, whose value is a tombstone marker indicating it's a deletion request, is written. The actual deletion and modification are actually performed during compactions.

A read operation can request for either one KV item (point search) or all items in a key range (range search). A point search starts at the MemTable and proceeds across the LevelTables in the order of levels until a key is found or it reaches the last (sub)level in the corresponding bucket. In the process, use of Bloom filters in each sublevel can avoid most or all access of files that do not contain the requested key. However, for range search every LevelTable must be read and searched for the keys in the specified range. The results from the sublevels in all relevant buckets are combined and returned to the requester. Therefore, range search is an I/O-intensive and expensive operation.

2.3.3 Efficiency and Persistence of MemTables

In LevelDB, MemTable is maintained as a sorted data structure (skip list), so that it can directly support range search and be readily written to the disk as a Level-0 SSTable. However, such a design can be problematic for WipDB.

WipDB may have a large number (e.g., a few thousands) of MemTables. Its enlarged working set and weakened access locality may lead to a very high CPU cache miss ratio. A write is always preceded by a lookup in a MemTable for its insertion location. A lookup in a sorted structure, such as a skip list, may require multiple memory accesses and incur multiple cache misses. To illustrate this, we set up systems of different number of MemTables, each with a capacity of 10K KV items, and write random keys into them. As shown in Figure 2.3, KV items in each bucket can be organized as a skip list ("SkipLists"), a hash table ("Hash"), or a hash table with huge page enabled in Linux ("Hash-Huge"). We also include results for all keys in one big skip list ("1-SkipList"). The system setup is described in Section 2.4.1. As shown, using skip lists causes much more cache and TLB misses, and accordingly produces a write throughput much lower than using hash table. This issue of

using a sorted in-memory table is particularly serious when high-speed SSDs are used and the store has a low write amplification.

To address the issue, WipDB uses a hash table to implement a MemTable. As shown in Figure 2.4, each entry of the hash-table's directory has 64 bytes (the cacheline size), which consists of eight 8-bytes slots. The entry is 64-byte-aligned. One memory access can retrieve all its eight slots into the cache. Conceptually, each slot stores a KV item. In reality, we hash the key into a two-byte tag. The tag and a six-byte pointer pointing to the space storing the KV item are stored in the slot. The eight slots in an entry are used as a log. New KV items are appended at the end of the log. A lookup in the entry starts from current end of the log. When the hash table is full (i.e., any of its entry has overflown) WipDB freezes the table, sorts its data items, and writes them to the disk as a LevelTable at Level 0. Meanwhile, a second empty hash table is set up to continue admitting incoming KV items.

Any KV items in the MemTables hosted in the DRAM are subject to loss due to power failure or system crash until they turn into LevelTable on the disk. Therefore, WipDB writes any new KV items into a write-ahead-log before their requests are acknowledged.

2.3.4 Support of Range Search

The hash-table-based MemTable does not directly support range search. When a range-search request arrives at a MemTable, WipDB immediately sorts the data items currently in the hash table and place them in a one-time-use buffer, which is discarded after the range search has completed its scanning. KV items are copied, rather than moved, from the hash table to the buffer.

This design choice is in stark contrast with FloDB [6], a KV store dedicated for improving operating efficiency of the skiplist based MemTable. FloDB adds a hash table on top of the MemTable and keeps pushing KV items from the hash table to the MemTable. WipDB cannot adopt such a design. With such a two-level in-memory structure a read request still needs to search the skip list (unless the item has been found in the hash table), causing many cache misses, which compromises performance. Interestingly, FloDB cannot use WipDB's choice either as it assumes a MemTable as large as 192 GB to take full advantage of memory's high speed. In contrast, each WipDB's MemTable has only one or a few megabytes, or a few thousand KV items. While a range search operation is very expensive, it is well affordable to sort this relatively small number of items in a hash table. In addition, if a bucket receives a large number of range queries during a time window, WipDB replaces the hash-based MemTable with a skiplist-based one to reduce sorting overhead for this bucket. If no more range-query requests arrive after next minor

compaction, it changes back to hash-based MemTable. This adaptive strategy adjusts the MemTable structure of each bucket individually based on the workload, so that the sorting overhead can be minimized.

To ensure that items arriving after a range search request from being considered, WipDB leverages a global unique and monotonically increasing sequence number assigned to any incoming item in the order of their arrival. Such a sequence-number mechanism is also adopted in other KV stores such as LevelDB and RocksDB. When a search request is received, the sequence number currently available for assignment is attached to the search. During a search any items whose sequence numbers are equal to or larger than the sequence number with the search are skipped.

2.3.5 Bucket Splitting and Merging

WipDB does not pre-assign a large number of buckets when a store is initialized as the key distribution is not known yet. Instead, it has only one or a few initial buckets. When the store grows or the key distribution of the incoming KV items changes, a bucket may become too large, and have too many levels (or sublevels), which degrades read performance. In principle the WipDB's structure is similar to a hash table, whose bucket capacity also needs to be capped for desired lookup performance. The difference is that KV items in a WipDB are sorted across its buckets. For this reason, it can be much more efficient to reduce a bucket's size. Instead of reshuffling the entire store, WipDB can individually split a bucket once it reaches its capacity.

Assuming each level of the LSM tree in a bucket consists of maximally T sublevels. When a bucket reaches its capacity each of its levels consists of T full sublevels. We choose to evenly split the bucket into N smaller buckets when it exceeds its capacity threshold. We consistently apply the same set of $N - 1$ splitters at each sublevel to produce N segments. Items in the ith segments ($i = 1, 2, \ldots, N$) of all sublevels constitute one of the N new buckets. As WipDB grows incrementally from a small number of buckets initially to thousands of buckets by this bucket splitting, the choice of the splitters is important to balancing buckets. To this end, similar to the sample sort, for each sublevel we first choose $N - 1$ splitters that evenly partition it. Assuming there are L levels, or $L \times T$ sublevels, we then sort the list of the $L \times T \times (N - 1)$ splitters and choose $N - 1$ splitters that evenly split the splitter list for the bucket splitting. During the splitting, the bucket continues servicing incoming requests. When N new buckets are created, items in the MemTable are written to one of the new buckets according to their keys. Therefore, N new Memtables are created, each for a new bucket, to receive incoming items. In the meantime, the old bucket becomes read-only to serve read requests that cannot be satisfied in the new buckets. WipDB carries

out a full compaction to turn the old bucket into one sorted list. It then partitions the list into N segments according to the selected splitters and places them in the new buckets respectively as their last levels. Eventually, all requests can be processed by the new buckets and the old bucket is removed.

A bucket shrinks after repeated deletion of its KV items. While existence of small buckets does not compromise performance, it may increase number of buckets and thus memory footprint. Small buckets can be removed by merging them with their neighboring buckets, which helps reduce WipDB's memory demand. Admittedly, by having multiple MemTables WipDB uses more memory for its in-memory data structure than other KV stores such as RocksDB and PebblesDB. However, its demand on memory is still very small. For example, for a 5TB disk filled with KV items of around 512B each, the memory demand is only 2GB, which is negligible on a commercial server equipped with several hundreds of GB memory. Furthermore, its use of the additional memory brings significant improvement of write performance, as will be shown in Section IV. This benefit is not available to existing KV stores even if a much larger memory is offered.

WipDB's write amplification (WA) is mainly attributed to (1) compaction operations on LSM trees within individual buckets; and (2) bucket splitting operations. As we have discussed, with its use of vertical approximate sorting WipDB's compaction-induced WA is bounded by L_{max}, the maximally allowed level count in a bucket. Assuming a full bucket's size is 1 and its splitting produces N new buckets, each with a size of $\frac{1}{N}$. Each of the new buckets will be split again with the entire bucket being re-written after it grows from size $\frac{1}{N}$ to 1 with a minimal $1 - \frac{1}{N}$ data written into the bucket from users. Accordingly, the split-induced I/O write amplification is upper-bounded by the ratio between amount of write for bucket splitting and amount of write of new data from users, which is $1/(1 - \frac{1}{N})$, or $\frac{N}{N-1}$. Considering write amplification due to both compaction and bucket splitting, the store's WA is upper-bounded by $L_{max} + \frac{N}{N-1}$, regardless of bucket count or the actual store size. Assuming a practically configured WipDB store whose $L_{max} = 3$, $N = 8$, the total WA is no more than 4.15.

2.3.6 Use of Write Ahead Log for DRAM resident MemTable

In order to tolerate power failure, WipDB adopts a basic principle similar to existing KV stores, such as LevelDB, which writes a KV item to a write-ahead log (WAL) when it is still in the MemTable. The main issue for WipDB to address is how to reclaim the log space occupied by the KV items that have been safely persisted into the KV store itself.

In LevelDB KV items are persisted on the disk in their arrival order, which is also their order in the log. Each item is assigned a monotonically increasing sequence number in

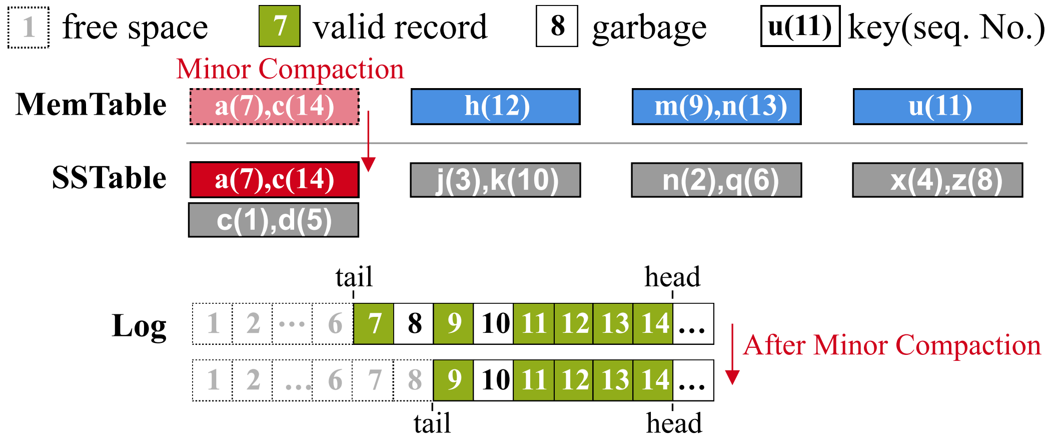


Figure 2.5: *Space reclamation in WipDB's Write-Ahead Log*

the order. Therefore, when an item with a particular sequence number is written to the disk, any items whose numbers are not larger than the sequence number in the log, or any items before a corresponding offset in the log file, can be removed. However, this is true only for items within individual MemTables in WipDB. WipDB has multiple MemTables. All new items are written to a common log. But they may be distributed in different MemTables.

We track the smallest sequence number in each MemTable among those whose corresponding KV items are not yet persisted. We then choose the smallest of every MemTable's smallest number. All items in the log whose sequence numbers not greater than this smallest number, which are contiguous in the log file, can be removed. In this way, the log space can still be efficiently reclaimed. The process is illustrated in Figure 2.5, where new items are written at the log head and free space starts at the tail. The smallest sequence number is at the tail (e.g., Sequence number 7). Note that there can be removable (garbage) items interspersed among valid items, such as items with Sequence numbers 8 and 10. After the leftmost MemTable is flushed to the disk, the tail moves forward to Sequence number 9. The space before the position can them be reclaimed and reused.

To prevent the log from becoming too large, WipDB sets up a threshold on the log size. When the threshold is reached, MemTables at the log tail are written to the disk to shrink the log.

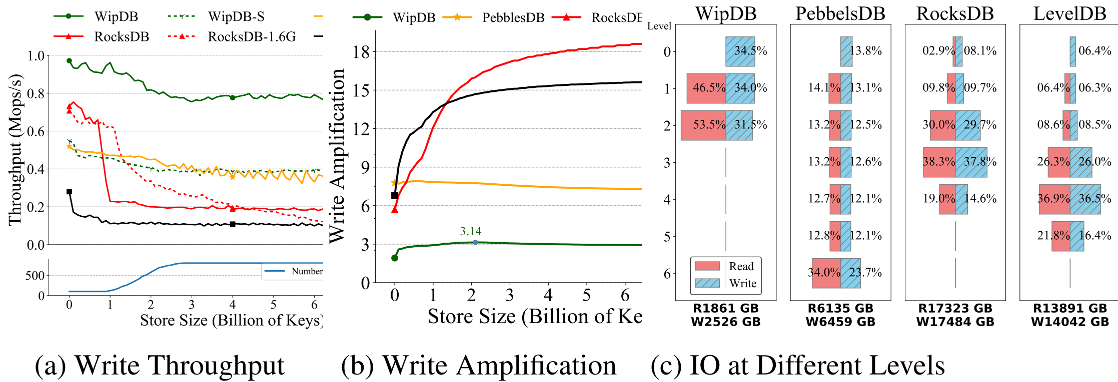
2.3.7 Read-aware Compaction Scheduling

In an LSM-tree-based multi-level KV store, read performance can be compromised by searching in SSTables in different levels. The more the levels, the more likely for Bloom filters in the SSTables to have false positives and for a read request to take more than one

disk access. To this end, compaction must be conducted to push new items downwards and reduce level count. However, intensive compactions can consume much I/O bandwidth and slow down the concurrent read requests. Should we be able to schedule compactions with priority on KV items being intensively read and reduce number of levels hosting these read items, the negative impact of compactions on read requests can be reduced. However, this is very difficult as LSM-tree's top levels are covered by a few SSTables. And it's hard to separate items being read from those being written into different compactions.

WipDB addresses the issue in its design. Each of its buckets, managed as a small LSM-tree, is only responsible for a fraction of the entire key space. As long as write and read requests do not intensively fall in the same buckets, WipDB can prioritize compactions on read-intensive buckets to reduce their levels and improve read performance. Even if the write and read key spaces are highly overlapped, it's likely the case where newly written items are immediately read in the following requests and the read requests can be serviced at a cache before they reach the storage system [82, 7]). For scheduling of compaction of all sublevels of a level in a bucket, WipDB considers two factors, which are number of current sublevels, denoted *sub_count* and number of times any of the sublevels are accessed to serve read requests since the last compaction of the level, denoted *read_count*. *sub_count* should be in the range of [*min_count*, *max_count*]. A level becomes eligible for compaction when it has at least *min_count* sublevels. This level receives the highest priority for compaction when it reaches *max_count* sublevels. Suppose average of eligible levels' *read_count*s is *avg_read_count*, the relative read count for a level of *read_count* reads is $\text{rela_read_count} = \frac{\text{read_count}}{\text{avg_read_count}}$. Also if the average of eligible levels' *sub_count* is *avg_sub_count*, the relative sublevel count for a level of *sub_count* sublevels is $\text{rela_sub_count} = \frac{\text{sub_count}}{\text{avg_sub_count}}$. The priority ($p$) of a level's compaction is qualified as $p = (read_weight) \times (rela_read_count) + rela_sub_count$, where $min_count \leq sub_count < max_count$.

The *read_weight* adjusts the weight of the read performance relative to importance of a balanced compaction across the buckets according to the sublevel count. The priority values are dynamically updated and the N levels with the highest values are selected for concurrent compactions. By adopting a large *read_weight*, WipDB allows read-intensive buckets to be aggressively compacted for high-performance read service. In the meantime, it leaves the write-intensive and read-little buckets lightly compacted to save more bandwidth and further improve read performance. Based on our empirical study, we set default values of *min_count*, *max_count*, and *read_weight* as 4, 20, and 10, respectively.



(a) Write Throughput (b) Write Amplification (c) IO at Different Levels

Figure 2.6: *Write performance. "WipDB" uses the hash table for MemTable. "WipDB-S" uses skiplist. In (c) percentages of read and write amount at each level of a store are marked in the graph. The total read and write amounts, including those for compaction, are shown under respective graphs.*

2.4 Evaluation

We implement a prototype of WipDB and evaluate it against three state-of-the-art KV stores: LevelDB (v1.20), RocksDB [26] (v5.18), and PebblesDB [62] (`git #220d0fa`). In the evaluation, we will answer the following questions:

- How does WipDB improve write performance?
- How effective is WipDB's MemTable?
- How does WipDB perform with workloads of changing key distribution?
- How does the compaction scheduling improve WipDB's read performance?

2.4.1 Experiment Setup

The experiments were run on a Dell T440 server with two 4-core Intel Xeon Gold 5122 CPUs and 64GB DRAM. To minimize the interference between threads or cores, hyper-threading is turned off from BIOS. The server runs a 64-bit Linux (v4.20.0) with an ext4 file system on an Intel 750 SSD (PCIe, 1.2 TB). The SSD has up to 1200 MB/s sequential write throughput and up to 440K IOPS for random 4KB reads.

In the evaluation, WipDB uses 2MB MemTable for each bucket. LevelDB, RocksDB, and PebblesDB use 64 MB Me-mTables. We configure WipDB with $L_{max} = 3$, $T = 8$ and $N = 8$. Another SSD of the same type holds the log file(s) to avoid impact of logging on systems' frontend operations in each of the four stores. Meanwhile, every 1000 write requests are logged as a batch for high efficiency.

2.4.2 Write Performance

WipDB aims to substantially reduce the high write amplification ratio (WA) of LSM-tree-based KV stores. To evaluate the performance improvement of WipDB for write-intensive workloads, we use 16-byte keys and 100-byte values with uniform distribution and send 8 billion write requests (around 900 GB data) to each store. WipDB is configured to have 100 buckets at beginning except stated otherwise. All the stores, except LevelDB which only supports one background thread, use seven compaction threads. We show throughput and WA of WipDB, LevelDB, RocksDB, and PebblesDB in Figures 2.6.

Write throughput and WA. As shown, WipDB has much higher throughputs than the other stores. This improvement is primarily due to its low write amplification (see Figure 2.6b). In particular, RocksDB and LevelDB's write amplification is about $5\times$ to $6\times$ higher than that of WipDB. Admittedly WipDB uses more memory for its MemTables. To reveal impact of this increase of MemTable size, we include an experiment of RocksDB whose MemTable is configured to be of 1.6GB, the peak size of WipDB's MemTables in the experiments. This RocksDB's results are shown as '*RocksDB-1.6G*' in Figure 2.6a. The throughput doesn't improve. A larger Memtable does help collect more writes for larger batched I/O and improved I/O efficiency. However, increasing the Memtable size will have diminishing return on fast devices and the dominant factor on the stores' performance is the amount of I/O, which is determined by a store's WA. PebblesDB's WA is also $2\times$ of that of WipDB [3] The total amount of I/O during the writes is shown in Figure 2.6c. PebblesDB's extra writes are caused by having more levels and constantly splitting SSTables to generate new guards, while WipDB maintains at most three levels and conducting bucket splitting only when a bucket reaches its the size limit.

Since WipDB has a consistently low WA, its throughput remains stable and high at about 0.8 Mops/s. It is worth noting that initially WipDB's WA increases as the store grows. At the moment it reaches its peak value, 3.14 as shown in Figure 2.6b, the overall WA drops slightly. The reason is twofold. On the one hand, as WipDB starts to split, the number of buckets grows, and more data can be stored at Levels 0 and 1. Hence the WA becomes smaller with fewer levels. On the other hand, the workload contains update requests, which makes the size of new SSTables generated by compaction smaller than the total size of the SSTables being compacted.

WipDB's MemTable. To improve its memory access efficiency, WipDB initializes with hash based MemTable, instead of skiplist. To assess the impact of this design,

[3] In the experiment with PebbleDB, we manually change its opensourced code by setting *top_level_bits*, a variable controlling probability of generating guards, from 27 to 31 to reduce number of guards. This is necessary to allow PebbleDB to finish its execution. Otherwise, the program would run out of memory (on our server with 64GB memory) when the store reaches two billion KV items.

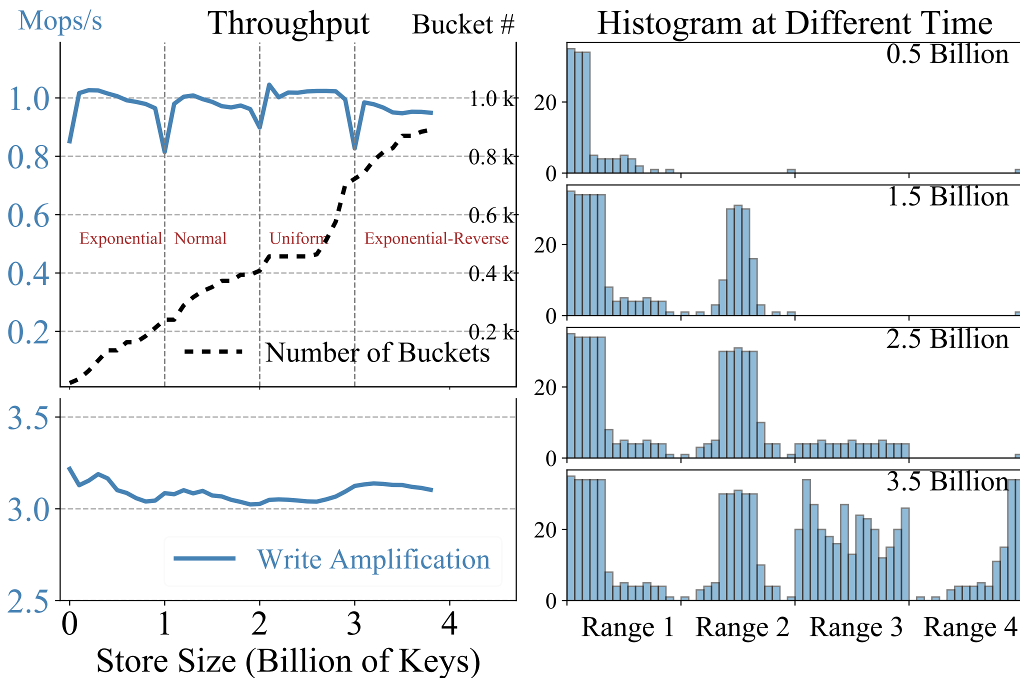


Figure 2.7: *Write throughput when key distribution changes. Blue bars in the right graph show the distribution of buckets (number of buckets in each 1/60 key space) in the key space at four sample time points when the store reaches 0.5, 1.5, 2.5 and 3.5 billion items, respectively.*

we include a version of WipDB that is initialized with skiplist-based MemTable, marked '*WipDB-S*' in Figure 2.6a. As shown, WipDB's write throughput is 2× of that of WipDB-S. WipDB-S's throughput remains stable at a lower rate (around 0.4 Mops/s), while its WA ratio is as low as that of WipDB. With the increase of bucket count from 100 to around 800, the working set of the MemTables grows much larger than the CPU cache size. Echoing observations shown in Figure 2.3, the degraded throughput of WipDB-S is due to its use of sorted skip list that causes much more CPU cache misses in its index walk than use of hash table. It is noted this issue of memory access performance arises only in the design of a high-performance KV store using high-speed storage devices.

Responding to changing key distribution.

Real-world workloads are usually skewed [4] and their key distribution is likely to change over time, causing some buckets in WipDB to grow much faster and having more (sub)levels than the others. WipDB addresses this issue with bucket splitting. To evaluate how WipDB responds to key distribution change, we run a WipDB store initially with only one bucket. In the meantime, we separate the entire key range into four equal-size and non-overlapping regions, and serve four write workloads with different key distributions, each

writing to a different region, one at a time, to simulate a workload of dynamical key pattern changes. The four regions, in the order of their key ranges, receive KV items (16 B keys and 100 B values) of exponential, normal, and uniform key distributions, and then exponential again with its key order reversed, respectively. Each region receives 1 billion KV items. The experiment results are shown in Figure 2.7. As shown in the left graph, the number of buckets increases with the increase of store size. The increase rate slows whey the workload switches to the uniform distribution without having skewed writes to rapidly fill and split a subset of buckets. The right graph of Figure 2.7 shows that the bucket distributions consistently match the key distributions in the key regions, demonstrating WipDB's adaptability in its bucket placement for equalized bucket sizes. During the execution, the WA may be modestly reduced. When buckets become filled and are then split into small ones with much fewer (sub)levels, their WAs accordingly become smaller. It is worth noting that the write throughput is the lowest at the moment when key distribution switches. Because new buckets are always generated by splitting existing buckets, this bucket splitting for accommodating new a key distribution leads to the throughput loss.

Note that in the WipDB's design, we do not assume an advance knowledge on number of buckets to be used and how the buckets should cover the key space (the store can be initialized with only one bucket). We only assume a long-term relatively stable key distribution to prevent an extreme scenario that may lead to excessively large number of buckets. In the scenario, some buckets receive many KV items and are filled. They are then split into new near-empty ones. The new buckets would receive few KV items afterwards. The consequence is that a very large number of buckets/MemTables leads to serious cache misses and compromised system's performance. However, from the experiment results we understand that the assumption on the key distribution does not have to be strong. The system's performance is highly tolerant to the change of the distribution and bucket count increase.

2.4.3 Read Performance

To evaluate read performance, we first build a store of 1 billion KV items (about 100GB). We then use eight threads to read items until another thread finishes sending 300 million write requests at a rate up to 150 Kops/s (by inserting time delay between requests). The read and write throughput is shown in Figure 2.8 (Note that the read Y axis is on the right of the graphs). In the meantime, to evaluate the impact of WipDb's read-aware compaction (RC) design, we include a version of WipDB that disable RC, marked 'WipDB-DRC'.

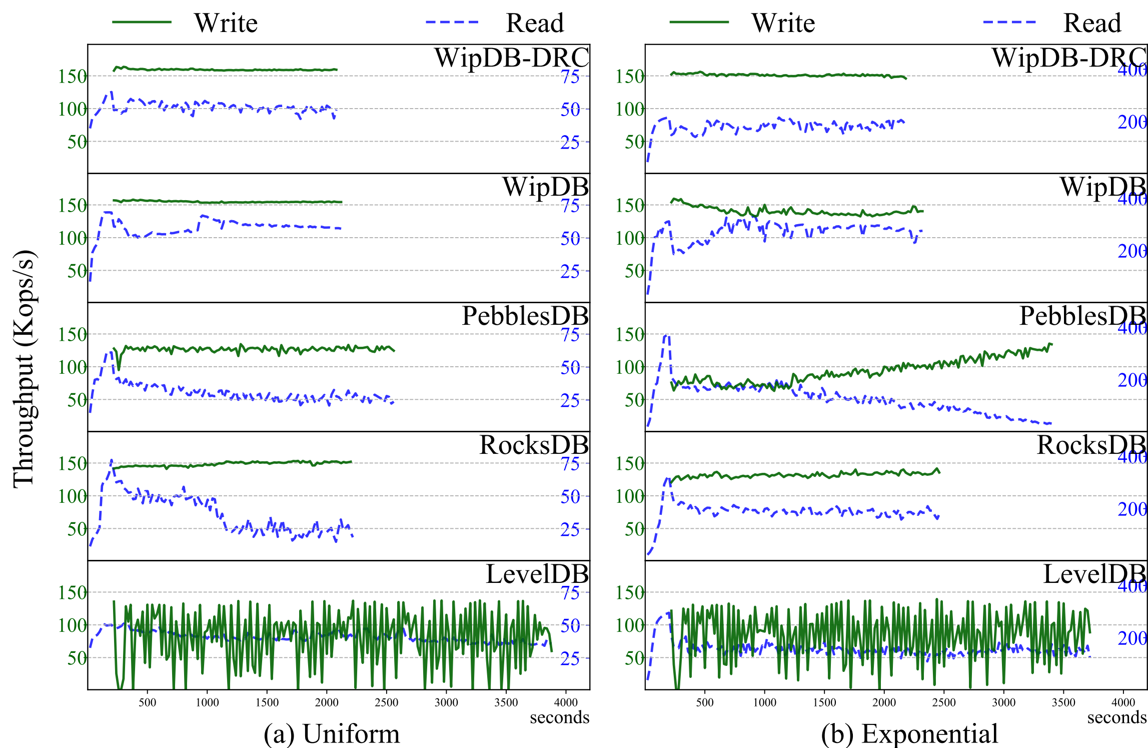


Figure 2.8: *Throughput with mixed read/write requests. One thread sends 300 million random (uniform) write requests at the rate of 150 Kops/s(if possible). Eight threads send read requests until all the writes finish. WipDB that Disable Read-aware Compaction is marked as 'WipDB-DRC'.*

In the experiment where read requests have no locality (uniform), as shown in Figure 2.8(a), read throughput of all the four stores become lower after the write requests arrive. However, only WipDB sees its read throughput recovered during the writes, while others' read throughput keep dropping and then stay at its low level. This is mainly due to WipDB's consistently lower WA values. *WipDB-DRC*'s read throughput shows slight difference with *WipDB*. As there exists weak access locality, RC makes a minor contribution to improving performance. LevelDB uses only one thread for compactions and aggressively compacts SSTables to the last two levels. Accordingly its write throughput is intensively fluctuated.

We then repeat the experiment with exponential key distribution for the read requests. The results are shown in Figure 2.8(b). With a very strong access locality the read throughput is multiple times higher than that with the uniform key distribution. When the write requests arrive, all stores, except WipDB, observe much lower read throughput. Only WipDB roughly maintains its read throughput with some fluctuations. During the time pe-

riod, WipDB's read throughput often more than doubles that of other stores. As shown, WipDB-DRC's read throughput is 30% lower than WipDB. This is apparently the consequence of WipDB's read-aware compaction scheduling design that leaves much of the compaction in the key range with light reads off the read requests' critical path. WipDB identifies read-intensive buckets and applies more aggressive compaction with priority on them. Therefore, the buckets that serve most read requests can be compacted faster and have fewer sublevels for fast read. Therefore, among all the stores WipDB has the lowest read latency, as shown in Table 2.1.

Table 2.1: Read Latency for 99 Percentile

Store	Uniform	Exponential
WipDB_DRC	439 us	247 us
WipDB	365 us	190 us
PebblesDB	1698 us	324 us
RocksDB	765 us	293 us
LevelDB	526 us	249 us

2.4.4 Impact of WAL on Restart Time

A WipDB store has hundreds of or even more MemTables. New KV items from different MemTables are written to a common log. This will lead to a log file much larger than those for stores using only one MemTable. When the store unexpectedly crashes and requires a re-start, it may take a longer time period to read the log for a restart. Figure 2.9 shows the log size and the restart time when a store is built with write requests of different key distributions at a size in terms of number of buckets. Initially, the log and the restart time grow linearly with the number of buckets. When the WipDB store reaches around 400 buckets, the log size and the restart time stay stable at around 1.5GB and 12 seconds, respectively.

2.4.5 Results of the YCSB Benchmarks

The Yahoo! Cloud Serving Benchmark (YCSB) [16] is a popular benchmark used to evaluate performance of NoSQL databases. We modify *db_bench* to support YCSB benchmarks and run eight threads and send 8 million requests for each workload. All the stores are pre-loaded with 1 billion (around 100GB) KV items. The results of the benchmarks are shown in Figure 2.10.

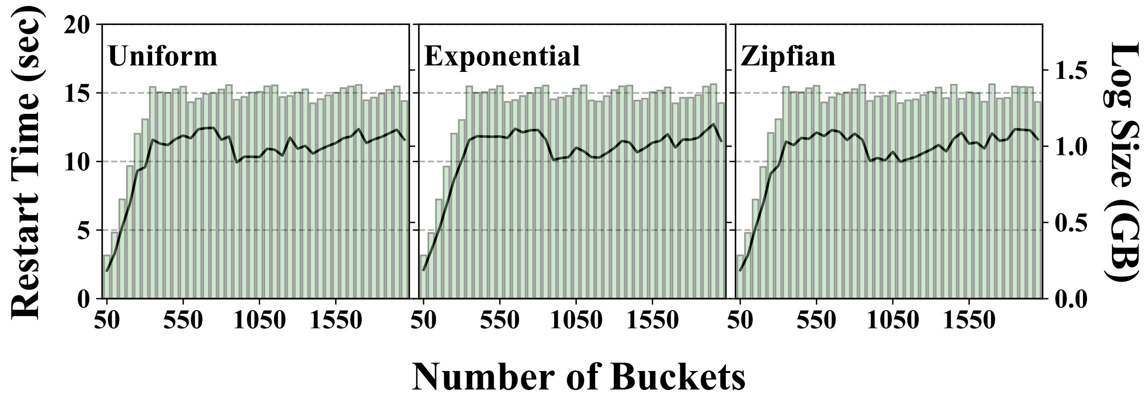


Figure 2.9: *Restart time (curves) and log size (bars).*

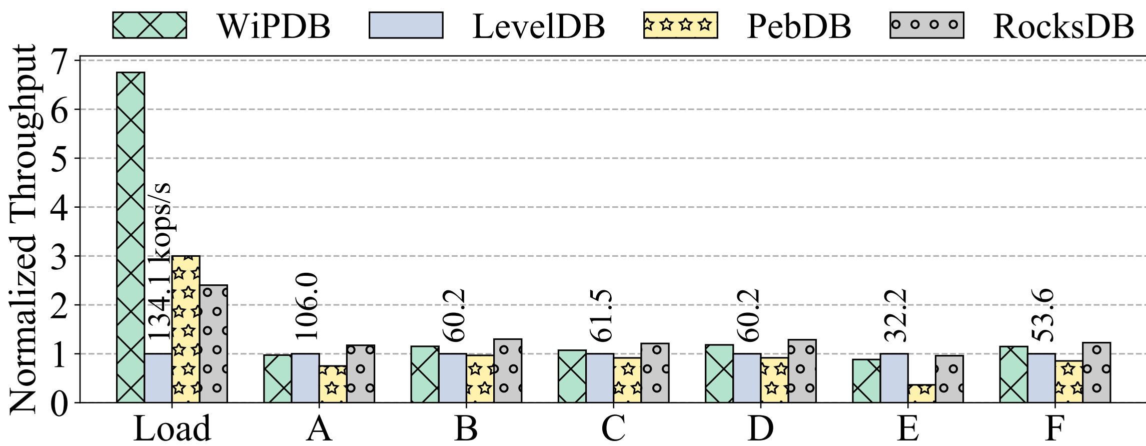


Figure 2.10: *Throughput of YCSB benchmarks*

For the all-write workload (“Load”) that pre-loads 1 billion items into the store, as expected WipDB has a much higher throughput than other stores. The other workloads consist of only or mostly read requests. WipDB is proposed to mainly optimize write operations so as to improve performance of writes and that of reads concurrent with writes. When writes accounts for only a fraction of total I/O load in the workload, WipDB is not expected to make a substantial difference. As shown, Workloads B and C have a 95/5 reads/write mix and a 100% read, respectively. WipDB’s read throughput is comparable to others’ performance. It’s a little lower than that of RocksDB. Note that RocksDB is a production-level system that has been carefully engineered with numerous optimizations. WipDB is an experimental prototype with only limited optimization efforts. Therefore, this small performance gap is not a surprise. Though Workload A has a 50/50 reads/write mix, its 50% read requests contribute over 90% of the execution time (as each read usually needs

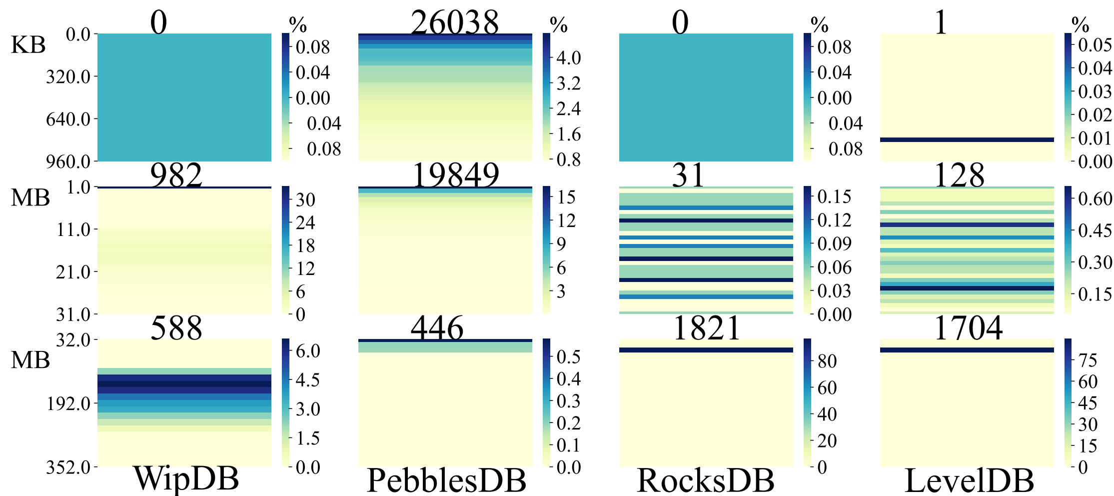


Figure 2.11: *File size histogram. The number above each figure is the number of files within the size range.*

Table 2.2: Latency for 99 Percentile

Store	**A**	**B**	**C**	**D**	**E**	**F**
WipDB	349	344	343	323	1133	351
LevelDB	253	342	328	341	688	364
PebblesDB	443	534	543	603	3008	671
RocksDB	237	241	241	241	638	248

to read an entire data block). Therefore, the performance gap still exists. Nevertheless, for almost all the workloads, WipDB outperforms LevelDB and PebblesDB. Workload E consists of short range queries that require searching a store's every (sub)level. WipDB has more sublevels than levels of LevelDB and RocksDB, and thus has a higher read amplification. The read latency (us) is shown in Table 2.2. WipDB's latency is comparable to that of LevelDB, expect for workloads A and E. PebblesDB has the highest latency because the randomly chosen guards cause file fragmentation. As shown in Figure 2.11, more than half of its file are smaller than 1 MB. Meanwhile, PebblesDB has over 20× files than the other store, causing more file-system overhead.

2.5 Related Work

LSM-tree has become the most popular data structure for the storage layer of NoSQL databases due to its optimized write performance compared to other structures such as B+-tree. LSM-tree achieves this by avoiding expensive in-place writes and moving the internal

data reorganization to the background. However, write amplification in LSM-tree-based systems can still easily go over $10\times$, which leaves a wide gap between the user-perceived performance and that offered by the low-level storage devices, the SSDs. Because of this, improving write efficiency in LSM-tree-based KV stores has become a daunting task for persistent KV stores. This section discusses representative works on amelioration of the high WA.

2.5.1 Optimizations for efficient compaction.

Real-world KV workloads often demonstrate skewed patterns [4], which have been leveraged by researchers to apply specialized optimizations. For example, bLSM [67] uses a merge scheduler that aims to minimize write stalls by coordinating compaction operations across multiple levels. Thonangi et al. [69] introduces ChooseBest, a compaction policy that selects an SSTable at L_k with the fewest overlapping SSTables at L_{k+1} to minimize the merge cost. Skip tree [79] allows KV items to be written to a deeper level without going through the level-by-level merges. VT-tree [68] reduces disk writes by reusing existing data in the old tables. TRIAD [5] takes advantage of the skewed workload where hot keys are likely to be short-lived. It identifies the hot keys and keeps them in the MemTable and the WAL instead of moving them to the Level-0 SSTables. In this way, the write traffic to the top levels can be effectively reduced. The above optimization are orthogonal to WipDB and may help to further improve the efficiency of WipDB.

2.5.2 The tiering merge scheme.

In LevelDB [30] and RocksDB [26], compaction operation needs to rewrite a significant amount of data at the deeper level (L_{k+1}) but moves only $1/N$ of that amount of data from L_k. The tiering merge scheme was proposed to eliminate the significant rewrites. By merging multiple SSTables from $Level_k$ and writing to $Level_{k+1}$ without rewriting any $Level_{k+1}$ data, write amplification can be effectively reduced to only L_{max}, the number of levels in the store. wB-Tree [14] uses a B^+-tree-like structure to organize the tables to maintain a small L_{max}. Similarly, LSM-trie [73] uses a prefix-tree (trie) structure for the same purpose. However, since both wB-tree and LSM-trie depend on hashing to maintain balanced tree structures, they gives up the ability of performing range operations. sDB [62] employs a probabilistic method to partition the keys at each level to enable tiering with the range-query capability retained. SifrDB [51] also employs tiering. In LSM-Bush[21] and Dostoevsky [20], a lazy-leveling scheme is introduced to use tiering for levels from 1 to $L_{max} - 1$ and use leveling at Level L_{max}, the last level. In this way, the WA is $O(L_{max} + T)$, where T is the size ratio between adjacent levels. Different from

the above schemes, WipDB employs a partitioning approach to limit its bucket's size and accordingly the number of levels. As a result, WipDB achieves a lower write amplification ($O(L_{max} + \frac{N}{N-1})$) (N is the split-ratio) without sacrificing read efficiency. Both of L-Store [65] and WipDB aim to improve write performance. Additionally, L-store converts the data from a write-optimized organization to a read-friendly one to serve read-intensive OLAP workloads. This is similar to regular LSM-based stores that finally compact all data into the last level if writes do not keep coming. WipDB is designed to opportunistically improve read performance even with write requests by prioritizing hot buckets' compaction for fast read.

2.5.3 Key-value separation.

It is observed that rewriting the values in KV items can contribute to a major amount of I/O in the compaction for relatively large values, compared to the size of metadata and key which is usually tens of bytes. WiscKey [50] proposed a simple yet effective method, KV separation, to write values into a log and keep them from the compaction. However, the value log requires regular GC operations to reclaim free space. The log GC can be particularly expensive under skew real-world workloads. Significant amount of cold data needs to be consistently removed, which drives up the store-wise write-amplification to up to 20$\times$ and offsets the benefit of KV separation [10]. To reduce this GC overhead, HashKV [10] replaces the log with a sophisticated mechanism that divides the log into partitions and separates the cold items from the hot data, which again shifts the overhead to read by adding another layer of indirection. WipDB solves the high write-amplification issue by directly partition the key space without creating any indirection.

2.5.4 In-memory key-value stores

. For applications demanding high concurrency and low latency, in-memory key-value stores are always preferred. HotRing[13] is proposed as a hotspot-aware and lock-free design to speed up multi-core performance for highly skewed workloads. Redis and Aerospike [19] provide a hybrid solution which provides memory-access speed as well as on-disk data persistency when specified conditions are met. In contrast, WipDB is still a on-disk KV store providing always data persistency and expecting on-disk data access for most read requests.

2.6 Summary

We introduce WipDB, a key-value store designed to manage small key-value items in a storage system of large capacity. By introducing approximate sorting and the write-in-place LSM-tree scheme, WipDB minimizes write amplification for LSM-tree-based KV stores. Meanwhile, the read-aware scheduling of compaction moves most compaction off the critical path of read service. Our results show that WipDB can significantly improve for both write and mixed read/write workloads. Source code of our WipDB implementation is available at https://gitlab.com/sjiang-lab/wipdb .

CHAPTER 3

TURBOHASH: A HASH TABLE FOR KEY-VALUE STORE ON PERSISTENT MEMORY

Major efforts on the design of a persistent hash table on a non-volatile byte-addressable memory are on efficient support of crash consistency with fence/flush (first fence and then flush operations) primitives and on table rehashing. When a data entry in a hash bucket cannot be updated with one atomic write, out-of-place update, instead of in-place update, is required to avoid data corruption after a failure. This often causes extra fences/flushes. Meanwhile, when open addressing techniques, such as linear probing, are adopted for high load factor, the scope of search for a key can be large. Excessive use of fence/flush and extended key search paths are two major sources of performance degradation with hash tables in a persistent memory.

To address the issues, we design a persistent hash table, named TurboHash, for building high-performance key-value store. TurboHash has a number of much desired features all in one design. (1) It supports out-of-place update with a cost equivalent to that for an in-place write and lock-free read. (2) Long-distance linear probing is minimized (only when necessary). (3) It only conducts shard resizing for expansion and avoids expensive directory-level rehashing; And (4) it exploits hardware features for high I/O and computation efficiency, including Intel's Optane DC's performance characteristics and Intel AVX instructions. In particular, all of the features are enabled with the consideration of a critical performance characteristic of the emergent Intel Optane DC persistent memory, which is its internal 256-byte block access. We have implemented TurboHash on the memory and conducted extensive evaluations. Experiment results show that TurboHash improves state-of-the-arts by $2\times$-$8\times$ in terms of throughput and latency.

3.1 Introduction

A hash table is a fundamental data structure for efficient organization of key-value (KV) data in the memory. It allows data to be quickly located with few intermediate index search. This is especially important for organizing a very large number of small KV items, where often it may take only one cache-line memory access to retrieve a data item. Had many non-sequential memory accesses been required on an index structure, such as B+

tree or skip list, the actual cost of reading a small piece of data would be amplified by multiple times [72, 53]. Accordingly, hash tables have been employed to manage key-value cache in the DRAM, such as MemCached [55, 28] and Redis. With emergence of byte-addressable non-volatile memory, efforts have been made to design persistent hash tables on the memory [84, 52, 15, 44, 83, 48, 86, 43, 71, 85, 32], as well as using persistent memory to build KV stores [76, 24, 38, 37, 78, 81].

One of the major issues and challenges on designing a high-performance persistent hash table is on its efficient support of crash consistency and atomic update. With crash consistency, a data structure can stay in a consistent state, or be restored to a consistent state after an unexpected crash. To have the consistency, one has to enforce a particular order on a set of actions. A simple example is that a hash table's bucket has to be allocated and initialized before its address can be assigned to a pointer in the table's directory. To enforce the order, fence/flush primitives, which are expensive, have to be used between the operations. More extensive use of fence/flush is required for updating a directory during a rehashing operation.

For data integrity, updating of a piece of data, such as a key or a value, must be atomic. After a recovery from an unexpected crash, either a version of the data before the update or the one after the update must be recovered. This means that the data cannot be modified in place if it is larger than an atomic write unit (8 bytes). Otherwise, it may destroy the old version without making the new version established at the time of a crash. Therefore, one has to use out-of-place write. However, this raises two potential performance issues for a modify/delete request as it needs to write at different places (creation of a new version and invalidation of the old one). First, currently available persistent memory, i.e., Intel Optane DC [35], has a 256-byte access unit (block) [77]. Two writes at two different blocks are significantly more expensive than one block write. Second, ideally the new version can be made visible and the old version is invalidated with one 8-byte atomic write. Otherwise, a write order has to be established between the two writes using a flush/fence.

Another major issue in the design of a hash table is how to resolve collision with both time and space efficiency. Upon occurrence of a hash collision in a full bucket, there are three approaches to resolve it. Approach A is to double the size of the hash table via rehashing to make each bucket be only about half occupied. Approach B is to apply Approach A only in a segment of buckets where the collision occurs to avoid global key reshuffling, such as extendible hashing [27]. And Approach C is to look for an idle slot in alternative buckets where the colliding key can be placed, such as cuckoo hashing [59]. Apparently, these approaches (from A to C) become increasingly less disruptive to existing hash structure and more time-efficient. More extensive structural changes will lead to more space allocations, pointer assignments, and key relocation. Using a less disruptive approach

can greatly help reduce impact of collision on the performance, especially on persistent memory where additional costs have to be paid for crash consistency.

Approach C is usually known as open addressing. Example techniques include linear probing, quadratic probing, cuckoo hashing, Horton Tables [8], and Hopscotch hashing [31]. However, not all of the schemes play well with the persistent memory, such as Optane DC. For example, cuckoo hashing and its variant Horton Tables need to constantly relocate keys along a path to reach an idle slot. It has to frequently use expensive fence/flush. Schemes such as quadratic probing and cuckoo hashing often write new keys to non-consecutive buckets. For a read request to locate the key on its probing path consisting of non-consecutive buckets, its access speed is much lower than linear probing where the probing path represents a contiguous memory space. Optane DC memory has a 256B access unit and is capable of sequential prefetching, with a sequential access speed 2-3X higher than that of random access [77].

While a linear probing can be efficient, there are some critical issues to be addressed. First, the adjacent buckets on a probing path must be physically contiguous for search efficiency; Second, the probing scope must be sufficiently large for a high load factor and less frequent rehashing. Load factor is the size ratio of the space used for holding actual data and allocated space. It quantifies the space efficiency of a hash table. Without a carefully designed key placement policy, a search, especially a negative search (whose search key is not in the hash table), may have to cover most or all buckets in a probing scope, compromising read performance.

Our Solution In this paper, we propose a persistent hash table design, named TurboHash, for a high-performance key-value store on the persistent byte-addressable memory (Intel Optane DC), or Pmem in short, by addressing all of the aforementioned issues. TurboHash first hashes keys into multiple shards. Each shard is a small hash table, which is set at a limited capacity (e.g., 1MB for 53,000 16-byte KV items) to cap the worst-case time, or tail latency, of requests serviced within a shard. In the meantime, we lavishly pre-allocate shards so that a TurboHash store's capacity limit can be way higher than the size of the physical Pmem a machine can can actually have. As each shard consumes only 8-byte metadata, this over-provisioning is well affordable. As an example, for a TurboHash with one million shards to provide an 8TB capacity limit (assuming a 8MB shard capacity), the space cost of representing shards is only 8MB. In return, this shard over-provisioning strategy avoids expensive resizing (or rehashing) over the entire hash table that seriously compromise tail latency. Instead, there are only localized and small-scale resizing within individual shards to minimize tail latency.

TurboHash achieves much desired features, such as lock-free reads and short probing paths, which are often objectives of other hash-table optimization efforts. A unique

contribution of TurboHash is its novel design that enables these features by accommodating Intel Optane Pmem's performance characteristics. The byte-addressable Pmem is more like a block device in terms of its performance behavior [81]. The Pmem has a 256-byte block access unit. Any small random access, such as updating of a few bytes of metadata, in the table may result in over 10X read/write amplification. Prior optimization techniques assuming an in-DRAM hash table may become ineffective for the Pmem. In one example, for lock-free reads, existing works, such as CLevel [15], carry out updating of a KV item usually by creating a new version and atomically switching the pointer pointing to the item from its old version to the new version. While the pointer and new version of the (small) item are often in two different 256B blocks, there would be two block writes, which is not efficient. In another example, to avoid holes due to deletions in a linear probing path, it has been suggested to move the item at the path tail to fill a hole [40]. However, this may involve two writes at different Pmem blocks: one is to invalidate the tail item, and the other is to write it into the hole.

To this end, TurboHash makes a number of innovative design choices to confine multiple writes/reads of data and metadata within one 256B Pmem block to unlock the Pmem's full performance potential. In particular, TurboHash's buckets are of 256 bytes each and are physically contiguous. It proposes to use near-place update (in the same bucket where the old version stays), instead of out-of-place, and one atomic write to respectively invalidate/validate old/new versions of a KV item within the a 256B bucket to enable efficient in-Pmem lock-free reads.

In summary, we make a number of contributions:

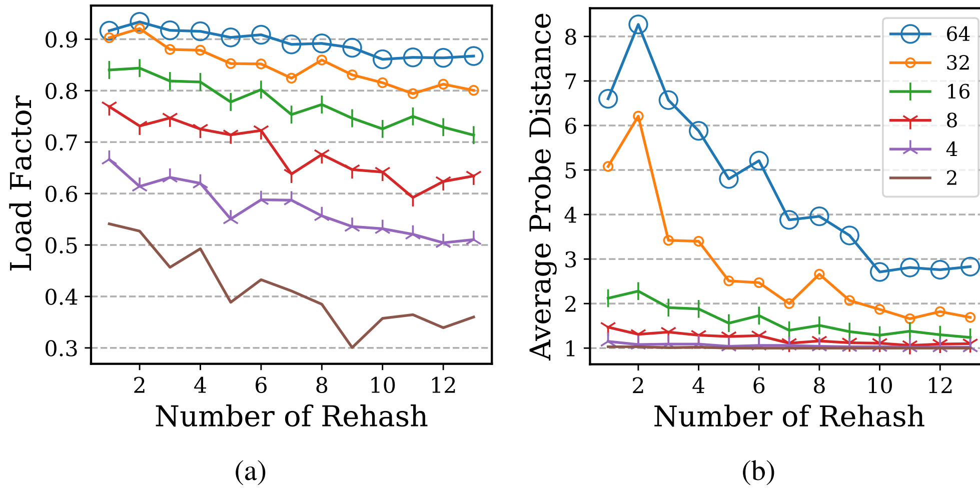
- Recognizing that a large probing scope is important for a high load factor and infrequent rehashings, we experimentally reveal that actual probing distances can be much shorter than the probing scope. We then propose a strategy that enables only necessary probing distances, especially for negative search.
- We tailor TurboHash's design for a large-scale KV store on persistent memory with its sharded structure and physically contiguous bucket layout for minimal use of fence/flush and efficient memory access.
- TurboHash introduces lock-free reads and efficient near-place updates.
- This work represents a first effort of extensively exploiting Intel Optane DC Pmem's block-access-like performance characteristic in its design to customize hash-table optimization techniques to the first widely-deployed Pmem.
- We evaluate TurboHash on Intel Optane DC memory in comparison with recently proposed persistent hash table designs, such as CCEH, dash, and clevel hashing. Experiment results show that TurboHash has $2\times$ to $8\times$ improvements in terms of throughput and latency.

3.2 Motivations

In this section we present experiment measurements on characteristics of linear-probing-based hash tables and Intel Optane DC Pmem to serve as rationale of TurboHash's design.

3.2.1 Probing Scope and Distance.

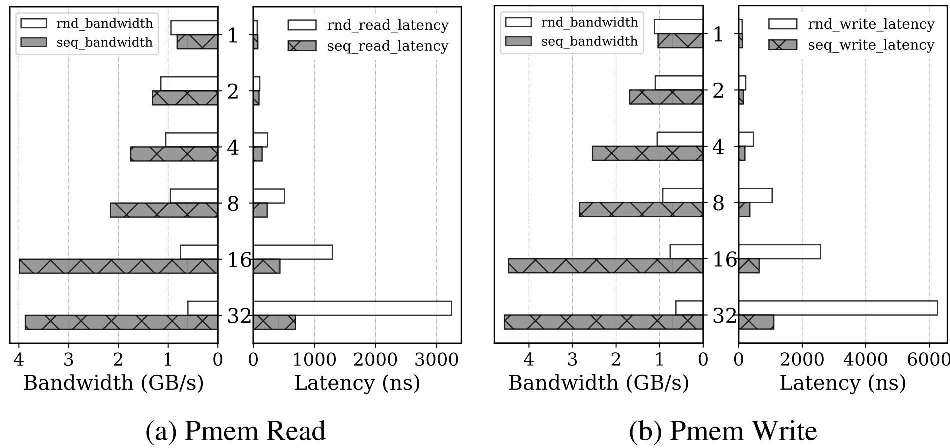
In a hash table using open addressing, the number of alternative buckets where a new key under collision can be placed is highly correlated to the table's load factor. The more alternative buckets, the more leeway a key has for its placement at locations other than its home bucket (the bucket initially given by the hash function). A key's probing scope refers to the set of buckets this probe is likely to reach (starting from its home bucket). Increasing the scope is important for high space efficiency. We assume a new key is placed in the first bucket with idle slot(s) on its linear search path. A key's search path grows within its probing scope when more keys are hashed to its home bucket. The path ends at the bucket where the newest key to the home bucket is inserted. In theory, a key probing can terminate at the last bucket on the path. The probing distance is the actual number of buckets a probe has to traverse on the path to either find the search key (positive search) or declare the key doesn't exist in the table (negative search). Therefore, the distance is capped by the path's length and can be shorter than the probing scope's size. Still, increasing the probing scope allows for longer probing distances, which may make a key search more expensive. We study the relationship between probing scope, load factor, and probing distance. To this end, we set up a linear-probing-based hash table of 1024 buckets. Each bucket has 16 16B-slots. The table is rehashed by doubling its size whenever a collision cannot be resolved within a probing scope. We keep inserting keys to an initially empty hash table with a given probing scope size. Figure 3.1a shows load factors right before every rehashing. As shown, the scope size has a significant impact on the load factor. Small scopes, such as 2 or 4 buckets, can lead to too-low load factors. A search path's length represents the necessary probing distance of a negative search for a key, as the key doesn't exist beyond the path's end. Figure 3.1b shows average search distance corresponding to load factors in Figure 3.1a at different probing scope sizes. As shown, the necessary search distance can be much shorter than the corresponding probing scope. For example, at the 6th rehashing the load factor is 0.8 with the 16-bucket scope. But the average probing distance is only 1.7 buckets. This implies that as long as the search path is recognized, the actual probing distance can be short for high access performance, and a large probing scope can be used for high load factor.



(a)

(b)

Figure 3.1: Load factors and average probing distances with different probing scope (in number of buckets) during insertion of 100 million KV items.



(a) Pmem Read

(b) Pmem Write

Figure 3.2: Random and sequential performance of the Pmem. The measurements of performance (throughput and latency) are collected via the PCM (Processor Counter Monitor) tool. For writes, each one is followed with a *clwb* to flush the data, and each group of writes is followed with *mfence*.

3.2.2 Sequential and Random Accesses.

To illustrate the performance gap between sequential and random accesses on a persistent memory (Intel Optane DC), we experimentally simulate access patterns where each probing is either on random memory locations or sequential locations. Each probing consists of a group of 8-byte accesses. These accesses are on different 64-byte memory spaces (simulating buckets), which are either randomly or sequentially placed. A probing always starts at a random location. The access group size is set to 1, 2, 4, 8, 16, or 32 (simulat-

ing probing distance). Figures 3.2a and 3.2b show the read/write throughput and average latency for each access group. As shown, for a reasonably large group, such as those with 4 or more accesses, sequential performance is significantly higher than that of its random counterpart in terms of either throughput or latency. The gap becomes even larger with a larger group. With a group of 16 accesses, the performance gaps are about 4-7X. The reason is that Intel Optane DC is accessed in the 256B unit. With a random access of 8 bytes, one 256B block is actually accessed at the persistent memory's media [77], causing a significant read or write amplification. A sequential access can also benefit from the prefetching mechanism available in the memory.

3.3 The Design of TurboHash

There are a number of challenges we must address to achieve TurboHash's design objectives, including high load factor, sequential and short search path, support of out-of-place updates, crash consistency, lock-free reads, and minimal use of fence/flush. In the design we will address these challenges: (1) how to limit a search within necessary distance, rather than the entire probing scope, especially for non-existing keys? (2) how to switch from old version to new version of an KV item in one atomic primitive? and (3) how to leverage hardware features such as Intel Optane's 256B access block and SIMD instructions for higher I/O and computation efficiency?

3.3.1 The Architecture

As we have stated, the entire key space is partitioned into many shards. Keys in each shard are organized in a hash table. Each of the hash tables can have thousands of 256B buckets. It supports efficient resizing, high load factor, and high-performance access. By using a highly randomized hash function, such as MurmurHash [2] or MD5, keys are uniformly hashed into the shards.

As shown in Figure 3.3, each shard has a descriptor. All descriptors form a shard directory. The directory stays in the persistent memory (Pmem) for its persistency, and is mirrored in the DRAM for access efficiency. Each shard descriptor records bucket count in the shard and a pointer to the shard. There are differences between shard descriptors in the Pmem and in the DRAM. In the Pmem, it includes two sets of bucket count and pointer to allow out-of-place updating of the count/pointer after a shard resizing. Additionally, it has a 1-byte version number indicating which set is currently in use. After writing a new set of count/pointer, an atomic update of the version number makes it effective. Afterwards,

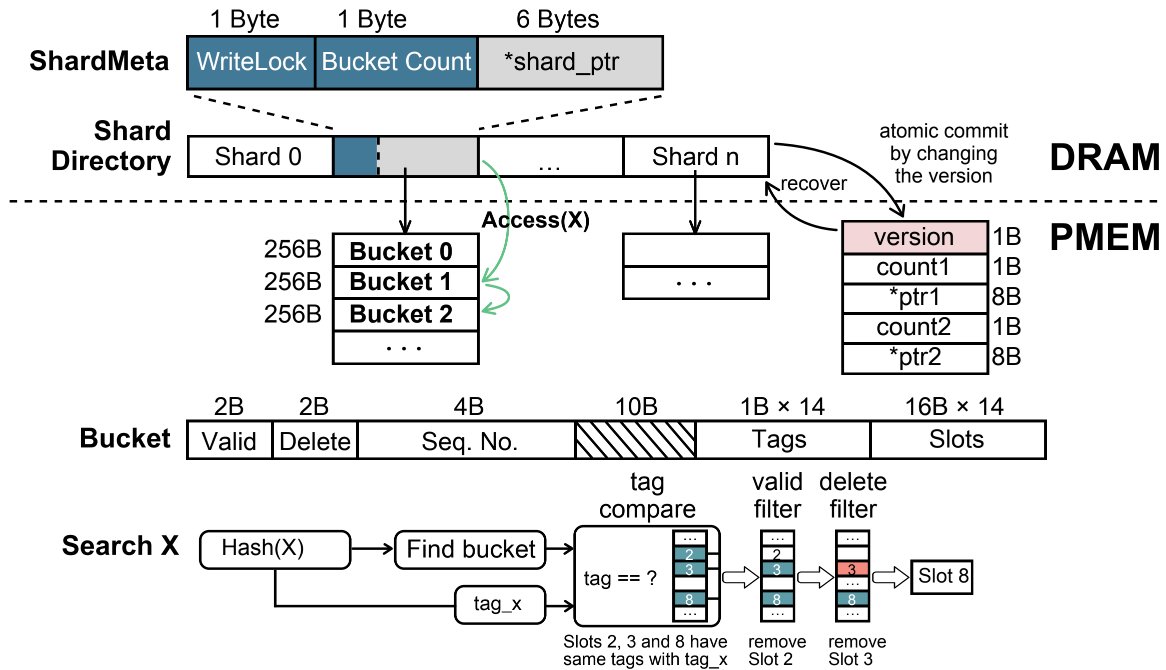


Figure 3.3: TurboHash's Architecture. In TurboHash, each bucket has 14 slots. Among them, 13 slots are available for storing KV items. TurboHash supports any size of KV items. When key or value is larger than 8 bytes, the slot in a bucket stores an 8-byte hashed key and an 8-byte pointer to the KV item, which is stored at a separate space.

count/pointer in the corresponding descriptor in the DRAM are updated. And the old shard will be recycled using the epoch-based reclamation [29].

There is a write lock (WriteLock) in the in-DRAM descriptor that establishes mutual exclusion among service of write requests (i.e., insert, update, or delete) in a shard. Because there can be many (a few thousands or more) shards in the hash table, the impact of the lock on concurrency is limited. In particular, this lock is only applied on write requests, which are more expensive as they likely involve writing KV items, updating metadata, and even shard rehashing. In contrast, TurboHash makes service of read requests fully lock-free.

3.3.2 Establishing the Search Path

A shard is allocated as a whole with all of its buckets in a contiguous space. Each bucket has a fixed number of slots. Each slot holds one KV item. For a linear probing scheme, in today's practice a new KV item can be placed into any empty slot within the probing scope. Slots can become available anywhere in the scope whenever their resident KV items are deleted. New KV items may be placed in any of these empty and other available slots in the scope. While such a placement without restriction is flexible and space

efficient, it often makes the search distance much longer. In the search for a non-existing key, which is the operation carried out before every new key insertion, the distance will always be the scope size. A search has to proceed until the search key is found or it reaches the boundary of the probing scope. The buckets do not contain information to establish a search path which can be (much) shorter than the scope size so that only necessary buckets are searched. As we have indicated, if we can keep new KV items in buckets as close as possible to their home bucket and establish a search path covering the buckets, a search does not have to walk beyond the path. In such a case, when the scope is set to a large size for high load factor, the cost of key search does not have to increase proportionally.

The solution appears at first sight to be a straightforward one, which is just to place a new KV item in the available slot on its linear search path and remember the path's last bucket. This is a valid idea. However, the difficult question is how to remember a path's last bucket. An intuitive approach would be to explicitly record this path-end information and update it whenever the path grows. But this approach leads to high time overhead and likely high space overhead. The possible places where the path-end can be recorded may be the home bucket, the currently end bucket on the path, or a separate data structure. In any of the places its updating overhead can be too high. When a new item is written into a new bucket and extends the path, the path-end must be accordingly updated (an ancillary write). These two writes are likely in two different Optane 256B access blocks, which are actually two block writes. Furthermore, the ancillary write must be completed before the KV write to guarantee following reads will reach the new item and the new item can be reached after a system crash. To this end, a fence/flush is required. If the path-end is recorded in the end bucket, two ancillary writes are required when the path grows (one in the new end bucket for indicating the new path-end and one in the old end bucket for invalidating it). Furthermore, the space overhead can be too high, as a bucket can simultaneously be the end bucket of multiple paths (up to the probing scope size). It would be too expensive for each bucket to pre-allocate such a large space for these possible paths. To address the issue, TurboHash doesn't record and update this path-end information at all. Its approach is motivated by three observations: (1) in most of the time a hash table has abundant empty bucket slots until a rehashing is to be triggered soon; (2) once a rehashing is carried out, many empty bucket slots are spread out in the table; and (3) even when a rehashing is near, the load factor is still less than 80-90% (see Figure 3.1a). Therefore, TurboHash uses a non-fully occupied bucket (with empty slots) to indicate a path end. To this end, it gives a slot whose data has been deleted a flag indicating that this slot is available for receiving a new KV item but isn't considered an empty slot. A path grows only when there are not empty slots in its current end bucket (and certainly not in any other buckets on the path). In this way, if a bucket with empty slot(s) are encountered during a search on a path, it is

guaranteed that the search key will not be found beyond this bucket and a continuous search is not necessary. Admittedly, this bucket may not be an accurate end bucket of the path (as it may not contain key(s) hashed to the path's home bucket). However, this is good enough to significantly reduce number of buckets involved in a search, as will be demonstrated in Section 3.4.

Another design issue is on the update operation. As we have indicated, for data crash consistency, an out-of-place write, instead of in-place overwrite, is required for an update operation. The challenges are similar to that with updating of the path-end information. First, there are two writes: one for writing the new KV item and an ancillary one for invalidating its old version. These two writes would be random accesses if without a careful arrangement. Second, the operation that makes the new version visible and the old version invisible has to be an atomic one for correctness if a lock-free read is allowed for high throughput. TurboHash's solution is to introduce near-place update, which limits the out-of-place write within the same bucket where the old-version KV item resides. As a bucket is of 256B and the metadata on (in)validating slots in a bucket are within a 8B atomic write unit, all the writes are in one the Optane access block with high efficiency.

3.3.3 A Bucket's Data Structure

As we have mentioned, for access efficiency each bucket is set as 256B long, the access unit of the Intel Optane Persistent memory. As shown in Figure 3.3, within a bucket there are 14 slots, each for storing a KV item with a 8B key and a 8B value. The 8B value can be a pointer to another space where the real value is stored should the value is larger than 8B. Besides the data, there are three types of metadata in a bucket. One is about slot status, including the valid bitmap and the delete bitmap. A bit in the 2-byte valid bitmap indicates whether the key in the corresponding slot is valid. A bit in the 2-byte delete bitmap indicates whether the KV item in the corresponding slot has been deleted. A slot's valid becomes 1 when a new KV item is written into it. When this item is deleted, its delete bit becomes 1. However, its valid bit remains as 1 as the key in the slot is still meaningful. A slot holding a deleted key isn't considered as an empty slot for the purpose of detecting a path end. Only a slot whose valid bit is 0 is defined as an *empty* one. However, the slot holding the deleted item is available to receive a new KV item by overwriting the deleted one. An insert operation always writes a new KV item in the first slot with a deleted item (if available) on its search path. After the overwriting, this slot's valid and delete bits become 1 and 0.

The second type of medadata is for concurrency control, including a 4-byte sequence number. This number is incremented by one whenever a write operation (delete, insert, or update) happens in the bucket. It facilitates lock-free reads.

The third type of metadata is for improving performance, including an array of tags. A tag is a 1-byte summary of a key by hashing the key in the corresponding slot. By grouping 14 summary keys in an array, TurboHash can use an SIMD instruction ("*_mm_cmpeq_epi8_mask*") for a quick preliminary search of all keys in a bucket.

As mentioned, TurboHash introduces the near-place update for high efficiency. To this end, it reserves an empty slot in each bucket for out-of-place updating of a KV item in the same bucket. That is, if a slot is the only (last) empty slot in a bucket, it doesn't accept a new KV item. This bucket is considered full for the purpose of detecting a path end.

3.3.4 Insert, Update, Delete, and Read

We describe the search operation before detailing how the four types of requests are served.

3.3.4.1 The Search Operation

Search for a given key is the most frequently used operation in a hash table. It not only is used to service a read request but also has to be employed before every insert/update/delete operation is performed at a bucket. The difference between read and the write requests is that the latter ones need to hold the write lock during the search. Like that in any linear-probing hash table, a search begins at the home bucket determined by the search key and the hash function, and continues on the sequential search path. At each bucket on the path, it needs to compare the search key with each of keys in the bucket. To speed up this process, TurboHash compares a 1-byte hashed value of the search key with each of the 14 tags in the bucket in parallel by using an Intel AVX SIMD instruction. Only the slots whose tags are matched and their valid bits are set and delete bits are not set will have their keys compared with the search key. If there is a match with the search key, the search key is found. If there isn't a match of the search key, the search will continue to the next bucket. It will terminate at a bucket where a matched key is found or at the path-end bucket, which is defined as the one with more than one slot whose valid bit is 0, or the one that has at least two empty slots (one of them is reserved for near-place update). The search operation is part of an insert/update/delete/read request service (see Algo. 1 and 4).

3.3.4.2 Insert

The thread servicing an insert request first acquires the write lock and proceeds with the aforementioned search operation for an empty slot or a valid slot that contains the search key within the search scope. Note that the empty slot does not include the one containing a deleted key. The search remembers position of the first deleted key it encounters. If the search arrives at a valid slot whose key matches the search key but has been deleted, this is a negative search, and the new key is inserted in the slot. If the search reaches the path end (a bucket with more than one empty slot with one of the them reserved for near-place updating), this is also a negative search. If a deleted key has been recorded on the path, the new key is inserted in its slot. Otherwise, it is inserted in one of the empty slots in the path-end bucket. To facilitate lock-free read, the insert thread takes two steps for the insert operation in a bucket. It first writes the key and value into the 16-byte slot's data segment as well as the corresponding tag (Algo. 2). It uses fence/flush to secure the data on the Pmem before moving to the second step, in which it makes the new data visible to reads. Specifically, it uses one 8-byte atomic write to set valid and delete bits to '1' and '0', respectively, and increments the sequence number by 1. Again it uses fence/flush to conclude the insert and then releases the write lock. If the search reaches end of the search scope and still cannot find an empty space or deleted space for an insertion, it performs shard rehashing and then inserts the key in the enlarged shard before releasing the write lock. If the search arrives at a valid slot holding the search key not yet deleted, this insert is actually an update request.

3.3.4.3 Update

As mentioned, TurboHash's update operation is a near-place update. Like the insert operation, an update within a bucket also takes two steps. Holding the WriteLock, the thread uses an empty slot in the bucket as the reserved slot to write the new KV item and updates the corresponding tag followed with a fence/flush. It then uses one atomic write to the slot's valid bitmap to turn the old version invalid and this new version visible, and increments the sequence number by one. With the atomicity, the valid bits for the old and new version slots can only be '10' or '01' at any moment during the operation. Without holding any lock, a read thread can see one and only one version at any moment. After this, the WriteLock is released (see Algo.2).

3.3.4.4 Delete

After a delete service thread identifies the slot storing the same valid key not yet deleted using the search operation with the WriteLock, it uses one atomic write to set the slot's delete bit and increments the sequence number by one. It then releases the lock. Note that it does not reset the slot's valid bit, and thus does not break any search paths. (see Algo.3).

3.3.4.5 Read

A read thread does not need any lock. It uses the search operation to look for the key. If it is not found within the probing scope, the read is completed declaring non-existence of the key. Otherwise, if a valid and non-deleted key is found in a slot, we cannot simply return the value in the slot. This is because without holding a lock the read value is likely modified right before the read and is thus a wrong one.

Within the slot there are two phases of read. One is carried out by the search operation, including reading the tags, valid/delete bits, and the keys for comparison. If the first phase finds the read key, the second phase is to read the corresponding value. These two phases of read are not atomic and cannot prevent other write threads from interfering. To eliminate a potential hazard, the read thread reads the sequence number before and after reading the value. Fence is placed between the two reads to ensure the ordering. It then compares the two numbers. If they are equal, the correct value has been read. Otherwise, the value may have been modified and could be wrong. And the read service is retried (see Algo. 1).

To explain why reading wrong values will not occur with the use of sequence number, we examine the timeline of the relevant read/write events. For the read thread there are four read events, which are R1 (read the tags), R2 (an atomic read of valid/delete bits and sequence number) for all slots with valid/non-deleted bits and matched tags, R3 (read the relevant keys and then value of the matched key), and finally R4 (read the sequence number again). We consider a write request (delete, insert, or update) is completed at its last atomic write to update bitmaps and increment the sequence number. If the two sequence numbers are equal, there must not be any write requests completed between R2 and R4 because any such a completion will cause the sequence number to be incremented. Furthermore, because service of write requests is serialized, there is at most one write request between R2 and R4. Otherwise, there must be a completion of a write request. Fortunately, if there is a write request between R2 and R4, it is guaranteed that it will not modify the bitmaps/sequence-number, or the read key and its value in the bucket. First, if

the write request is an insert, it must have matched with a key different from the read key. Otherwise, the valid/non-deleted status of the read key would keep it from inserting in the bucket. Second, if it is a delete or an update, both do not change the read key and value. Therefore, if the read thread confirms that the sequence number does not change, the read value must be correct. In addition, we show that a read thread can always find its target key and value if they are in the table. Obviously this is not an issue if the target key is not involved in any write requests during the read service. The target key is not available for the read if it is read after a delete or before an insert of the key. This is not an issue either. If the key is the target of an update during the read (between the R2 and R4 read events), the read thread can always read a value (assuming the key is not yet deleted). If the update completes before R4, a retry will be performed to read the new value. Otherwise, it will read the old value.

3.3.5 Shard Resizing & Failure Recovery

Shard resizing is performed by an insert thread when it cannot find an available slot within the probing scope. Then the shard, which is up to a few MB, is sequentially read to the DRAM, where an enlarged shard is built. The new shard is then sequentially written to the Pmem. During the period of time, read requests can still be served on the old version of the shard. When the new shard has been persistently written, the pointer to the shard in the shard's descriptor is updated (first in the Pmem descriptor then atomically updated in the DRAM). After this, all requests will be served on the new version. When the read threads on the the old shard complete their service, the space for the old version will be reclaimed when no other threads access it [29].

Since TurboHash does not change its directory structure and always commits a new shard with a 8-byte atomic update by modifying its version number in the Pmem and the 8-byte ShardMeta in the DRAM (shown in Figure 3.3), a system crash will not cause consistency issue for its directory. And as all writes are only available after committing the valid/delete bitmaps, partially updated key-value items are not visible to users. The only effort for a failure recovery is to read the directory in the Pmem and rebuild the in-DRAM directory.

3.4 Evaluation

In this section, we experimentally evaluate TurboHash by comparing it with several state-of-the-art hash tables for persistent memory, including CCEH [52], Dash [48], P-CLHT [44], and clevel hashing (CLEVEL) [15]. We implement TurboHash using persistent

allocator, Ralloc [9]. CCEH is a dynamic hash table for persistent memory. It supports resizing through segment splitting and directory resizing. Dash is also a dynamic hash table. It is similar to CCEH, but with several optimizations, including fingerprinting [58], and version based search. P-CLHT is a linked-list based persistent hash table, based on CLHT [18]. Each bucket of P-CLHT can store three key-value items. CLEVEL is an upgraded version of level hashing [84] by enabling asynchronous resizing.

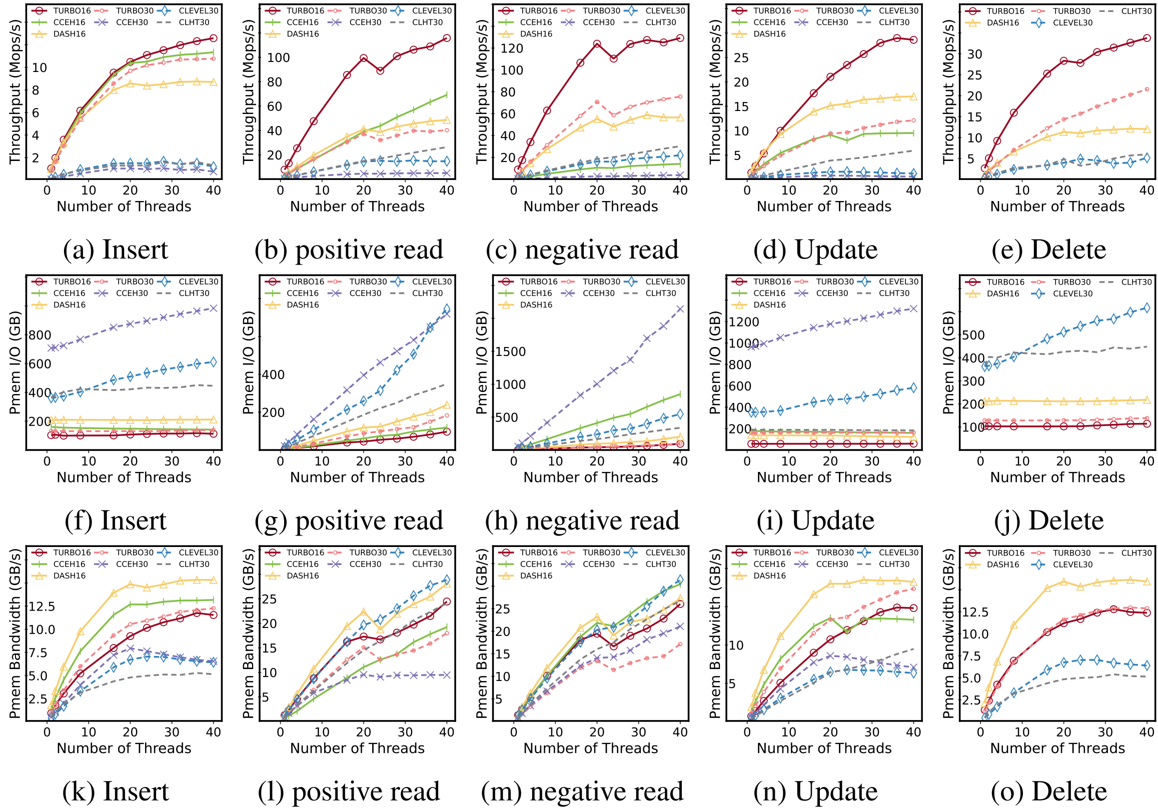
3.4.1 Experiment Setup

Table 3.1: Experimental Setup

Processor	Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz
Core	20 per socket
L3 Cache	55 MiB
Pmem	6 * 128GB Intel Optane DC
Mode	Interleaved App-Direct Mode
OS	CentOS Linux 8
Compiler	GNU 8.3.1, cmake 3.11.4

In our experiments, we use two different key-value sizes for comparison, 16 bytes (8B key and 8B value) and 30 bytes (15B key and 15B value). In the case of 30-byte KV items, real key and value are in a separately allocated space, and a 8B hashed key and a 8B pointer to the space are stored in the hash table.*CCEH16*[1] and *DASH16*[2] assume 16-byte KV item (8B key and 8B value). *CCEH30*, *CLEVEL30*, and *CLHT30* use 30-byte key-value items (15B key and 15B value). They are implemented using `libpmemobj` by authors of the clevel hashing paper (`git:#13ad3f2`) [3]. The TurboHash with 16B and 30B KV items are named *TURBO16* and *TURBO30*, respectively. All the threads in an experiment are pinned to one socket using `numactl`. All the hash tables are initialized with a capacity of 12 million KV items. In the experiments TurboHash is initialized with 64K shards (each has 16 buckets to have a total of 12 million capacity at the beginning), and uses a 16-bucket probing scope. CCEH16 adopts a 8-bucket scope (We use the copy-on-write version as it provides better read performance). CCEH30 uses a 4-bucket scope, as suggested in their code. CLEVEL30's scope is between 4-8, depending on its level count. CLHT30 conducts probing within a hash bucket, and buckets are organized on a linked list

[1]Source code of CCEH16: `https://github.com/DICL/CCEH`
[2]Source code of DASH16: `https://github.com/baotonglu/dash`
[3]`https://github.com/chenzhangyu/Clevel-Hashing`



(a) Insert (b) positive read (c) negative read (d) Update (e) Delete

(f) Insert (g) positive read (h) negative read (i) Update (j) Delete

(k) Insert (l) positive read (m) negative read (n) Update (o) Delete

Figure 3.4: Throughput and Pmem I/O volume with different hash-table requests.

for collision resolution. All the experiments are run on a server with an Intel Xeon Gold 6230 20-core processor, 64GB DRAM and 6 $\times$ 128GB Intel Optane DC.

3.4.2 Overall Performance

To evaluate the performance of the hash tables, we conduct extensive experiments, including insertions of new KV items (*Insert*), reading existing keys (*Positive Read*), reading non-existing keys (*Negative Read*), overwriting existing KV items (*Update*), and deleting all KV items (*Delete*). Experiment results are shown in Figure 3.4. In each experiment different number of threads (from 1 to 40 threads) are used. For *Insert*, *Update*, and *Delete*, each thread sends $120million/Number_of_threads$ requests. For *Positive Read* and *Negative Read*, each thread sends 10 million requests. Figure 3.4 reports throughput of the hash tables (number of requests serviced per second) and the corresponding raw Pmem I/O volume. This I/O volume represents all read/write data amount on the Optane Pmem's media, including amplified I/O due to existence of its 256B access unit. It is measured with `ipmwatch`, available in the Intel VTune Amplifie tool.

Insert. TurboHash conducts about four rehashing within a shard during Insertion. Other hash tables in comparison also conduct four rehashing. All the hash tables have a load factor of around 60%. For small key-value size (8B key and 8B value), CCEH16, DASH16, and TURBO16 store the data in the hash table slots. As shown in Figure 3.4a, with fewer than 20 threads TURBO16 and CCEH16 have similar *Insert* performance. When the number of thread is more than 20, TURBO16's throughput is 10% higher than CCEH16. The performance gap between TURBO16 with CCEH16/DASH16 is due to use of lock: TURBO16 uses a DRAM spinlock while CCEH16 uses an in-Pmem writer lock on its hash table segment and DASH16 uses Pmem lock on its bucket. The lock on the Pmem causes more Pmem write I/O. The write locks in the Pmem are frequently updated. Each update is a small write on a 256B Optane block. Optane has a small write buffer to exploit write locality [77]. However, with high concurrency (of many threads), the working set for the lock updates is larger than the buffer, causing frequent buffer evictions and write amplification. Figure 3.4f shows that CCEH16 causes 18% more Pmem I/O and DASH16 has 60% more Pmem I/O, which are mainly caused by more Pmem writes due to the Pmem lock. For 30-byte KV size, TURBO30's throughput is about 6$\times$ higher than the others. For CCEH30, we see a much larger amount of Pmem I/O (10$\times$ more than TURBO30's) (see Figure 3.4f). The main reason is that in order to support the atomic update operation for data size larger than 8 bytes, CCEH30 uses `libpmemobj`'s transaction feature for out-of-place writes. It uses undo logging for application object updates, and introduces large write amplification [80]. Because of the limited Pmem bandwidth, this much increased I/O leads to CCEH30's large write throughput degradation. As CLHT30 uses a linked list to organize its buckets, searching on the list causes random small access on the Pmem. As we indicated in Figure 3.2, random access to the Pmem can be 4X slower than sequential writes. Clevel hashing has a bottom-to-top searching strategy in its multi-level structure. It has to search all the possible buckets to ensure the key does not exist before any insertion. Searching the randomly located buckets introduces large read amplification, causing its Pmem I/O volume 5$\times$ more than TURBO30, as shown in Figure 3.4f.

Positive Read. We see significant performance improvement for TURBO16 over CCEH16. The source code of CCEH16 implements double hashing for reads, which aims to increase load factor at the cost of larger read amplification. As we can see in Figure 3.4g, CCEH16 has more I/O than TURBO16. TURBO16 also has better read performance than DASH16. This is due to DASH16's higher I/O volume because of the stash search. For 30-byte key-value items, TURBO30 has 140% to 600% throughput improvement over the others. The major reason is that TurboHash uses sequential reads as much as possible to minimize read amplification and thus has the least I/O volume, as shown in Figure 3.4g.

Negative Read. TurboHash has the best negative-read performance among the schemes. A major reason is that it reads much less amount of data during the key search (see Figure 3.4h). TurboHash reads only KV items before the end of a search path. DASH16 has to search all the target buckets and the stash buckets. CCEH also needs to search its entire scope. CCEH30 reads more data than CCEH16 because CCEH30 doubles the bucket size to accommodate large key and value, causing more data to be read within a probing scope. CLEVEL30 and CLHT30's search is on a path consisting of non-contiguous memory locations which comprise the performance. With reduced search paths and sequential Pmem access, the negative read performance of TurboHash is much higher than the others, shown in Figure 3.4c.

Update. In the *Update* experiment all existing key-value items are updated. *Update* is a special case of *Insert*. Compared to *Insert* throughput, TURBO16's *Update* throughput improves the most (almost 2 × as high as its *Insert* throughput). There are two reasons. One is that *Update*'s search path is shorter than *Insert*'s. The other is that *Update* does not cause shard rehashing. TURBO30 also achieves a higher throughput, though with a smaller improvement. Frequent random access to the KV items outside of the slots in TURBO30 makes the Pmem and CPU resources more constrained, leaving less room for improvement.

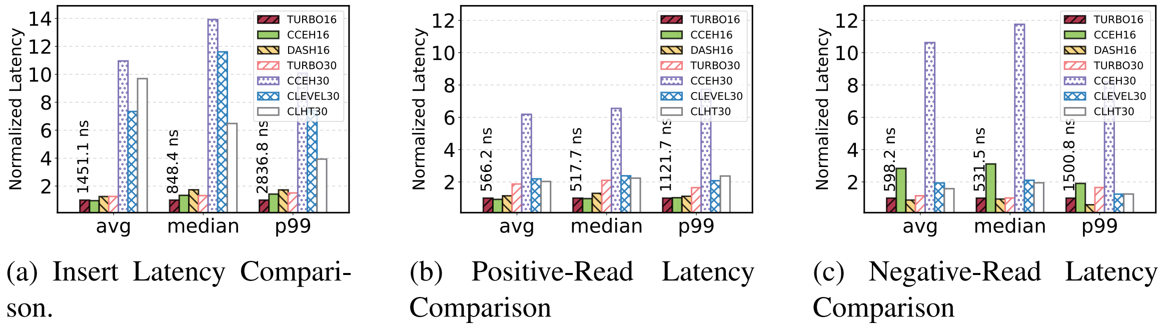
Delete. TurboHash always achieves the highest throughput because it has the lowest I/O volume due to its shorter and sequential search paths (*Delete* not implemented in CCEH).

3.4.3 Latency Comparison

In this section, we evaluate the read/write latency of all hash tables. We use 16 threads to write 120 million key-value items. Each thread sends 10 million read requests.

As shown in Figure 3.5, TurboHash almost always has the lowest latency among the hash tables in all the test workloads (*Positive Read*, *Negative Read*, and *Insert*). By applying the linear probing and using only necessary probing length, TurboHash achieves the lowest I/O volume. It also avoids random access to the Pmem. As shown in Figure 3.6, CCEH16 has 4× I/O volume as much as that of TURBO16 for *Negative Read*, because it reads more buckets than necessary to find an non-existing key. The extra I/O causes the higher latency.

In most of the experiments, TURBO16 outperforms DASH16, except the P99 latency for *Negative Read*. It is because with the large linear probing scope (16 buckets), TURBO16 may have to search for a long distance to reach some KV items placed at the other end of the scope, causing longer tail latency. This issue can be alleviated by reducing the probe scope. CCEH30 has 3×, 5×, and 5× more I/O volume than TURBO30 for *Posi-*



(a) Insert Latency Comparison.

(b) Positive-Read Latency Comparison

(c) Negative-Read Latency Comparison

Figure 3.5: Latency comparison between the hash tables (with 120 million insertions and 160 million reads using 16 threads)

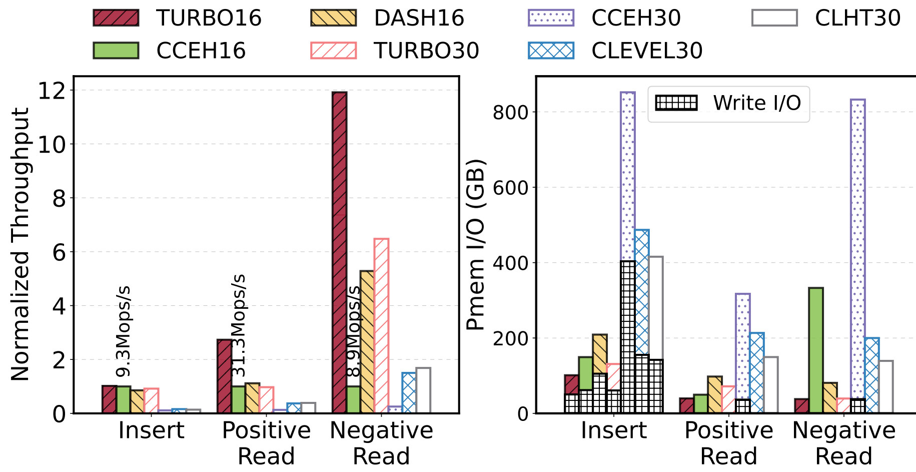


Figure 3.6: Throughput and raw Pmem I/O volume (with 120 million Inserts and 160 million Reads using 16 threads). Grey bars in (b) indicate the portion of write volume.

tive Read, *Negative Read*, and *Insert*, respectively. In addition to large read volume and random access, CCEH30 amplifies writes volume in its support of 15-byte atomic writes with undo logging. Thus, CCEH30 has the largest I/O traffic and the worst latency for 30-byte key-value items. CLEVEL30 has 5× higher average *Insert* latency than TURBO30. There are several reasons. First, it searches from the bottom to the top levels of its multi-level hash table before an insertion can be carried out. That is, a fixed overhead is added to any *Insert*. Second, its bucket size (64 byte) does not match the Pmem access block size (256 bytes). Therefore, each read to a bucket introduces 4× read amplification. As we can see in the left graph of Figure 3.6, CLEVEL30 has 3× more I/O volume than TURBO30. Third, as buckets on the search path are non-contiguous in the Pmem, CLEVEL30 introduces more random access on the Pmem, which compromises latency.

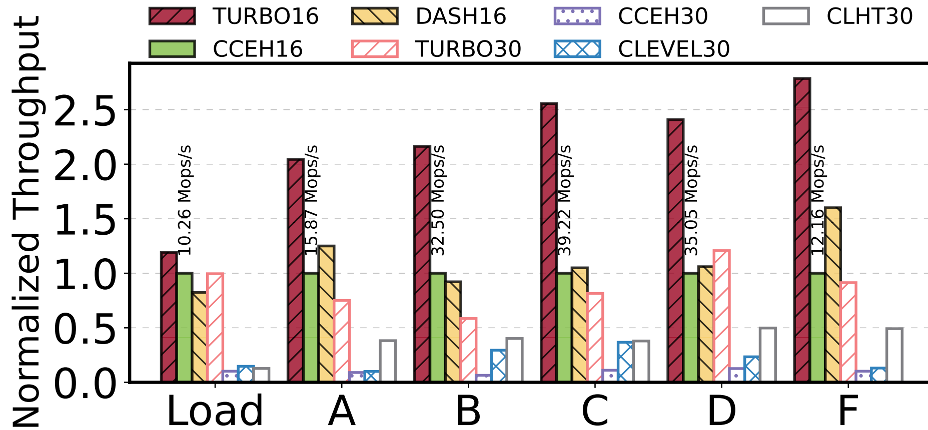


Figure 3.7: Throughput normalized to CCEH16. Load: 100% write. A: 50% write, 50% read, B: 5% write, 95% read, C: 100% read, D: 5% write, 95% read the latest, F: 50% read-modify-write, 50% read.

As a consequence, both the write latency and throughput are compromised, as shown in Figure 3.5a and the left graph of Figure 3.6. For average *Positive Read* latency, TURBO30 is about 15% lower than CLEVEL30. The improvement is not significant even though Clevel hashing has more I/O and random access. This is because current load factor for both hash tables is around 50%, which means that most of keys can be found at their home buckets in the search paths. The performance gap is more noticeable for *Negative Read*. We see 2× lower average read latency for TURBO30 compared with CLEVEL30, which has to search all the buckets (around 4 to 8) at random locations. CCEH30 has the highest read latency. As shown in the right graph of Figure 3.6, CCEH30 generates some write volume at the Pmem during service of read requests. It places a read lock in the Pmem and frequently updates the lock, which degrades its latency. CLHT30's latency is similar to that of CLEVEL30. For *Insert*, it also reads all the buckets at random locations along its search path (a linked list) before any insertion. So the I/O amplification and write latency are both high. As a consequence, the write throughput becomes lower, which is only 17% of TURBO30, as shown in the left graph of Figure 3.6.

3.4.4 Results of YCSB Benchmarks

The Yahoo! Cloud Serving Benchmark (YCSB) [16] is a popular benchmark used to evaluate performance of NoSQL databases. We write a test bench for all the hash tables to support YCSB benchmarks, which generate uniform workloads of different access behaviors ("Load", "A",..."F") (see Fig 3.7). The results for skewed workloads are similar to

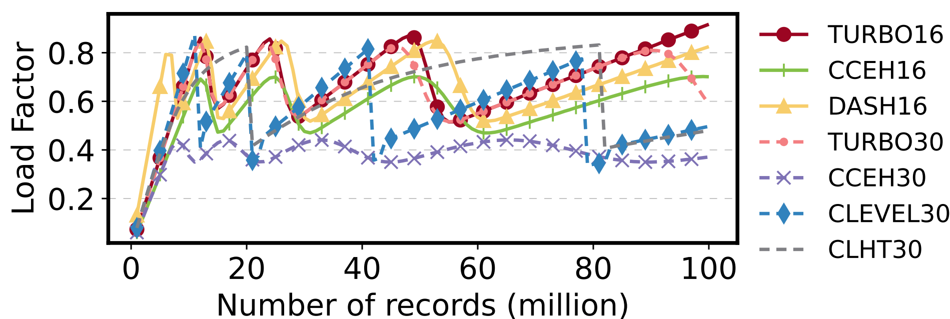


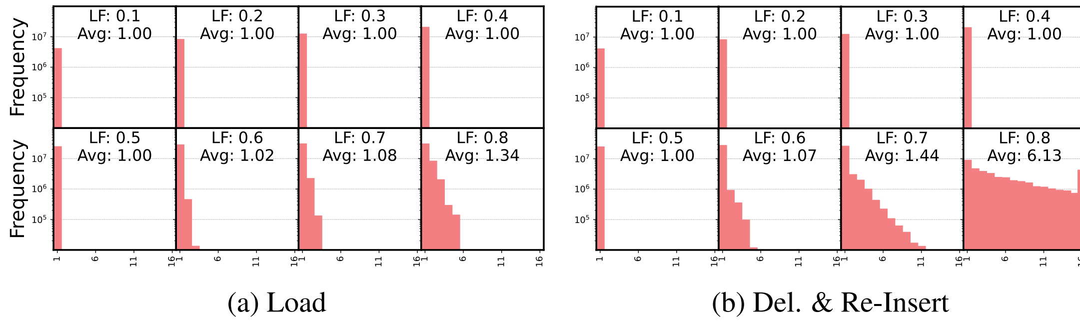
Figure 3.8: Load factor after every one million inserts. Each hash table is initialized with a capacity of 12 million items.

that for uniform workloads and we omit the results due to space limits. We run 20 threads and send 10 million requests in each workload. For the all-write workload ("*Load*") that loads 120 million items to the hash table, as expected TurboHash (either TURBO16 or TURBO30) has a higher throughput than others, as shown in Fig 3.7. Each of the other workloads starts after the "*Load*" and consists of only or mostly read requests. For all the workloads, TurboHash has up to 2.6× higher throughput than CCEH. CCEH30 has the lowest throughput, as it always requires a bucket lock for reads. The extra write due to lock acquisition during a search compromises its performance. TURBO30 has 1.5×-4× higher throughput over CLHT30 and CLEVEL30. This is because TURBO30 has much less I/O than the others due to its efficient search strategy.

3.4.5 Load Factor

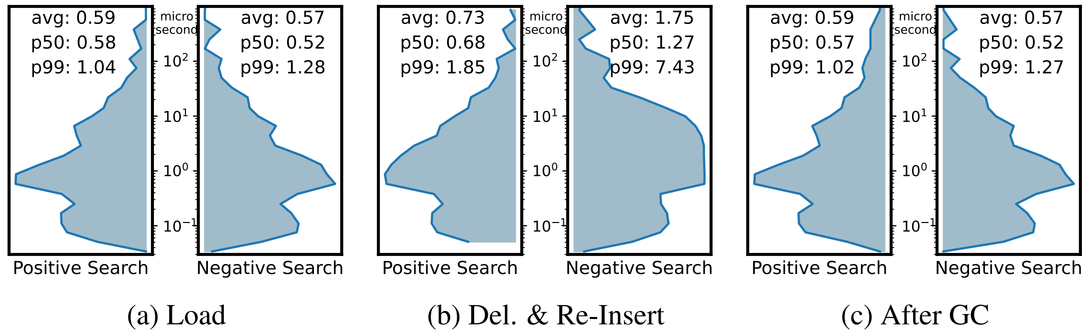
Load factor is a critical metric measuring a hash table's space efficiency. One may trade space efficiency for access performance of a hash table. TurboHash achieves both space efficiency and high performance with short search path and sequential Pmem access. In this section we compare load factor variations during insertion of 100 million key-value items in the hash tables. The results are shown in Figure 3.8.

The maximum load factor of CCEH30 is less than 44% because it only probes at most 4 buckets before a hash collision occurs. CCEH16 has a higher peak load factor (70%) because the implementation optimized by its inventors enables double hashing inside the segments. Though both DASH and TurboHash achieve the highest peak load factor (85%), TurboHash has a much higher throughput (see Figure 3.4). Though CLEVEL30 and CLHT30 can achieve similarly high peak load factor (80%), they allow more flexible placement of KV items for collision resolution, which leads to probing at random Pmem locations and compromises access performance. TURBO30 always searches buckets se-



(a) Load (b) Del. & Re-Insert

Figure 3.9: Probe distance histogram and average probe distance (Avg) for non-existing keys under different load factor (LF). (a) are the histograms after loading new keys to reach various load factors. (b) shows the results after we delete all the keys in each hash table and then insert another set of keys to reach the same load factor. The probing scope is 16 buckets.



(a) Load (b) Del. & Re-Insert (c) After GC

Figure 3.10: Read latency histograms with a 0.8 load factor. (a) Loading new keys into empty hash tables. b) The results after we delete all the keys and then load a new set of keys to the same load factor. (c) The results after a garbage collection.

quentially for strong spatial locality, which produces up to 8× higher performance in terms of both throughput and latency (see Figure 3.5).

3.4.6 Probing Distance

To evaluate the probing efficiency in terms of probing distance, we measure the distance with the searching of non-existing keys in TurboHash under different load factors. As shown in Figure 3.9, though the probing distance increases with the load factor, its impact on the distance is limited. First, the increase even with a large load factor is still small. For example, with the 0.8 load factor, the longest distance is only 6 (much shorter than the probing scope size, which is 16) and the average distance is 1.34. Second, long distances

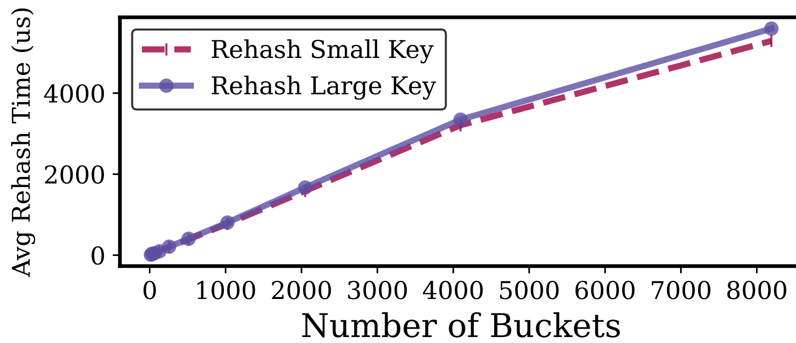


Figure 3.11: Rehashing time with different shard size.

hold only a small fraction of all of the distances in a histogram (note that the graph is of a logarithmic scale).

As slots with deleted keys do not break a search path, they may unintentionally make the path longer and compromise probing efficiency. To observe the effect, for each of the hash table shown in Figure 3.9(a), we delete all of its keys and then insert another set of keys to the same load factor level. We then redo the experiments to obtain a new set of histogram graphs (see Figure 3.9(b)) Now with a high load factor, such as 80%, the average probe distance for non-existing keys increases to 6.13, which causes the average negative-read-latency increases to 1.75μs (from 0.57μs). However, this experiment represents a rare scenario. And an easy remedy exists. When it is determined that there are too many deleted keys in a shard, TurboHash may conduct a garbage collection operation, which is a special shard rehashing, to remove the deleted keys. The only difference of this special shard rehashing from a regular one is that it will keep the new shard of the same size. Figure 3.10 shows the latency histograms before and after a delete-and-reinsert, as well as after a garbage collection. It shows that while a delete-and-reinsert operation can increase the latency, especially for the long-tail latency of negative search, its good performance can be recovered with a garbage collection.

3.4.7 Shard Rehashing

As shown in Figure 3.11, the rehashing time is proportional to the shard size. Each rehashing doubles number of buckets in the shard, and each key will be relocated to a bucket in the new shard. Because the shard size is doubled, TurboHash can simply double the index of a bucket in the old shard (a bucket array) to obtain the index of a bucket in the new shard where all of its keys are relocated. Even if a key is larger than 8 bytes and the full key is not in the bucket, the relocation doesn't involve additional Pmem access because the

hashed keys are stored in the table. For a shard of 8192 buckets, the average rehashing time is 5-6 μs. While the rehashing operation and its impact on the tail latency can hardly be avoided in any hash table design, TurboHash has its unique design that helps to bound its impact. TurboHash assumes a large number of shards in its design by over-provisioning. For a hash table of 128K shards and each shard initially of 16 buckets, it can store 10 billions 16-byte KV items when a shard is resized to 8192 buckets. At this time the hash table occupies 256GB Pmem and 1 MB DRAM for a directory of 128K shards. The 1MB directory can be effectively buffered in the LLC cache for high performance. For hosting a large KV store, the directory can be comfortably set at a larger size to accommodate more relatively smaller shards without concern on its DRAM demand.

3.5 Related Work

TurboHash is a variant of dynamic hashing. It resizes the table as needed and uses fine-grained resizing/rehashing (per shard). The dynamic hash table has been used in many real systems [60, 66]. And our focus is on the design of a high-performance dynamic hashing for persistent memory.

3.5.1 Hash Table for Persistent Memory

As commercial persistent memory has been available in recent years, a well-designed hash table for persistent memory is on demand to leverage Pmem to provide high performance service. There have been some recent works on designing hash tables for Pmem, such as PFHT [22], level hashing [84], clevel hashing [15], CCEH [52], and P-CLHT [44]. PFHT is a variant of cuckoo hashing for Pmem that avoids cascading writes by allowing only one cuckoo displacement. It uses a stash to prevent full resizing and improves the load factor. Level hashing has two hash tables, making it a two-level structure (top level and bottom level). While the bottom level is half of the top level, each resizing only moves around one third of the data. Clevel hashing is a variant of level hashing. It implements asynchronous resizing, so the write tail latency can be improved. CCEH is an extendible hash table which expands the capacity by segment split and directory doubling. P-CLHT is a linked list based persistent hash table. Each bucket of P-CLHT is of 64 bytes and can store three key-value items. While all those hash table have made significant efforts to improve access efficiency, none of them considers the 256-byte access granularity of existing Pmem (Intel Optane DC). TurboHash is designed for sequential probing on reduced number of buckets on the Pmem. Its access efficiency is further improved by avoiding

across-shard resizing, which is possible for a hash table dedicated for a KV store whose scale is often known in advance.

3.5.2 Hash Table Concurrency Control

CCEH [52] uses reader/writer lock in its directory to support concurrency control. Level hashing [84] adopts a fine-grained locking for each slot, and each insertion may lock up to two slots. Clevel hashing [15] makes use of Compare-and-Swap (CAS) primitive to support concurrent writes while read is lock-free. P-CLHT [44] employs bucket lock for writes and the read is lock-less. All of these hash tables, except Clevel hashing, have a globally-locked rehashing scheme (directory resizing for CCEH), which may disrupt the service and cause long tail latency. Clevel hashing proposes a dynamic multi-level hash structure to asynchronously resize the table at the cost of searching more levels (bottom-to-top searching), which compromises read performance as more buckets at random locations have to be searched. In TurboHash, we only have a write lock at a shard, which caps the tail latency while maintains high read/write performance. The support of lock-free access in existing hash tables, such as clevel hashing and P-CLHT, will not be available when a key or a value is in-place updated in a non-atomic manner (certainly with a lock). TurboHash does not have this limit by using a sequence-number-based approach.

3.6 Summary

We introduce TurboHash, a persistent hash table designed for high-performance key-value stores in this paper. By enabling out-of-place update at a cost equivalent to that for an in-place write, conducting probing on a path sequentially and only for a necessary length, and utilizing Intel Optane DC's hardware feature, TurboHash minimizes the Pmem I/O traffic and achieves $2\times$ to $8\times$ improvement of access performance over state-of-the-art Pmem hash table designs in terms of both throughput and latency.

3.7 Appendix

Algorithm 1: Lock Free Search

```
1  Function Search(key, callback):
2      [slot, isfind] = FindSlot(key)
3      if isfind then
4          callback (slot)
5          return TRUE
6      return FALSE
7  Function FindSlot(key):
8      key_hash = Hash(key)
9      shard = LocateShard(key)
10     bucket = LocateBucket(shard, key_hash)
11     hash_tag = ExtractTag(key_hash)
12     probe_distance = 0
13     while probe_distance < MAX_DISTANCE do
14         seq_no = bucket.meta.seq_no
15         match_pos = bucket.SIMDmatch(hash_tag)
16         foreach pos ∈ match_pos do
17             slot = buckets.slots[pos]
18             if key == slot.key then
19                 if bucket.NeedRetry(seq_no) then
20                     bucket = LocateBucket(bucket.id)
21                     go to 14 // Retry in current bucket
22                 return {slot, TRUE}
23         if bucket.ReachSearchEnd() then
24             return {"", FALSE}
25         bucket = bucket.Next()
26         probe_distance++
       /* Reach the maximum probe distance.  */
27     return {"", FALSE}
```

Algorithm 2: Insert and Update in TurboHash

```
1  Function Insert (key, value):
2      shard = LocateShard (key)
3      WriteLockGuard guard(shard)
4      is_find, bucket_id, slot_id, old_slot_id = FindSlotForInsert(key)
5      if is_find == TRUE then
6          bucket = LocateBucket(bucket_id)
7          bucket.Insert(key, value, slot_id, old_slot_id)
8          return TRUE
9      else
10         shard.Rehash()
11         go to 2
12 Function Bucket::Insert (key, val, slot_id, old_slot_id):
13     slot = slots[slot_i]; tags[slot_i] = key.tag
14     slot.key = key; slot.val = val
15     CLWB + SFENCE
16     new_meta = this->meta;
17     new_meta.valid = new_meta.valid | ( 1 << slot_i )
18     if old_slot_i ≠ -1 then
           // Flip the old slot bit in the valid bitmap.
19         new_meta.valid ⊕ = ( 1 << old_slot_i )
           // Epoch based reclamation.
20         epoch.markForDeletion(slots[old_slot_i])
21     new_meta.delete & = ∼( 1 << slot_i )
22     new_meta.seq_no++
23     this->meta = new_meta
24     CLWB + SFENCE
```

Algorithm 3: Delete

```
1  Function Delete(key):
2      shard = LocateShard (key)
3      WriteLockGuard guard(shard)
4      bucket = LocateBucket(shard, key_hash)
5      probe_distance = 0
6      while probe_distance < MAX_DISTANCE do
7          match_pos = bucket.SIMDmatch(hash_tag)
8          foreach pos ∈ match_pos do
9              slot = buckets.slots[pos]
10             if key == slot.key then
11                 bucket.Delete(pos)
                   /* Epoch based reclamation */
12                 epoch.markForDeletion(slot)
13                 return TRUE
14         if bucket.ReachSearchEnd() then
15             return FALSE
16         bucket = bucket.Next()
17         probe_distance++
       /* Reach the maximum probe distance. */
18     return FALSE
19 Function Bucket::Delete(slot_id):
20     new_meta = this->meta;
21     new_meta.delete |= 1 << slot_id
22     new_meta.seq_no++
23     this->meta = new_meta
24     CLWB + SFENCE // Persist the 8-byte bucket meta.
```

Algorithm 4: Ancillary Functions

```
1  Function FindSlotForInsert(key):
2      key_hash = Hash(key)
3      shard = LocateShard(key)
4      bucket = LocateBucket(shard, key_hash)
5      bucket_id = -1
6      slot_id = -1
7      hash_tag = ExtractTag(key_hash)
8      probe_distance = 0
9      while probe_distance < MAX_DISTANCE do
10         match_pos = bucket.SIMDmatch(hash_tag)
11         foreach pos ∈ match_pos do
12             slot_key = bucket.slots[pos].key
13             if key == slot_key then
14                 slot_id = bucket.PickEmptySlot()
15                 return {TRUE, bucket.id, slot_id, pos}
16         if slot_id == -1 And bucket.meta.delete ≠ 0 then
17             bucket_id = bucket.id
18             slot_id = bucket.PickDeleteSlot()
19         if bucket.ReachSearchEnd() then
20             if slot_id == -1 then
21                 bucket_id = bucket.id
22                 slot_id = bucket.PickEmptySlot()
23             return {TRUE, bucket_id, slot_id, -1}
24         bucket = bucket.Next()
25         probe_distance++
26     if slot_id ≠ -1 then
           // Find deleted slot on the searching path
27         return {TRUE, bucket_id, slot_id, -1}
28     return {False, -1, -1, -1}
29 Function Bucket::SIMDmatch(hash_tag):
30     hash_vec = _mm_set1_epi8 (hash_tag)
31     res = _mm_cmpeq_epi8_mask (hash_vec, tags);
32     return res & valid & (∼ delete)
33 Function Bucket::NeedRetry(old_seq_no):
34     if old_seq_no ≠ meta.seq_no then
35         return TRUE
36     return FALSE
37 Function Bucket::PickEmptySlot():
       /* choose from empty slot */
38     return __builtin_ctz(∼ valid )
39 Function Bucket::PickDeleteSlot():
       /* choose from deleted slot */
40     return __builtin_ctz(delete)
41 Function Bucket::CanInsert():
42     return delete != 0 or ReachSearchEnd()
43 Function Bucket::ReachSearchEnd():
44     return __builtin_popcount(valid) < 13
```

CHAPTER 4

Spot-On: Optimizing Use of DRAM to Improve Performance of Index Structures on Intel Optane DC Persistent Memory

Intel Optane DC Persistent Memory (PMEM) is the first commercially available and rapidly deployed persistent memory. Compared to the DRAM, PMEM's larger capacity and cheaper price makes it a lucrative platform to persist data with its unrivaled high throughput, low latency properties. This motivated a large number of efforts to incorporate conventionally fast data management approaches like hash-table, B+ tree and other index structures into PMEM to better utilize its performance and persistence characteristics. However, it is observed that PMEM despite being byte-addressable, behaves like a block device, having an access unit of 256 bytes (XPLine). Consequently, as the nature of updates in these index structures is inherently smaller than PMEM's XPLine, PMEM is observed to incur huge performance penalty due to high write amplification. A similar cost can also be seen for index reads. While the idea of translating DRAM as a read cache and/or write-back buffer seems to be an effortless solution to close the performance gap, existing index designs fail to provide a holistic solution that manages both read and write amplifications while simultaneously proving persistence, crash consistency, and quick recovery after a crash.

In this work we propose a framework "Spot-on" that enhances persistent memory index structures using application-managed caching and buffering. This framework is built upon observations from applying a DRAM layer upon some representative persistent-memory index structures, including a B+ tree, a skip list, and a hash table. Using spot-on framework we develop a B+-Tree, `SPTree`, a novel index structure that facilitates applications to selectively cache read-intensive parts of an index and buffer writes to index structure, while providing crash consistency and quick recovery upon crash. Compared to the state-of-art indexes, SPTree provides higher write and read throughput up-to 2X and 4X respectively.

4.1 Introduction

The memory/storage hierarchy, which consists of multiple levels including CPU cache, DRAM, and block devices such as SSDs and HDDs, has been stable for decades.

Accordingly, the principal management designs for data across its levels, such as set-associative CPU caches, page-based virtual memory and block-based read cache and write-back buffer, are well established by carefully considering individual devices' performance characteristics to maximize the hierarchy's performance. However, with emergence of Intel Optane DC persistent memory (Optane PMEM for short), the first commercially available persistent byte-addressable memory, a new level is introduced into the hierarchy. We contend that it is necessary for the DRAM to serve as a cache level for the PMEM to boost its effective performance.

4.1.1 DRAM as a Cache of PMEM

Like DRAM, the Optane PMEM is a byte-addressable memory device that can be directly accessed via load and store instructions. However, its performance gap with the DRAM is still significant (around 2.5X-3X worse than that of DRAM in terms of its latency and throughput). In the meantime, the PMEM can have a 5-10x increase of per-module capacity over the DDR4 DRAM while its per-GB price is 2X-5X less expensive than DRAM. Therefore, placing the PMEM underneath the DRAM in the hierarchy has the potential of taking advantage of both DRAM's high performance and PMEM's large capacity and low price. Indeed, the Optane PMEM has a memory mode in which DRAM acts a cache for data accessed on the PMEM. Though little is known on how Intel CPU's IMC (Integrated Memory Controller) enables this transparent caching, a constraint in the use of the mode highlights the challenge in the management of this memory level. The constraint is that the persistent memory has to be treated as a volatile memory, or data in the memory cannot survive a system restart. The capability of retaining data on the PMEM is one of its major features attractive to many potential users.

4.1.2 DRAM as a Write Buffer of PMEM

While DRAM is used as PMEM's cache, it not only should be used as a read cache, but also must be used as a write buffer to enable the write-back policy for three reasons. First, DRAM is faster than PMEM by around 2-3X for reads and is around 4-6X for writes in terms of bandwidth. Second, recent studies have shown that Optane PMEM has an access unit of 256 bytes to the memory's media. Any write smaller than the size leads to a write amplification and reduction of effective throughput. For example, with 64-byte random writes the PMEM's throughput is reduced to about 1/4 of its peak one [81]. Third, to ensure crash consistency for written data, an application may have to frequently use expensive fence and flush instructions between writes to the PMEM, which may significantly degrade write efficiency. DRAM can used as a write buffer to coalesce writes and flushing buffer to

PMEM. In doing so random writes to PMEM are transformed into one big sequential write that aligns with PMEM's access granularity so as to extract high throughput from PMEM. It is tempting to use large amounts of DRAM to fix shortcomings of PMEM, however it is difficult to retain PMEM's persistence feature with a DRAM as its cache.

Some fundamental challenges exist due to unique characteristics of the DRAM-PMEM layers. Unlike CPU-caches/DRAM layers, the DRAM as a cache for PMEM presents unique challenges. This is primarily due to DRAM being much larger than the CPU cache. While the CPU cache can be battery/capacitor protected to keep dirty data in it from being lost upon a power failure, the DRAM cache is unlikely to have such a support. This presents a dilemma about the usage of DRAM as a cache for PMEM as using DRAM to enhance performance naturally leads to using large amounts of DRAM while simultaneously presenting a risk of losing data due to DRAM's volatile nature.

To address these issues, we propose a framework, Spot-on, for designing DRAM-PMEM hybrid indexes to provide low latency, high throughput and persistency features on the persistent memory (Intel Optane NVDIMM). The framework provides guidelines which can be summarized as **buffered, out-of-place merging then caching(BOC)**. **B:** For write-intensive workload, we can apply write buffer to batch small writes and flush them together to PMEM to reduce write amplification. **O:** During batch merging, we should always use out-of-place merging if possible to reduce usage of flush&fence to improve the efficiency. **C:** For indexes facing pointer chasing problem, we should also cache the frequent accessed parts in DRAM to reduce the overhead of random access in PMEM. Meanwhile, based on these guidelines, we design a persistent B+-Tree, Spot-on Tree (SP-Tree), which leverages the DRAM for caching and buffering. In SPTree, the internal nodes are stored in DRAM while the updates to the internal nodes are propagated to a persistent B+-Tree asynchronously for crash consistency. Existing keys in SPTree are also cached in bloom filters to avoid access on PMEM for negative reads of non-existing keys. Write buffers can also be configured in SPTree to reduce write amplification.

4.2 Background

In this section, we describe the background of non-volatile memory and some existing persistent index designs.

4.2.1 Non-Volatile Memory

Intel Optane Persistent memory is the first commercially available non-volatile memory (NVDIMM). It can be configured in two different modes. The first one is called

`Memory Mode`, in which the CPU regards the Optane as a larger main memory and uses the DRAM as its cache. In this mode, the Optane NVDIMM does not provide persistency. The second mode is `App-Direct Mode`. In this mode, the Optane NVDIMM works as a persistent device. A file system which supports Direct Access (DAX) provides direct access to the persistent memory, and bypasses the file system block I/O.

Though Optane NVDIMM provides the processor with cacheline (64-byte) access granularity, the physical media access granularity is 256 bytes (XPLine) [77, 74, 81]. Any non-contiguous writes of data smaller than XPLine size requires a read-modify-write operation, leading to write amplification and reduced effective memory bandwidth. To reduce write amplification, Optane NVDIMM employs a write-combining buffer to merge adjacent small writes.

4.2.2 Persistent Indexes

There are mainly two kinds of persistent indexes. The first one is persistent hash table, such CCEH[52], Level hashing[84] and Dash[48]. And the the second is persistent range indexes, including FastFair[34], FP-Tree[58].

Persistent Hash Table. Existing works on persistent hashing mostly focus on building a hash index by directly updating the index in-place on persistent memory, such as PFHT [23], level hashing [84], and CCEH [52]. PFHT is a cuckoo hashing variant that is optimized to reduce memory writes during serving write requests by allowing at most one displacement in a write. Level hashing applies a two-level hash scheme so that each key can have three buckets as the candidates for insertion, which helps improve the load factor. Instead of double hashing, CCEH is an extendable hashing that uses a linear probing strategy so that a successful insertion only requires one memory write. All of these works try to reduce persistent memory writes during insertion in order to design a write-optimized hashing. However, for the existing persistent-memory device, Optane PMEM, the in-place update designs still cause large write amplifications as Optane PMEM's physical media access granularity is 256B [77].

Persistent Range Indexes. There have been some works on designing b-tree for persistent memory. FastFair[34] is lock-free read B+-tree which avoid expensive copy-on-write and logging to tolerate transient inconsistency. BzTree[3] relies on Persistent Multi-word Compare-And-Swap(PMwCAS) primitive to implement a lock-free tree. FP-Tree[58] store inner nodes of the tree in DRAM to achieve high performance. However, it has to scan all nodes on PMEM to reconstruct the inner nodes after reboot or crash. PACTree[39] employs persistent trie index as the internal nodes and asynchronously update the internal nodes using a structural-modification-operation (SMO) log.

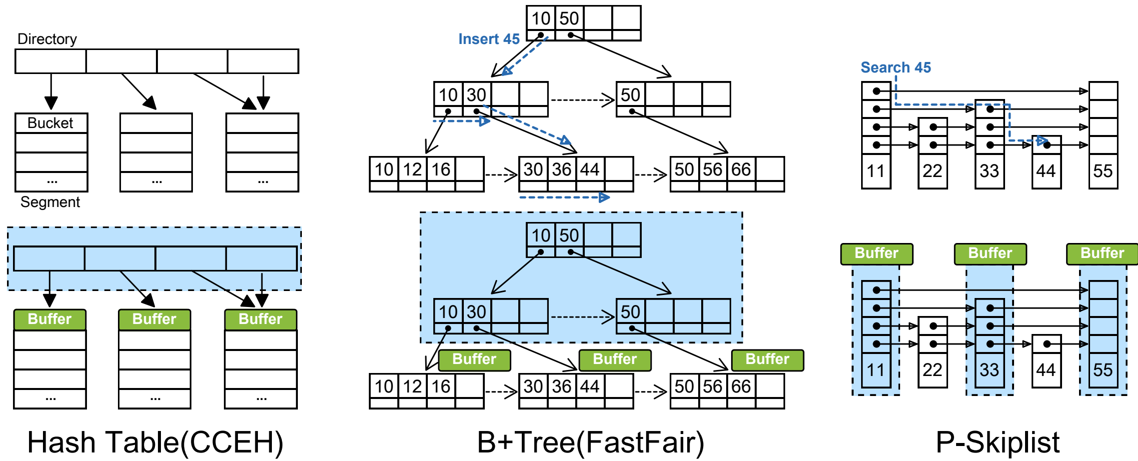


Figure 4.1: Three Data Structure

4.3 Case Study

In this section, we analyze three persistent index data structures and propose guidelines for designing hybrid DRAM-PMEM data structures. We extensively study three data structure: CCEH[52] (persistent hash table), FastFair [34] (persistent b-tree) and P-Skiplist (our version of persistent skiplist) [61], as shown in Figure 4.1. From the empirical analysis, we summarize three lessons on designing hybrid DRAM-PMEM index data structure. We coined the lessons as ***buffered, out-of-place merging then caching(BOC)***.

The key BOC guidelines are that, the read-modify-write in XPLine and long latency of random access in NVDIMM is the fundamental performance bottleneck for persistent index data structure. We propose that an persistent index should ① buffer small writes in DRAM then update them to PMEM in a batch manner, and use ② out-of-place merging to reduce flush&fence. When there is pointer chasing problem in the indexes, we should ③ cache the searching path in DRAM to minimize the search latency.

4.3.1 Case Study: Buffering

Optane NVDIMM differs from DRAM in several ways. One of them is that there is a mismatch between CPU cache-line access granularity(64-byte) and the 3D-Xpoint media access granularity(256-byte) in both 1st and 2nd generation of Intel Optane NVDIMM devices[74]. To overcome this mismatch, Optane NVDIMM has a write-combining buffer (16KB) to merge small writes and reduce write amplification[74]. Given the small size, it is hard to exploit the locality and hit the buffer.

For writes in persistent index data structure, most of them are small writes, such as insertion of a new record(16-byte key-value pair), structural modification operations (SMO)

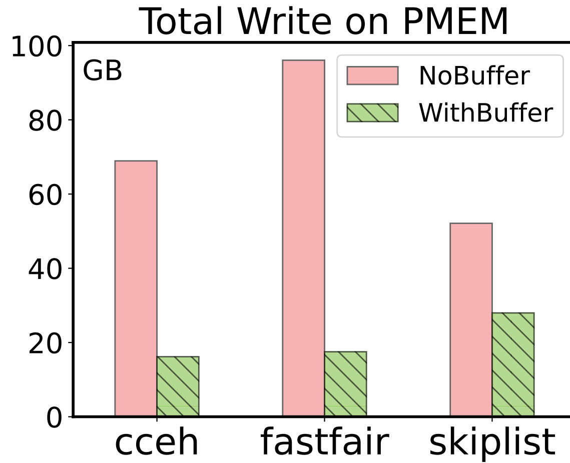


Figure 4.2: Total data written on PMEM with 120 million insertion of 16-byte key-value pairs.

in b+-tree, which may lack of locality considering the small write buffer in NVDIMM. With this mismatch between CPU and NVDIMM, most of the small writes result in read-modify-write operations, which leads to high write amplification and reduced effective memory bandwidth.

In this case study, we add write buffers in DRAM for all three data structures. For CCEH, write buffers with half of the segment size are allocated. For FastFair, each leaf node also has a write buffer with half of the leaf node size. P-Skiplist only create write buffers for nodes whose height is higher than two, and each buffer can store at most 16 records.

We run a benchmark to insert 120 million key-value pairs(8-byte key and 8-byte value). As shown in Figure 4.2, with write buffer, the total amount of the data written to PMEM can reduce by at most 4 times.

4.3.2 Case Study: Out-place-update

An potential issue of adding write buffer for persistent indexes is that when we merge the records in the buffer to PMEM in an in-place manner, we may need to repeatedly add flush&fence within a small contiguous range of PMEM space (segments in CCEH, leaf nodes in FastFair) to maintain crush consistency in case of power failure . It is reported that reading a recently flushed cacheline after fence instructions could experience much higher latency as the read has to wait the flush to complete[74].

To analyze the effect of flush&fence in buffer merging, we implement out-of-place buffer merging in CCEH and FastFair. Since P-Skiplist's key-value pairs are stored sepa-

rately in the linked list nodes, it is always an out-of-place manner for insertion. So we do not consider P-Skiplist in this case.

In CCEH, when a write buffer of a segment is full, first we copy the segment to DRAM. Then we merge the records in write buffer to the DRAM copy and write it back to PMEM in a new space. Finally we atomically change the directory pointer to this new segment. For FastFair, the similar approach is also employed for the leaf nodes. When merging the write buffer, a new leaf node is created to hold all the records in both the old leaf node and write buffer. Then the new leaf replace the old leaf by changing the parent pointer and left sibling's next pointer.

As shown in Figure 4.3's first row, for the single thread results, the performance of the indexes which use in-place merging for write buffers (cceh-B and fastfair-B) is slower than that which use out-of-place merging strategy (cceh-BO and fastfair-BO). Though both of them have similar I/O, the main reason is the flush&fence overhead.

4.3.3 Case Study: Caching

For index data structures whose operations contain many random reads (in the form of pointer chasing), the performance could be bottlenecked by slow random NVDIMM reads. Read latency on Optane NVDIMM is considerably higher than DRAM(2X-3X) because reads need to fetch data from the 3D-Xpoint media, which has longer media latency [77]. Meanwhile, most of the pointer chasing happen in the internal nodes whose size may only occupy small portion of the entire data structure size. So it is worth caching the internal nodes in DRAM to boost the lookup performance.

In this case study, we cache the internal nodes in all three indexes (directory in CCEH, inner nodes of FastFair, nodes with height higher than 2 in P-Skiplist) and measure their insert/read latency for 120 million requests.

As shown in Figure 4.4, caching internal nodes have almost no effect on CCEH as it only has one pointer chasing for each operation, which is locating the segment from directory. However, we observe significant improvement for FastFair and P-Skiplist as they need to do more pointer chasing(random reads) in internal nodes before finding a target node.

4.3.4 Combine Together

In the previous three case studies, we summarize the guidelines as ***buffered, out-of-placed merging then caching(BOC)***. We apply those guidelines to all three indexes one by one and measure the performance improvement during insertion of 120 million key-value pairs.

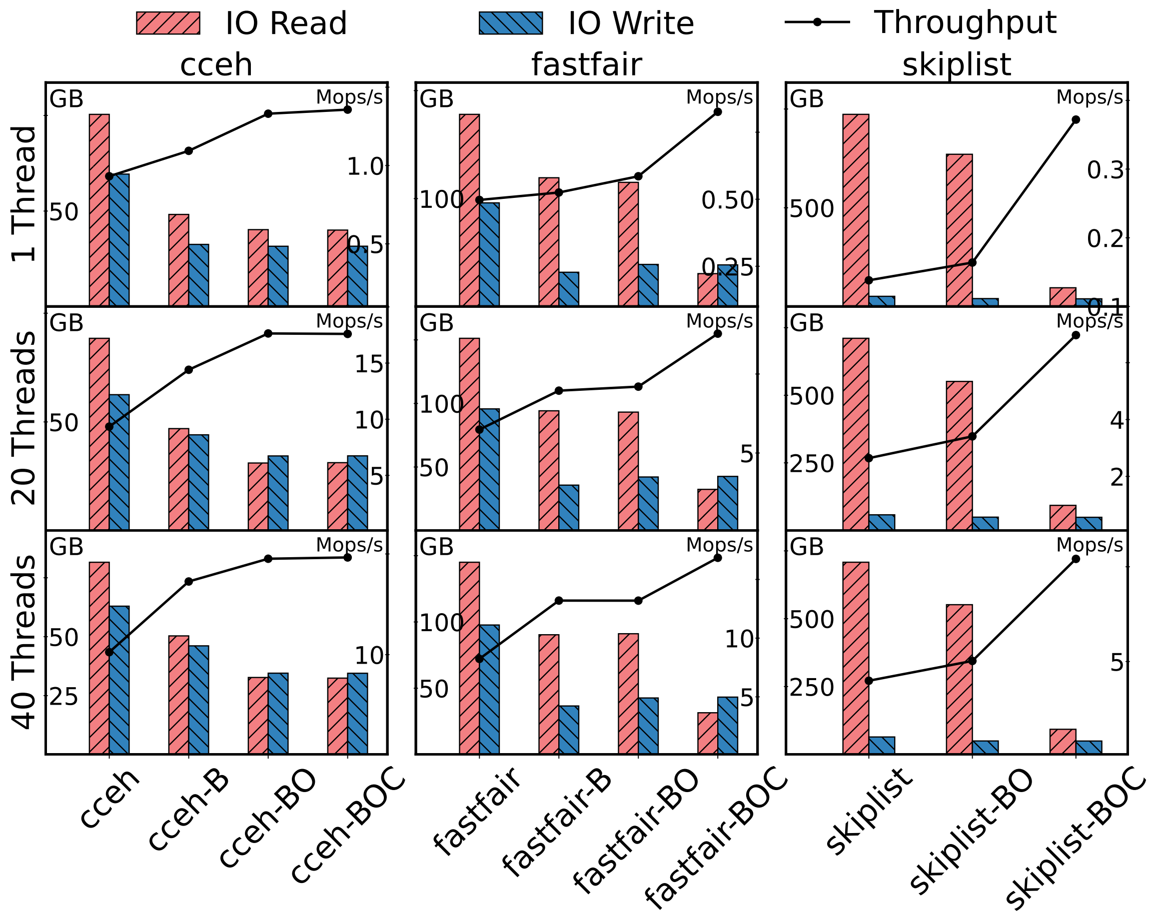


Figure 4.3: I/O and throughput for inserting 120 million key-value pairs (8-byte key and 8-byte value).

As shown in Figure 4.3, the write buffer not only reduces write amplification but also read amplification. This is not a surprise since during the buffer merging, the PMEM data to be merged can be cached in DRAM. The cost of reads is amortized via batching. Meanwhile, the benefits of out-of-place update (merging) are consistent from single thread case to multi-thread case. The throughput of out-of-place version (cceh-BO, fastfair-BO) is always higher than the in-place version (cceh-B, fastfair-B). Finally, caching the internal nodes not only reduces latency of pointer chasing, most importantly it reduces huge amount of read I/O during search (fastfair-BOC, skiplist-BOC), and significantly improves the effective PMEM memory bandwidth.

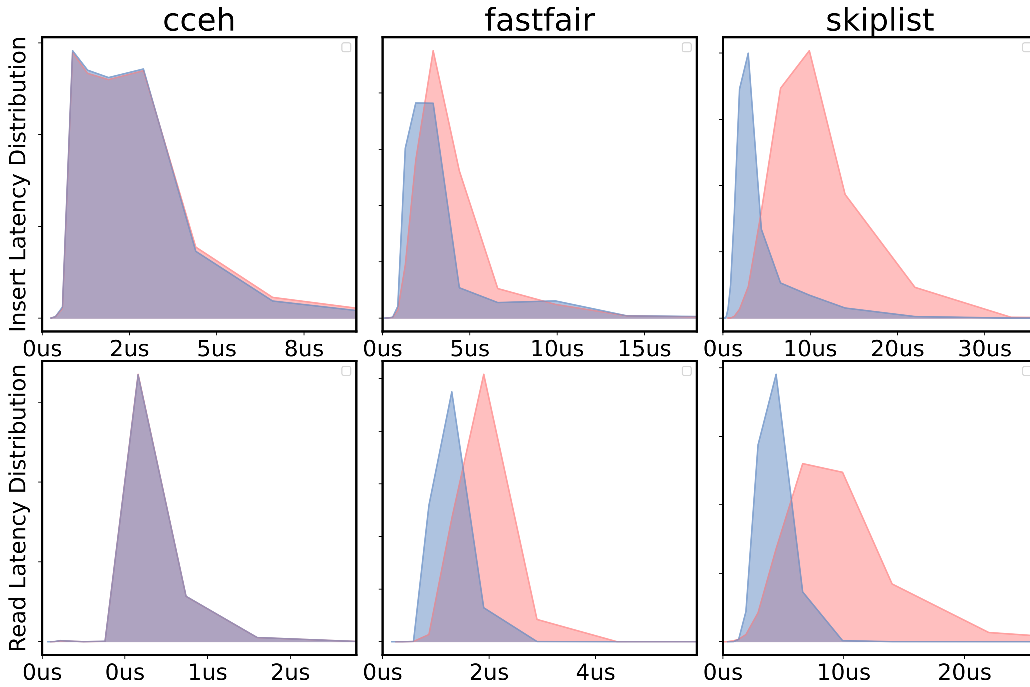


Figure 4.4: Case study for Insert/Read latency. Read area represent the results for indexes with caching.

4.4 Design

Following the guidelines, we propose SPTree, a DRAM-PMEM hybrid persistent tree index that utilize caching and buffering to address issues such as, long latency of pointer chasing, write amplification due to mismatch between key-value pair size and XPLine size(256-byte), quick recovery after power failure. As show in Figure 4.5, SPTree consists of three layers: toy layer, middle layer, and bottom layer. The top layer caches the internal nodes in DRAM. Meanwhile it also has a PMEM backup which is updated asynchronously. If a crash happens, we can recover the internal nodes from the PMEM backup instantly. The middle layer caches the inserted keys in bloom filters, which can filter out most of the point queries for non-existing keys. Meanwhile, the middle layer can also be assigned with write buffers based on available DRAM resources for write-intensive workloads to reduce write amplification. The bottom layer stores the leaf nodes on PMEM. The leaf node groups fingerprints in an array (tags) for a quick preliminary search. Meanwhile, It applies a two-phase insertion method to reduce the usage of flush and fence instructions. The first phase writes key-value pairs and theirs tags to the leaf node. Then the second phase will validate the bitmap in the leaf node to expose the inserted records. In this way, only two flush&fence are needed even for the batch insertion.

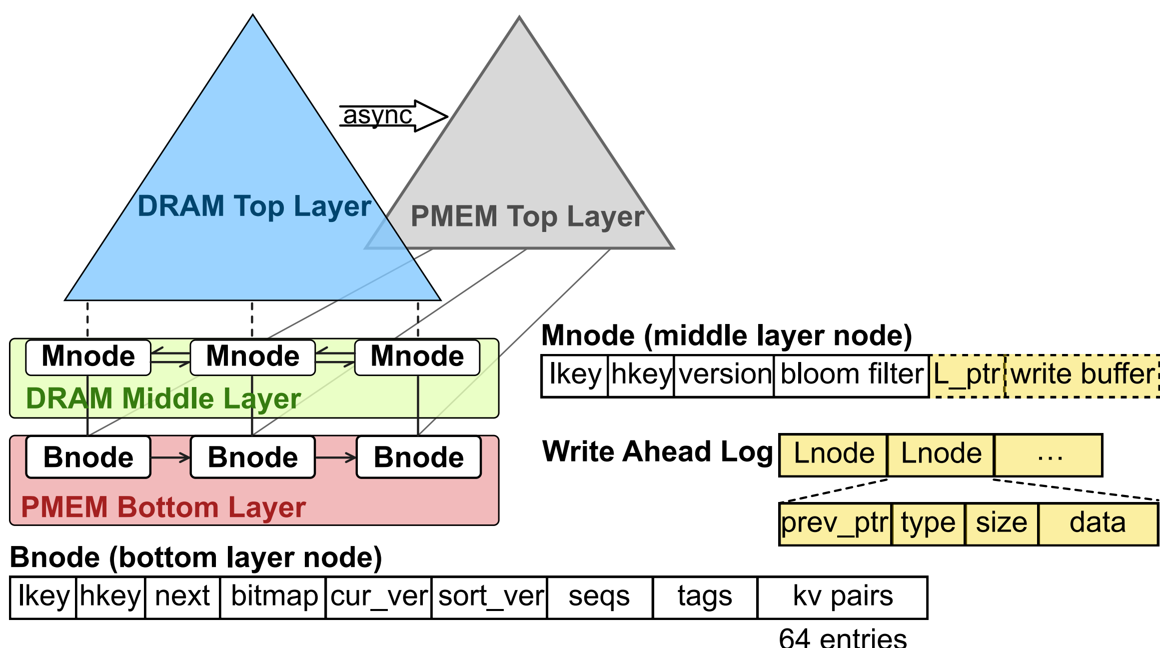


Figure 4.5: SPTree Architecture

4.4.1 Three layers in SPTree

Top Layer. The top layer of SPTree consists of two parts, the DRAM part and PMEM part. The DRAM top layer takes charge of indexing the middle layer nodes, which contain the information (key range) of the bottom layer's leaf nodes. SPTree uses the DRAM top layer to address the high latency issue of pointer chasing in PMEM, i.e., all operations in SPTree only search in the DRAM top layer before a target leaf node in the bottom layer is found. We modify the artree[45] as the DRAM top layer. In SPTree, the PMEM top layer is used for recovery after a reboot or power failure. It is updated asynchronously by background threads to move the slow updates in PMEM off critical path. Every time when a leaf node split, the new leaf node's indexing information is synchronously updated in the DRAM top layer, and sent to background threads and then propagated to a persistent b+-tree (FastFair) asynchronously. If some of the updates have not been updated in the PMEM top layer when a power failure happens, we can still recover those missing information by checking the possible smallest key(lkey) and largest key(hkey) in the leaf nodes. Please refer the following section of recover for details.

Middle Layer. The middle layer consists of Mnodes(middle layer nodes) in DRAM. Mnode stores the key range information of a leaf node belonging to this Mnode, which are lkey (smallest key) and hkey (largest key). When a search reaches to a Mnode, the

search key is checked with the key range in the Mnode. If the search key does not belong to this Mnode, we need to go back to the top layer to retry. A Mnode also stores a bloom filter which remembers all the existing keys in its leaf node. Every insertion will update the bloom filter in its belonged Mnode. In this way, by caching the existing key in a probabilistic manner, we can avoid most of non-existing key search to the PMEM leaf node. A Mnode may also be configured with a write buffer, which buffers all the new inserted key-value pairs. Once a buffer is full, all the records in the buffer will be merged to the relative leaf node.

Write Ahead Log. If some of the Mnodes contain write buffers, then the key-value pairs stored in those buffers may be lost during a power failure. To address issue, SPTree also keeps a write-ahead-log (WAL) for recovery if the write buffer is enabled in Mnodes.

Bottom Layer. The leaf nodes, or Bnodes (bottom layer nodes) are the data nodes which store all the key-value pairs in persistent memory. The Bnodes are organized a singly linked-list. Each Bnode can store 64 key-value pairs. Key fingerprints (tags) are used for a quick preliminary search of all keys by using SIMD instruction (`_mm_cmpeq_epi8_mask`). Two-phase insertion is enable in the Bnode, which writes batched key-value pairs in the slots first followed with one flush&fence. Then the bitmap is updated with another flush&fence. In this way, SPTree minimizes the usage of flush&fence to improve efficiency.

4.4.2 Concurrent Control

SPTree relys on `Optimistic Lock Coupling`[46] for concurrent control. An optimistic lock consists of a lock and a version counter (packed in 8-byte). For writers, the optimistic lock provides exclusion which only allows one writer at a time. When unlock, the lock is released and the version number increases by one atomically. For readers, they do not acquire the lock. Instead, they wait until the write lock is freed. Then they will compare the version before and after reading the value. If the version changes, they will retry until success. Both of the DRAM top layer and the middle layer apply optimistic lock.

4.4.3 Search Operation

Search for a given key is the most frequent used operation in the SPTree. It is not only used to service a read request, but also has to be employed before every insert /update /delete /lookup /scan operation is performed. A search operation first traverses the DRAM top layer to find a largest key that is smaller or equal to the search key. This largest key is the lkey in a Mnode. Then the search key is compared with the hkey in the Mnode to ensure the search key is in the key range of this Mnode. Otherwise, if the search key is out of range (probably due to split), we then travel around in the middle layer (double linked

list) to locate the correct Mnode. During the search operation, the PMEM top layer is never touched since it is only used for recovery.

4.4.4 Insert/Update/Delete/Lookup/scan

Insert. The thread servicing a insert request first searches the DRAM top layer to locate a Mnode. A necessary middle layer traversal is performed to jump to the target Mnode, whose key range includes the inserted key. Then the write lock is acquired on Mnode. If the Mnode contains a write buffer, the key-value pair is inserted into the write buffer, and appended to a write ahead log (WAL) for crash consistency. Otherwise, the thread inserts the key-value pair directly in the related Bnode on PMEM. In the first case, if write buffer is full, SPTree applies a minor compaction to flush all the key-value pairs to the related Bnode, and clear the write buffer. SPTree uses two-phase insertion to do minor compaction. ① All the key-value pairs and their tags are written to the Bnode followed by one flush&fence. ② The bitmap to indicate valid items is set and the version (cur_ver) advances by one. Meanwhile, A split is conducted if necessary. Finally, the bloom filter in Mnode is set for the inserted key.

Update. Update is similar with insert for locating the target Mnode. After a target Mnode is found, the bloom filter is checked to see if the key could exist. If not, we return directly. Otherwise we acquire the write lock. If there is a write buffer in the target Mnode, we try to update the key-value in the buffer, and write the new value to the WAL if update success. Otherwise, we check the related Bnode for update.

Delete. For delete, after we reach a target Mnode, we also check the bloom filter first to avoid unnecessary access to PMEM. If bloom filter return true, then we try to remove the search key in write buffer if it exists in the Mnode. A successful remove in Mnode's write buffer will be followed with an WAL write to record this deletion. Whether the Mnode has write buffer or not, we always try to remove the search key in the Bnode. This is different with update which will return if update success in the write buffer. The reason is that if there is a write buffer in Bnode, we may have a new key and an old key in the buffer and Bnode, respectively. Once we delete the new key in the buffer without removing the old key in Bnode, a follow up read for this key will return wrong answer (the old key-value pair instead of NULL).

Lookup. Lookup operation in SPTree is lock-free read. By applying the optimistic lock, we will check the version in Mnode before and after the read. If the version does not change, that means no inserts happen during the read and we return the value safely. Otherwise, we retry lookup from the beginning.

Scan. Scan operation first finds the target Mnode whose lkey is equal or smaller than the lower bound of the scan range, while the hkey is larger than the lower bound. Then it checks whether a reconstruct of the sequence array (`seqs`) is needed by comparing `cur_ver` and `sort_ver`. If `cur_ver` is larger than `sort_ver`, it means new inserts happen since last last reconstruct of the `seqs`, then we acquire the write lock and sort the keys in Bnode and store the ascending orders in the `seqs` array. Meantime, we set `sort_ver` equal to `cur_ver` to indicate this Bnode is ready for scan. When there are keys stored in the Mnode write buffer, a minor compaction is triggered before scanning. During the scan of a Bnode, the keys fall in the scan range are collected in a buffer. Then the value of `cur_ver` before and after the scan is compared. If they are equal, those buffered key-value pairs are appended in the output array. Otherwise, we retry scan in this Bnode. Scan continues to the sibling Bnode until the output array is full or the key is out of scan range.

4.4.5 Split and Merge

When a Bnode (leaf node) is full, SPTree conducts a split, which creates a new Bnode along with its Mnode. Then a [lkey, Mnode pointer] mapping is inserted to the DRAM top layer while a [lkey, Bnode pointer] is sent to the background thread to asynchronously update the PMEM top layer. During split, ① the writer first acquires the lock for current Bnode and its next sibling. ② Then it allocates a new Bnode, move the right half of the key-value pairs from the full node to the new Bnode, and set the `next` pointer in the full node pointing to the new node. These three operations are atomically conducted using the leak-free PMEM allocator (such as Intel PMDK's `pmemobj_alloc()`) to prevent memory leak. ③ Then `bitmap` and `hkey` in the full Bnode is modified to evict the split-out keys. ④ Last, the mapping information is updated in the DRAM top layer and propagate to the PMEM top layer.

When a delete operation detects that two adjacent nodes have keys less than the half of one node's capacity, a merge operation is triggered. ① We first acquire the two nodes' write locks. ② Then we shift the key-value pairs in the right Bnode to the left Bnode using two-phase insertion (set the kv pairs first, then the bitmap). ③ Next we modify the left Bnode's hkey as the the right Bnode's hkey, mark the right Bnode as deleted in its `cur_ver` and drop the Bnode. ④ Finally we update the mapping in the top layer.

4.4.6 Recovery and Crash Consistency

Since SPTree uses DRAM top layer and DRAM middle layer to provide service, these two layers need to be reconstructed after a reboot or a power failure. SPTree can rebuild the DRAM top layer and Mnodes in a quick way. All the updates in the DRAM top

layer are propagate to the PMEM top layer, and the PMEM top layer stores the pointers to the Bnodes in PMEM. During recovery, SPTree can quickly collect all the pointers to Bnodes by scanning the PMEM top layer, which is only around 2% of entire index size. Then we rebuilds the Bnode's Mnode and DRAM top layer parallelly, which is not possible if we scan the singly linked list of Bnodes one by one on recovery.

If this recovery is after a crash, then we need to fix the incomplete state. If a crash happens before split step ② complete, SPTree can recover to the state before split (guaranteed by the allocator). If crash happens after step ② and before step ③, then a dummy leaf node is linked to bottom layer. The incomplete status looks like following: [split node, lkey: 10, hkey:100] -> [dummy node, lkey: 50, hkey:100] -> [next node, lkey: 100, hkey:200] ... The dummy node will have a lkey lower than its previous node's hkey. If we find a dummy node during recovery, we fix it by dropping the dummy node. If crash happens during step ③ (hkey is set and the bitmap has not been set), then the split node will have keys which do not belong to it. This can be easily fixed by ignore those out-of-range keys during next split. If crash happens before step ④ finishes, we only need to fix the missing mapping in the top layer. We can identify the missing mappings by comparing the adjacent Mnode's lkey and hkey after a normal recovery. A Mnode's hkey is always same as its next sibling's lkey. If not, then we scan the linked list from current Mnode's Bnode to recover the missing Mnodes.

If a crash happens before a merge step ② competes, SPTree can recover to the state before merging because the shifted kv pairs have not been exposed by the bitmap. If the crash happens before ③, the shifted kv pairs can be filtered out using hkey. If the crash is after ③ setting the hkey and before dropping the right node, then the incomplete status looks like: [merge-left node, lkey: 10, hkey:100] -> [merge-right node, lkey: 50, hkey:100] -> [next node, lkey: 100, hkey:200]. The merge-right node has a lkey lower than merge-left node's hkey. We fix this by dropping the merge-right node during recovery. If the crash is before ④, the dummy mapping will point to a deleted Bnode, then we delete this mapping during recovery.

For consistency in Bnode, all the key-value pairs that have not completed the second phase (setting the bitmap) will not be exposed. So we will not observe any partially updated items. If some Mnodes have write buffers, the content in the buffer can be recovered after we replay the WAL.

4.5 Evaluation

In this section, we experimentally evaluate SPTree by comparing it with several state-of-the-art B+-Tree for persistent memory, including FastFair[34] and PACTree[39].

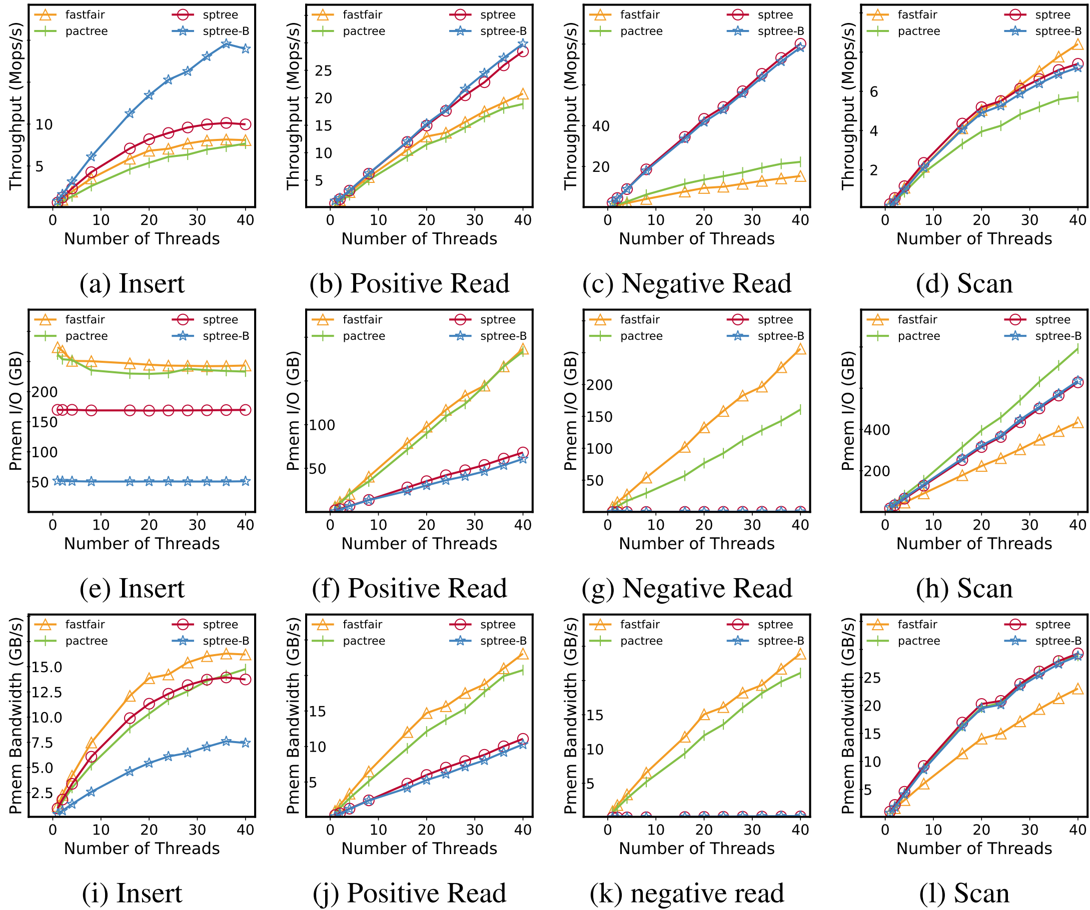


Figure 4.6: Throughput and PMEM I/O volume with different requests.

4.5.1 Experiment Setup

In our experiments, we use 16 byte key-value pairs for comparison. All the threads in an experiment are pinned to one socket using `numactl`. SPTree's DRAM footprint is about 12% of the total size of the PMEM bottom layer when write buffer is not enabled. If all the Mnodes in the SPTree have write buffer (sptree-B), the DRAM footprint is about 33%.

All the experiments are run on a server with an Intel Xeon Gold 6230 20-core processor, 64GB DRAM and 6×128GB Intel Optane DC.

4.5.2 Overall Performance

To evaluate the performance of the trees, we conduct extensive experiments, including insertions of new KV items (Insert), reading existing keys (Positive Read), reading

non-existing keys (Negative Read), range queries (Scan). Experiment results are shown in Figure 4.6. In each experiment different number of threads (from 1 to 40 threads) are used. For ***Insert*** each thread sends $120million/Number_of_threads$ requests. For ***Positive Read*** and *Negative Read*, *Scan* each thread sends 10 million requests. Figure 4.6 reports throughput of the btrees (number of requests serviced per second) and the corresponding raw PMEM I/O volume. This I/O volume represents all read/write data amount on the Optane PMEM's media, including amplified I/O due to existence of its 256B access unit. It is measured with `ipmwatch`, available in the Intel VTune Amplifie tool.

Insert. As shown in Figure 4.6a, SPTree always outperforms the others for insert performance. This is mainly because for the other two trees, the pointer chasing in the internal nodes causes large read amplification, resulting reduced effective memory bandwidth. As shown in Figure 4.6e and 4.6i, though the bandwidth of FastFair and PACTree during insertion is equal or higher than that of SPTree, their high I/O actually results in the lower performance. When the write buffer is enable for SPTree (sptree-B), the total I/O is reduced by 3X. This is mainly because by using write buffer, we reduce the write amplification as well as the read amplification as mentioned in section 4.3.4.

Positive Read. We see 25% performance improvement for SPTree over the others. This advantage mainly comes from the reduced I/O during search in the internal nodes. As shown in Figure 4.6f, SPTree's I/O is only 1/3 of the others. When write buffer is enabled, we see slightly performance improvement over the one without buffer. This is because the write buffer also functions as a read buffer for `Positive Read`. Hence a small portion of the request does not touch the Bnode (PMEM). That's why I/O of sptree-B is lower, as shown in Figure 4.6f.

Negative Read. SPTree has 4X performance improvement compared with the other two trees, as shown in Figure 4.6c. This is because we cache the existing keys in the Mnodes' bloom filter, which filters out most of the unnecessary PMEM access during **Negative Read**. As shown in Figure 4.6g and 4.6k, there is nearly no PMEM I/O and we barely read the PMEM. All the saved bandwidth can be used to servicing other requests.

Scan. Both SPTree and PACTree use an indirection array for sorting keys in the leaf node. This strategy comes with a cost of higher read amplification compared with the physically sorted key-value pairs in FastFair. As shown in Figure 4.6h, SPTree and PACTree have higher I/O during `Scan`. However, thanks to the low overhead of the DRAM top layer, SPTree's scan performance is only 10% lower than FastFair, while PACTree is 30% lower.

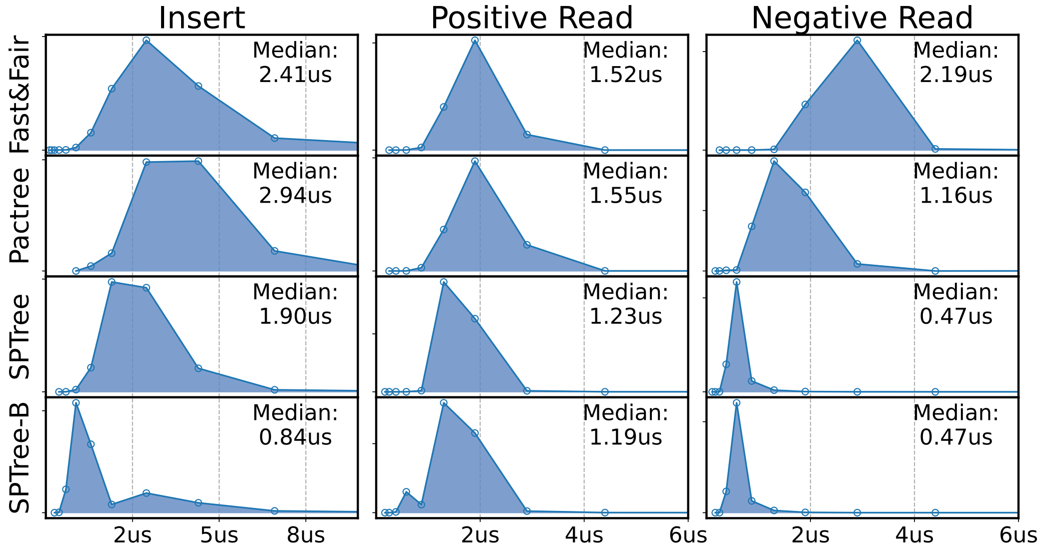


Figure 4.7: Latency Comparison

4.5.3 Latency Comparison

In this section, we evaluate the read/write latency of all the trees. We use 20 threads to write 120 million key-value items. Each thread sends 4 million read requests.

As shown in Figure 4.7, SPTree always has the lowest latency among the other two trees in all the test workloads (`Insert, Positive Read, Negative Read`). By caching the internal nodes in DRAM, the random pointer chasing problem in PMEM is avoided. Meanwhile, caching inserted keys in bloom filter also help SPTree avoid the access to PMEM for non-existing keys. In our practice, after 120 million key-value pairs are inserted, the false positive rate is around 5%. It is worth noting that when write buffer is enabled, the insert latency for SPTree (SPTree-B) reduces 3X compared with the one without write buffer, which is at the cost of only around 30% DRAM footprint.

4.5.4 Recovery

If write buffer for SPTree is not enabled, SPTree can start servicing the requests without rebuilding the DRAM top and middle layer because the PMEM top layer can be used. In the meantime, it can rebuild the DRAM layers in the background and then put it into usage. As shown in Figure 4.8, the time to rebuild the DRAM top and middle layer is negligible (1 second for 250 million keys) because we rebuild the DRAM layers based on the packed PMEM top layer in parallel.

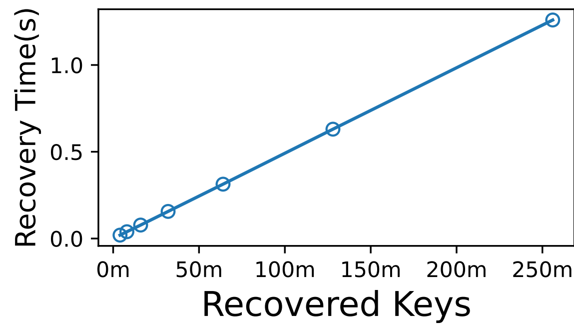


Figure 4.8: Recovery time to rebuild DRAM top and middle layer.

4.6 Summary

We introduce SPTree, a persistent B+-Tree designed for high-performance system in this paper. By enabling caching (internal nodes and existing keys) and write buffer at a cost of reasonable DRAM footprint, and two-phase writes to reduce usage of flush&fence, SPTree minimizes the PMEM I/O traffic and achieves 2X to 4X improvement of access performance over the state-of-the-art PMEM B+-Tree designs in terms of both throughput and latency.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

5.1 Contributions

The contribution of this dissertation can be summarized as follows:

- We introduce WipDB, a key-value store designed to manage small key-value items in a storage system of large capacity. By introducing approximate sorting and the write-in-place LSM-tree scheme, WipDB minimizes write amplification for LSM-tree-based KV stores. Meanwhile, the read-aware scheduling of compaction moves most compaction off the critical path of read service. Our results show that WipDB can significantly improve for both write and mixed read/write workloads.
- We design TurboHash, a persistent hash table designed for high-performance key-value stores in this paper. By enabling out-of-place update at a cost equivalent to that for an in-place write, conducting probing on a path sequentially and only for a necessary length, and utilizing Intel Optane DC's hardware feature, TurboHash minimizes the Pmem I/O traffic and achieves $2\times$ to $8\times$ improvement of access performance over state-of-the-art Pmem hash table designs
- We propose Spot-on, which turns an index structure designed for persistent memory into a much faster one with application-managed caching and buffering. We develope SPTree, which provides higher write and read throughput up to 2X - 4X respectively compared with the state-of-the-art design.

5.2 Future Work: Data rearrangement supporting log-structured storage

Log-structured stores have been widely used in storage systems for storing massive amount of data over the past decades [41, 42, 17, 75, 63, 47, 56, 11, 49, 70]. It is a log-based architecture that optimizes write by avoiding random writes. Since all writes, including updates, are appended in a circular log, an in-memory index is maintained to retrieve the valid data. As a consequence, the space occupied by invalid data is recycled by garbage collection (GC) process. For update-intensive workloads, as more requests are received, frequent GC operations will be triggered, leading to massive I/Os due to the movement of valid data. Such frequent GC incurs a high write amplification (WA). A work [49] reports

that the WA can reach to 20× for update-intensive workload with Zipfian distribution. The high WA is harmful to both performance and SSD endurance as SSD has limited amount of P/E cycles [33].

To reduce the write amplification caused by frequent garbage collection, many research works focus on improving GC efficiency. One approach, HashKV [11], uses hashing to partition the data based on the access frequency (hotness awareness) such that hot data and cold data can be separated into different logs. The main idea of the hotness awareness is that updates to the hot data can be accumulated and be recycled together, which reduces the rewrite of the cold data and decreases WA. Another optimization (KVell[47]) allows overwriting in the append log. All insertions are appended to a circular log while a list is maintained to archive the position of the deleted data. Those spaces are used to store new records of the same size. However, there are restrictions for those optimizations to work. For HashKV, it inevitably rewrites the cold data to a separate log during GC to enable Hot-cold data separation. The second work only supports fixed-size records.

For cloud storage that is built on the distributed log-structured store, network throughput is another concern besides write amplification. As most of the data movement passes through the network, high write amplification may cause high network saturation, which harms the quality of service in terms of latency as delivering high network throughput results in rising latency. Mohammad et al.[1] provides a solution that sacrifices part of network bandwidth for latency. By capping utilization of the link capacity (e.g. 90%), they free up space for sensitive requests and avoid buffering as well as the associated large delays. However, this is a trade-off between latency and available network bandwidth. For distributed log-structured stores, as the high WA may quickly saturate the entire network capacity, there will not be available network resources for this trade-off.

With the support of new hardware and software design, e.g., page re-mapping support in the FTL of SSDs, the data movement cost can be optimized and reduced considerably. I am interested in analyzing the workload and the distribution of garbage data, and extend my research to support efficient data movement in log-structured store design in the future.

REFERENCES

[1] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 253–266, San Jose, CA, 2012. USENIX.

[2] A. Appleby. Murmurhash. `https://sites.google.com/site/murmurhash/`, 2008.

[3] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson. Bztree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment*, 11(5):553–565, 2018.

[4] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[5] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka. Triad: Creating synergies between memory, disk and log in log structured key-value stores. In *USENIX ATC '17*, pages 363–375, 2017.

[6] O. Balmau, R. Guerraoui, V. Trigonakis, and I. Zablotchi. Flodb: Unlocking memory in persistent key-value stores. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 80–94, New York, NY, USA, 2017. ACM.

[7] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 47–60, USA, 2010. USENIX Association.

[8] A. D. Breslow, D. P. Zhang, J. L. Greathouse, N. Jayasena, and D. M. Tullsen. Horton tables: Fast hash tables for in-memory data-intensive computing. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 281–294, Denver, CO, June 2016. USENIX Association.

[9] W. Cai, H. Wen, H. A. Beadle, M. Hedayati, and M. L. Scott. Understanding and optimizing persistent memory allocation. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, page 421–422, New York, NY, USA, 2020. Association for Computing Machinery.

[10] H. H. Chan, C.-J. M. Liang, Y. Li, W. He, P. P. Lee, L. Zhu, Y. Dong, Y. Xu, Y. Xu, J. Jiang, et al. Hashkv: Enabling efficient updates in {KV} storage via hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 1007–1019, 2018.

[11] H. H. W. Chan, Y. Li, P. P. C. Lee, and Y. Xu. Hashkv: Enabling efficient updates in kv storage via hashing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 1007–1019, USA, 2018. USENIX Association.

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[13] J. Chen, L. Chen, S. Wang, G. Zhu, Y. Sun, H. Liu, and F. Li. Hotring: A hotspot-aware in-memory key-value store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 239–252, Santa Clara, CA, Feb. 2020. USENIX Association.

[14] S. Chen and Q. Jin. Persistent b+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7):786–797, Feb. 2015.

[15] Z. Chen, Y. Huang, B. Ding, and P. Zuo. Lock-free concurrent level hashing for persistent memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 799–812, online, July 2020. USENIX Association.

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[17] H. Dai, M. Neufeld, and R. Han. Elf: An efficient log-structured flash file system for micro sensor nodes. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, page 176–187, New York, NY, USA, 2004. Association for Computing Machinery.

[18] T. David, R. Guerraoui, and V. Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. *ACM SIGARCH Computer Architecture News*, 43(1):631–644, 2015.

[19] A. Davoudian, L. Chen, and M. Liu. A survey on nosql stores. *ACM Comput. Surv.*, 51(2), Apr. 2018.

[20] N. Dayan and S. Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 505–520, New York, NY, USA, 2018. ACM.

[21] N. Dayan and S. Idreos. The log-structured merge-bush & the wacky continuum. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 449–466, New York, NY, USA, 2019. Association for Computing Machinery.

[22] B. Debnath, A. Haghdoost, A. Kadav, M. G. Khatib, and C. Ungureanu. Revisiting hash table design for phase change memory. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '15, New York, NY, USA, 2015. Association for Computing Machinery.

[23] B. Debnath, A. Haghdoost, A. Kadav, M. G. Khatib, and C. Ungureanu. Revisiting hash table design for phase change memory. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '15, New York, NY, USA, 2015. Association for Computing Machinery.

[24] B. Debnath, S. Sengupta, and J. Li. Flashstore: High throughput persistent key-value store. *Proc. VLDB Endow.*, 3(1–2):1414–1425, Sept. 2010.

[25] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[26] Facebook. Rocksdb. https://rocksdb.org.

[27] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing—a fast access method for dynamic files. *ACM Trans. Database Syst.*, 4(3):315–344, Sept. 1979.

[28] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384, Lombard, IL, Apr. 2013. USENIX Association.

[29] K. Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, Feb. 2004.

[30] Google. Leveldb. https://github.com/google/leveldb, 2020.

[31] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch hashing. In G. Taubenfeld, editor, *Distributed Computing*, pages 350–364, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[32] D. Hu, Z. Chen, J. Wu, J. Sun, and H. Chen. Persistent memory hash indexes: An experimental evaluation. *Proc. VLDB Endow.*, 14(5):785–798, Jan. 2021.

[33] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, New York, NY, USA, 2009. Association for Computing Machinery.

[34] D. Hwang, W.-H. Kim, Y. Won, and B. Nam. Endurable transient inconsistency in Byte-Addressable persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200, Oakland, CA, Feb. 2018. USENIX Association.

[35] Intel. Intel optane persistent memory. `https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html`, 2021.

[36] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, et al. Betrfs: Write-optimization in a kernel file system. *ACM Transactions on Storage (TOS)*, 11(4):1–29, 2015.

[37] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y. 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, Feb. 2019. USENIX Association.

[38] 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)*, pages 993–1005, Boston, MA, July 2018. USENIX Association.

[39] W.-H. Kim, R. M. Krishnan, X. Fu, S. Kashyap, and C. Min. Pactree: A high performance persistent range index using pac guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 424–439, New York, NY, USA, 2021. Association for Computing Machinery.

[40] D. E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., USA, 1998.

[41] J. T. Kohl, C. Staelin, and M. Stonebraker. Highlight: Using a log-structured file system for tertiary storage management. In *USENIX Winter*, 1993.

[42] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai. The linux implementation of a log-structured file system. *SIGOPS Oper. Syst. Rev.*, 40(3):102–107, July 2006.

[43] R. M. Krishnan, W.-H. Kim, X. Fu, S. K. Monga, H. W. Lee, M. Jang, A. Mathew, and C. Min. TIPS: Making volatile index structures persistent with dram-nvmm tiering. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 773–787. USENIX Association, July 2021.

[44] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 462–477, New York, NY, USA, 2019. Association for Computing Machinery.

[45] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 38–49, Washington, DC, USA, 2013. IEEE Computer Society.

[46] V. Leis, F. Scheibner, A. Kemper, and T. Neumann. The art of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, DaMoN '16, New York, NY, USA, 2016. Association for Computing Machinery.

[47] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel. Kvell: The design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 447–461, New York, NY, USA, 2019. Association for Computing Machinery.

[48] B. Lu, X. Hao, T. Wang, and E. Lo. Dash: Scalable hashing on persistent memory. *Proc. VLDB Endow.*, 13(8):1147–1161, Apr. 2020.

[49] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 133–148, Santa Clara, CA, Feb. 2016. USENIX Association.

[50] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Trans. Storage*, 13(1):1–28, 2017.

[51] F. Mei, Q. Cao, H. Jiang, and J. Li. Sifrdb: A unified solution for write-optimized key-value stores in large datacenter. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 477–489, New York, NY, USA, 2018. ACM.

[52] M. Nam, H. Cha, Y. ri Choi, S. H. Noh, and B. Nam. Write-optimized dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 31–44, Boston, MA, Feb. 2019. USENIX Association.

[53] F. Ni and S. Jiang. Rapidcdc: Leveraging duplicate locality to accelerate chunking in cdc-based deduplication systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 220–232, New York, NY, USA, 2019. Association for Computing Machinery.

[54] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling memcache at facebook. In *USENIX NSDI '13*, pages 385–398, 2013.

[55] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, Apr. 2013. USENIX Association.

[56] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley db. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, page 43, USA, 1999. USENIX Association.

[57] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[58] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 371–386, New York, NY, USA, 2016. Association for Computing Machinery.

[59] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, May 2004.

[60] S. Patil and G. Gibson. Scale and concurrency of giga+: File system directories with millions of files. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*, volume 11, pages 13–13, San Jose, CA, Feb. 2011. USENIX Association.

[61] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, June 1990.

[62] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 497–514, New York, NY, USA, 2017. ACM.

[63] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 497–514, New York, NY, USA, 2017. Association for Computing Machinery.

[64] K. Ren and G. Gibson. Tablefs: Enhancing metadata efficiency in the local file system. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 145–156, Berkeley, CA, USA, 2013. USENIX Association.

[65] M. Sadoghi, S. Bhattacherjee, B. Bhattacharjee, and M. Canim. L-store: A real-time OLTP and OLAP system. In M. H. Böhlen, R. Pichler, N. May, E. Rahm, S. Wu, and K. Hose, editors, *EDBT 2018, Vienna, Austria, March 26-29, 2018*, pages 540–551. OpenProceedings.org, 2018.

[66] F. Schmuck and R. Haskin. Gpfs: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, page 19–es, USA, 2002. USENIX Association.

[67] R. Sears and R. Ramakrishnan. blsm: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 217–228, New York, NY, USA, 2012. ACM.

[68] P. Shetty, R. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok. Building workload-independent storage with vt-trees. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, pages 17–30, Berkeley, CA, USA, 2013. USENIX Association.

[69] R. Thonangi and J. Yang. On log-structured merge for solid-state drives. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 683–694. IEEE, 2017.

[70] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi. Logbase: A scalable log-structured database system in the cloud. *Proc. VLDB Endow.*, 5(10):1004–1015, June 2012.

[71] Q. Wang, Y. Lu, J. Li, and J. Shu. Nap: A black-box approach to numa-aware persistent memory indexes. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 93–111. USENIX Association, July 2021.

[72] X. Wu, F. Ni, and S. Jiang. Search lookaside buffer: Efficient caching for index data structures. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 27–39, New York, NY, USA, 2017. Association for Computing Machinery.

[73] X. Wu, Y. Xu, Z. Shao, and S. Jiang. Lsm-trie: An lsm-tree-based ultra-large key-value store for small data. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 71–82, Berkeley, CA, USA, 2015. USENIX Association.

[74] L. Xiang, X. Zhao, J. Rao, S. Jiang, and H. Jiang. Characterizing the performance of intel optane persistent memory: A close look at its on-dimm buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 488–505, New York, NY, USA, 2022. Association for Computing Machinery.

[75] J. Xu and S. Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, Feb. 2016. USENIX Association.

[76] B. Yan, X. Cheng, B. Jiang, S. Chen, C. Shang, J. Wang, K. Huang, X. Yang, W. Cao, and F. Li. Revisiting the design of lsm-tree based OLTP storage engine with persistent memory. *Proc. VLDB Endow.*, 14(10):1872–1885, 2021.

[77] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, Feb. 2020. USENIX Association.

[78] 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)*, pages 17–31, online, July 2020. USENIX Association.

[79] Y. Yue, B. He, Y. Li, and W. Wang. Building an efficient put-intensive key-value store with skip-tree. *IEEE Transactions on Parallel and Distributed Systems*, 28(4):961–973, April 2017.

[80] L. Zhang and S. Swanson. Pangolin: A fault-tolerant persistent memory programming library. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 897–912, Renton, WA, July 2019. USENIX Association.

[81] 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*, EuroSys '21, page 194–209, New York, NY, USA, 2021. Association for Computing Machinery.

[82] zhichao Cao, S. Dong, S. Vemuri, and D. H. Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, Santa Clara, CA, Feb. 2020. USENIX Association.

[83] P. Zuo and Y. Hua. A write-friendly and cache-optimized hashing scheme for non-volatile memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 29(5):985–998, 2018.

[84] P. Zuo, Y. Hua, and J. Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 461–476, Carlsbad, CA, Oct. 2018. USENIX Association.

[85] P. Zuo, J. Sun, L. Yang, S. Zhang, and Y. Hua. One-sided rdma-conscious extendible hashing for disaggregated memory. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 15–29. USENIX Association, July 2021.

[86] Y. Zuriel, M. Friedman, G. Sheffi, N. Cohen, and E. Petrank. Efficient lock-free durable sets. *Proc. ACM Program. Lang.*, 3(OOPSLA), Oct. 2019.

BIOGRAPHICAL STATEMENT

Xingsheng Zhao was born in Xuancheng, Anhui in 1991. He received his B.E., M.E. degrees of Automation from Southeast University, Nanjing, China, in 2013 and 2016, respectively. He received his Ph.D. degree of Computer and Information Sciences from University of Texas at Arlington in 2022. His main areas of research interest are high performance key-value stores, persistent index data structures and storage system.