FluidKV: Seamlessly Bridging the Gap between Indexing Performance and Memory-Footprint on Ultra-Fast Storage

Ziyi Lu
Huazhong University of Science and Technology
China
luziyi@hust.edu.cn

Qiang Cao*
Huazhong University of Science and Technology
China
caoqiang@hust.edu.cn

Hong Jiang
UT Arlington
TX, USA
hong.jiang@uta.edu

Yuxing Chen
Tencent Inc.
China
axingguchen@tencent.com

Jie Yao
Huazhong University of Science and Technology
China
jackyao@hust.edu.cn

Anqun Pan
Tencent Inc.
China
aaronpan@tencent.com

ABSTRACT

Our extensive experiments reveal that existing key-value stores (KVSs) achieve high performance at the expense of a huge memory footprint that is often impractical or unacceptable. Even with the emerging ultra-fast byte-addressable persistent memory (PM), KVSs fall far short of delivering the high performance promised by PM's superior I/O bandwidth. To find the root causes and bridge the huge performance/memory-footprint gap, we revisit the architectural features of two representative indexing mechanisms (single-stage and multi-stage) and propose a three-stage KVS called FluidKV. FluidKV effectively consolidates these indexes by fast and seamlessly running incoming key-value request stream from the write-concurrent frontend stage to the memory-efficient backend stage across an intermediate stage. FluidKV also designs important enabling techniques, such as thread-exclusive logging, PM-friendly KV-block structures, and dual-grained indexes, to fully utilize both parallel-processing and high-bandwidth capabilities of ultra-fast storage hardware while reducing the overhead. We implemented a FluidKV prototype and evaluated it under a variety of workloads. The results show that FluidKV outperforms the state-of-the-art PM-aware KVSs, including ListDB and FlatStore with different indexes, by up to 9× and 3.9× in write and read throughput respectively, while cutting up to 90% of the DRAM footprint.

PVLDB Reference Format:
Ziyi Lu, Qiang Cao, Hong Jiang, Yuxing Chen, Jie Yao, and Anqun Pan. FluidKV: Seamlessly Bridging the Gap between Indexing Performance and Memory-Footprint on Ultra-Fast Storage. PVLDB, 17(6): 1377 - 1390, 2024.
doi:10.14778/3648160.3648177

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/luziyi23/FluidKV.

*Qiang Cao is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 17, No. 6 ISSN 2150-8097.
doi:10.14778/3648160.3648177

1 INTRODUCTION

Persistent memory (PM) and solid-state drive (SSD) have made great strides in both I/O bandwidth and latency over the past decade. PM, with its byte-addressability and fast persistency, is providing researchers with an opportunity to reshape the storage landscape. Meanwhile, key-value stores (KVSs), as a building block of modern data-processing platforms, maintain a global index on a series of persistent key-value pairs (KVs) to support simple-semantic KV-accesses (e.g., Get/Put/Scan). Therefore, KVS on ultra-fast storage has become the eye of the KVS research storm.

At the heart of a KVS is its indexing structure, and, depending on how keys are organized and operated on in this structure, for the purpose of this paper we broadly divide KVSs into two categories, *single-stage indexing KVS* and *multi-stage indexing KVS*. Specifically, the single-stage indexing KVSs have a monothetic data structure (e.g., B+-tree) in DRAM, storage, or DRAM-storage hybrid, to index all persistent KVs. On the other hand, the multi-stage indexing KVSs, e.g., Log-structured merge-tree (LSM-tree [40]), dynamically and periodically migrate the incoming KVs from a small in-memory KV-set to in-storage large KV-sets, each of which has its own corresponding index.

Existing PM-based KVSs, be they single-stage and multi-stage indexing, fall far short of achieving the high performance promised by ultra-fast storage (e.g., PM) without heavily relying on a huge DRAM capacity, as elaborated by our in-depth experimental analysis in §2 and summarized by the following 5 key observations.

For the single-stage indexing KVSs, the DRAM-only indexing ones, where the entire pivot index resides in DRAM [3, 9, 53], show superior concurrency and performance (*Observation 1*) at the cost of a huge DRAM footprint, making it difficult to adapt to the growth of data volume (*Observation 2*). Storing part of the index (e.g., leaf nodes) on PM [20, 31, 35, 57, 65] reduces DRAM footprint but sacrifices the overall performance.

The multi-stage indexing KVSs, such as LSM-tree, build an in-memory KV-grained index only for a limited amount of incoming unsorted data in the first stage and small coarse-grained indexes for the other stages, making DRAM footprint controllable (*Observation 3*), while introducing the notorious write stalls and write/read amplifications due to the periodic compactions to merge KVs between

adjacent stages. Recently, PM-aware LSM-trees [12, 30, 62] are designed to optimize for PM idiosyncrasy and reduce write stalls. However, both read and write performances of such multi-stage indexing KVSs remain far lower than DRAM-only single-stage indexing KVSs due to limited write-concurrency (*Observation 4*). In addition, both single-stage and multi-stage indexing KVSs still insufficiently utilize the bandwidth of PM (*Observation 5*).

In other words, it is very challenging for existing KVS indexing structures, single-stage or multi-stage, to achieve high write/read throughput and controllable DRAM footprint simultaneously. Fortunately, *Observation 5* offers a hint to effectively and efficiently leverage the power of modern hardware. To this end, we propose a fast-flowing three-stage KVS architecture, FluidKV, to seamlessly consolidate such two indexing mechanisms. The key idea behind FluidKV is to combine the high concurrency of single-stage indexes and the controllable DRAM consumption of multi-stage indexes by quickly and efficiently merging KVs from the former to the latter.

FluidKV comprises three consecutive processing stages, i.e., FastStore, BufferStore, and StableStore. FastStore employs a concurrent and key-grained index, along with thread-exclusive logging, to fast absorb incoming KVs in a sequential but unsorted manner, thus achieving high write performance. Adapting to available hardware bandwidth and fluctuating workload, BufferStore dynamically flushes FastStore data into a series of persisted and sorted KV-sets and merge-sorts them into the backend StableStore, to control the memory footprint in time. StableStore maintains a global key-range-grained index on large-scale persistent data, thus minimizing the DRAM footprint while maintaining read performance. FluidKV presents a PM-friendly index and data block structure in BufferStore and StableStore to store the persisted index and KVs to efficiently exploit the ultra-fast storage.

We implement a FluidKV prototype and evaluate it under a variety of workloads. The results show that FluidKV outperforms ListDB, a state-of-the-art PM-aware multi-stage indexing KVS, by up to 9x and 3.8x in write and read throughput respectively while cutting 90% of ListDB's DRAM footprint. Compared to state-of-the-art single-stage indexing KVSs, FluidKV also ensures a controllable DRAM footprint with similar or higher read/write performance.

The contributions of this paper include:

- An in-depth experimental analysis of the performance/memory-footprint gap among representative KVS indexing mechanisms;
- A three-stage fast-flowing KVS architecture that effectively utilizes the I/O capacity and parallel-processing capability of ultra-fast storage;
- A write-optimized first stage (FastStore), an adaptive second stage for fast data migration (BufferStore), and a DRAM-footprint-cutting PM-aware backend third stage (StableStore);
- Evaluation of a FluidKV prototype against representative PM-aware KVSs demonstrating high write performance, low DRAM footprint, and acceptable read performance.

2 BACKGROUND AND ANALYSIS

In this section, we provide the necessary background for and an in-depth analysis of the characteristics of emerging high-performance storage devices and indexing mechanisms of existing key-value stores (KVSs), which help reveal their performance pitfalls.

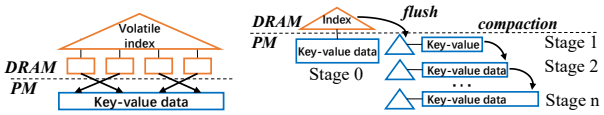


Figure 1: Classification of KVS indexing mechanism.

2.1 Ultra-Fast Storage

The rapid development of solid-state drive (SSD) and persistent memory (PM) technologies has unceasingly advanced both storage capacity and performance, especially accelerating bandwidth with increasing parallelism. Different from traditional block devices such as SSD and hard-disk drives (HDD), PM (e.g., PCM [52], STT-MRAM [2], Memristor [59], 3DXPoint[23], Memory-Semantic SSD [24, 25, 42, 60]) is capable of memory semantics (i.e., load/store) and accesses to byte-level small-sized data at GB/s-level I/O bandwidth and 100ns-level latency. Because of the superior byte-addressing capability and fast persistency, KVSs as a fundamental building block of modern data-processing infrastructure leverage PM to accelerate intensive small-sized KV workloads [16, 27, 29, 61] that are prevalent in industrial and commercial applications [5]. Nowadays, PM not only works as main memory, which can be accessed via memory bus for low latency, but can also be attached on the PCIe bus with the emerging CXL technology [1] for high scalability.

2.2 Key-value Store Indexing

A KVS generally consists of indexes and persistent data. In this paper, we focus on the index structure, which is the core of KVS for accurate and quick access to the persistent data on storage. To understand the impact of different indexing mechanisms on the overall performance of a KVS, we first classify the existing KVS indexes into two groups, i.e., single-stage indexing (e.g., B+-tree) and multi-stage indexing (e.g., LSM-tree) as shown in Figure 1, and identify their respective performance characteristics and pitfalls (*Observation 1~5*) through experiments.

2.2.1 Single-stage indexing. A single-stage indexing KVS maintains a monolithic KV-grained index to precisely record the location of each KV in the persistent data, as shown in Figure 1a. Common single-stage indexes include range indexes (e.g., B-tree variants, trie, and skiplist) and hash indexes. Considering that KVSs require support for range queries, this paper focuses only on range indexes.

Most KVSs optimized for ultra-fast storage keep the whole index in DRAM to prevent the indexing from becoming a bottleneck. For example, KVell [32] adopts a large B+-tree index and page cache in memory to ensure read performance on fast SSDs. Flatstore [9] builds an efficient multi-log structure on PM for persistent KVs and employs an existing volatile index for fast searching.

To demonstrate the impact of single-stage indexing on KVS performance, we test the read and write performances of Flatstore with different numbers of user threads under workloads of 200M writes and reads respectively. The sizes of key and value are equal and 8 bytes each (see §6.1 for detailed experimental setup). As shown in Figure 2, Masstree, which denotes Flatstore with a DRAM-only B+-tree (i.e., Masstree [38]), achieves read and write throughputs

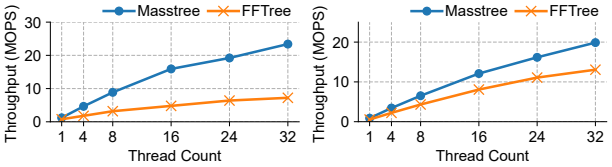


Figure 2: Scalability of single-stage indexing KVSs [*OB1*].

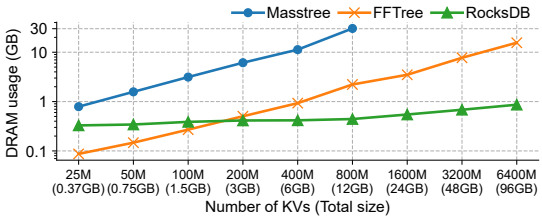


Figure 3: The DRAM consumptions of typical single- and multi-stage indexing KVSs [*OB2, OB3*].

of over 20 MOPS, demonstrating extremely high performance and parallelism of single-stage indexing. This leads to **Observation 1 (OB1): modern single-stage indexing has good concurrency and scalability in both reads and writes.**

However, in the face of ever-increasing KV data volumes, even when the PM space is sufficient, the ever-expanding index consumes a vast DRAM space. Especially for small-sized KVs, the DRAM footprint of the index may be larger than the amount of KV data itself because there are many inner nodes within the index besides the leaf nodes. Figure 3 illustrates the DRAM consumptions of Masstree as a function of dataset size. The results reveal **Observation 2 (OB2): the DRAM footprint of single-stage indexing KVS increases linearly with the dataset size at a steep slope.** When inserting about 1,600M 8+8 byte KVs (24GB in total), its index runs out of the 64GB DRAM of our hardware platform, demonstrating the dominant impact of the index DRAM footprint on KVS data capacity.

Although accommodating all or part of a KVS index in byte-addressable PM [35, 36, 41] can reduce DRAM consumption, it incurs the high cost of very noticeable performance degradation for two reasons. First, it causes I/O contention between index updates and KV data accesses. Second, small-sized accesses to PM are significantly more inefficient than to DRAM [54, 56]. As shown in Figure 2, Flatstore with Fast&Fair B+-tree [20] (denoted as FFTree), which is a persistent B+-tree on PM, underperforms its DRAM-only counterpart Masstree in write performance by 40%~70% and read performance by 35%~40%. Figure 3 also shows that FFTree trades read/write performance for reduced memory footprint, which is still proportional to data size. In summary, the performance of a given single-stage indexing KVS is dominated by its index structure, which needs to strike a careful balance between performance and DRAM footprint to accommodate an increasing data volume.

2.2.2 Multi-stage indexing. LSM-tree [40] is the most classic and representative multi-stage indexing structure in the last three decades.

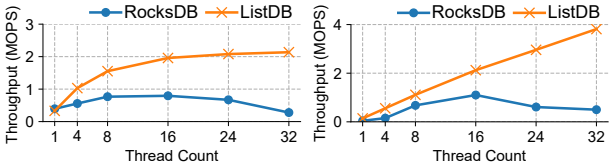


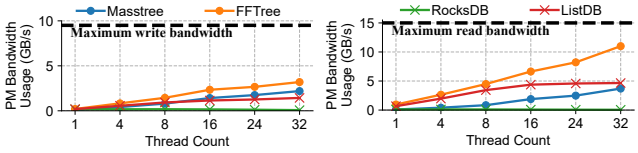
Figure 4: The performance of LSM-tree-based KVS [*OB4*].

As shown in Figure 1b, a typical LSM-tree implementation on traditional block devices, such as LevelDB [15] and RocksDB [14], comprises a small DRAM-storage hybrid stage 0 and multiple persistent stages with exponentially increasing capacities. Stage 0 consists of an in-memory index called memtable and a persistent write-ahead log (WAL) to fast persist user writes with sequential I/O. The other stages store sorted KVs with small coarse-grained persistent indexes (e.g., each index entry for 4KB block). The background threads asynchronously and periodically merge the KVs in one stage into the next stage and update the persistent indexes. We measured the DRAM footprint of RocksDB[1] at different data volumes. As shown in Figure 3, its DRAM consumption is consistently within 1GB and is primarily derived from memtable. Therefore, we conclude **Observation 3 (OB3): multi-stage indexing KVS can decouple DRAM footprint from data volume.**

The key drawback of LSM-tree, namely, the I/O (write/read) amplification, means that each KV written by the user is rewritten multiple times among stages, and a read request sequentially queries multiple stages to retrieve the target KV. Many prior works [6, 43, 55] have been proposed to alleviate I/O amplification for LSM-tree on high-performance storage devices. ListDB [30] is a state-of-the-art LSM-tree optimized for PM, replacing the sorted structure with persistent skiplists. Therefore, the background compaction can update the skiplist index by modifying pointers instead of rewriting all merged KV data to reduce data copying. We also test the performance of RocksDB and ListDB on PM. The results shown in Figure 4 uncover **Observation 4 (OB4): the read/write performance of multi-stage indexing KVS is limited.** The performances of RocksDB and ListDB are significantly lower than that of single-stage indexing KVSs such as Masstree and FFTree. Particularly, both RocksDB and ListDB exhibit limited write parallelism. One reason is the inefficiency of logging and index, e.g., the shared logging that imposes synchronization overhead and the persistent skiplist that induces small random I/Os. Another reason is the tension between write and read amplification. For example, increasing the number of stages effectively reduces write amplification (e.g., by using tiering structure[10, 11, 44]), but increases read amplification.

2.2.3 Indexes on ultra-fast storage. To understand the actual I/O bandwidth-usage of the underlying PM, we loaded 200M KVs (3 GB) to all the aforementioned KVSs. As shown in Figure 5, the KVSs use less than 1GB/s read and 0.2GB/s write bandwidth with a single thread. Even with 32 threads, the utilizations of write and read bandwidth are only up to 30% and 70% respectively. Among them, FFTree and ListDB utilize more bandwidth because they build persistent indexes on PM. Whereas Masstree uses a DRAM-only index and

[1]We use pmem-rocksdb[22], a RocksDB version adopted for PM, to run on our platform.



(a) Write bandwidth utilizations (b) Read bandwidth utilizations

Figure 5: I/O bandwidth utilizations of single- and multi-stage indexing KVSs when loading 200M KVs (3 GB) [*OB5*].

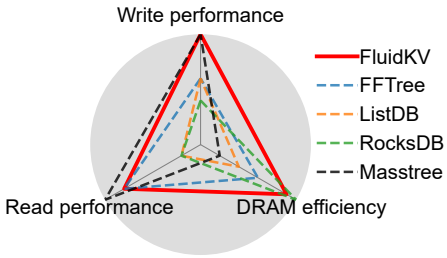


Figure 6: FluidKV design goals compared with the existing KVSs (verified and detailed in §6.2.1).

RocksDB has poor parallelism, which results in their insufficient utilization of PM bandwidth. Through the experimental results we obtain **Observation 5 (OB5): for both single-stage and multi-stage indexing, the bandwidth of PM is not a performance bottleneck.**

In summary, the performance pitfalls and differences between single-stage and multi-stage indexing KVSs lie in their architectural features. The former achieves excellent concurrency on both read and write [*OB1*] at the cost of significant DRAM consumption [*OB2*]. In contrast, the latter controls DRAM footprint by moving incoming writes through stages [*OB3*] but suffers from a low overall performance and unbalanced read/write amplification [*OB4*]. Moreover, both indexes underutilize the high PM bandwidth [*OB5*].

3 MOTIVATION AND OPPORTUNITY

Based on the above comparative analysis of the two types of KVS indexing mechanisms, we pose and attempt to answer the question of: *how to build a balanced key-value store on PM that accomplishes the three design goals of read performance, write performance and DRAM efficiency simultaneously, as shown in Figure 6?*

Goal 1 (GO1): high write scalability. Efficiently handling highly concurrent requests is critical for enhancing write throughput. This requires an efficient indexing structure that avoids performance bottlenecks due to multiple threads competing for the in-memory index and the shared WAL as discussed in §2.2.2.

Goal 2 (GO2): controllable DRAM footprint. To rationalize the use of memory space and avoid unlimited growth of DRAM footprint with the increasing data volume, a memory-efficient indexing mechanism is needed to carefully store the vast portion of the global index of KVS in PM, thus reducing the actual DRAM requirement.

Goal 3 (GO3): low read latency. The read latency of LSM-tree consists of the index querying and PM accesses on multiple stages. Because current commodity PM exhibits 5× higher read latency than DRAM [54, 56], it is important to reduce PM accesses in the critical read path of the multi-stage querying.

While it is very challenging to achieve all three goals simultaneously, our observations demonstrate two important opportunities to design a KVS with improved overall performance.

First, it is an opportunity to **combine the techniques of existing single-stage and multi-stage indexing to obtain their respective advantages.** As aforementioned, single-stage index and multi-stage index exhibit different performance advantages due to their different structures and techniques. Therefore, a KVS design that combines their advantages [*OB1, OB3*] is expected to achieve an overall performance improvement over both.

Second, with the prevalence of high-performance hardware including multi-core CPUs and fast PMs [*OB5*], it is an opportunity to **leverage hardware parallelism to amortize and reduce the overhead of consolidating the two indexing schemes.** The superior processing power of modern hardware can perform multiple stages quickly and simultaneously. Also, index structures with good concurrency are likely to make full use of hardware parallelism.

These opportunities inspire and motivate us to design a dynamic multi-stage KVS to fast and seamlessly flow incoming data from a frontend small-scale write-optimal index to a backend large-scale memory-efficient index across an intermediate bridging stage.

4 FLUIDKV DESIGN

4.1 Overview

We propose FluidKV, a dynamically balanced and parallelized key-value store, to achieve both high performance and low DRAM consumption on ultra-fast storage such as PM. FluidKV is designed as a three-stage architecture that includes a small and high-performance FastStore at the frontend, a large and memory-efficient StableStore at the backend, and a BufferStore bridging the frontend and backend stages to provide the fluidity, as shown in Figure 7.

FastStore (§4.2) adopts a KV-grained concurrent volatile range index and multiple thread-exclusive logs. FastStore is responsible for the fast processing of highly concurrent user writes and allows dynamically trading more memory for throughput under write-intensive workloads [*GO1*].

StableStore (§4.3) stores and indexes sorted key-value pairs using a set of data blocks and their index nodes on PM, respectively. The structures of the index node and data block are I/O-efficient for PM. As a result, StableStore maintains large-scale KV data and a read-optimized index with extremely low DRAM consumption [*GO2*] while guaranteeing acceptable query latency [*GO3*].

BufferStore (§4.4) is an adjustable stage between FastStore and StableStore, enabling the seamless flow and fluidity of FluidKV. It converts the KV-grained indexes of FastStore into small sorted-block-grained indexes to quickly reduce memory overhead [*GO2*] while merging these sorted block indexes into StableStore to reduce read amplification [*GO3*].

FluidKV offers standard KV interfaces such as Get, Put, and Delete, and also supports range scan, variable-sized values, and crash consistency. FluidKV prototype employs the commodity Intel Optane DCPMM as a practical PM device for the proof-of-concept

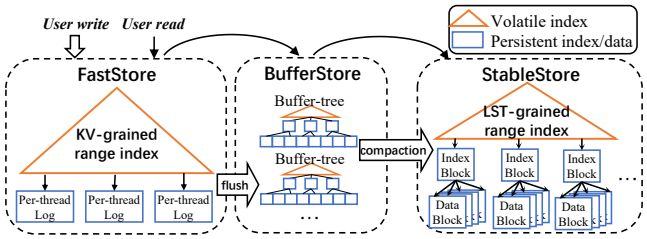


Figure 7: Architecture of FluidKV.

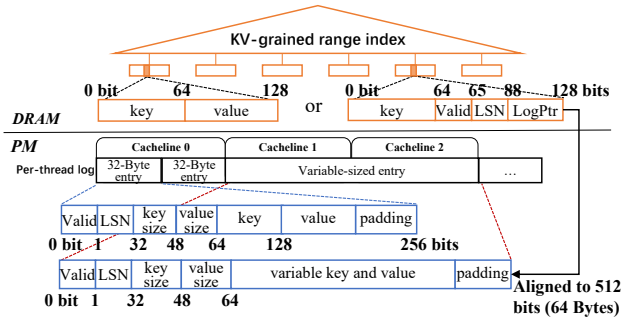


Figure 8: Data structures of the volatile index and thread-exclusive log in FastStore.

purpose, but FluidKV's design principles and key techniques can be applied to other types of PM such as ultra-fast CXL-based SSDs.

4.2 FastStore: Fast and Concurrent Writing

FastStore, as the frontend of FluidKV, is designed to fast process highly concurrent writes. It primarily uses an in-memory B+-tree index with high parallelism and employs the logging mechanism from LSM-tree, which can provide storage-friendly sequential I/O. Furthermore, FastStore uses the log structure to quickly persist user data but employs a thread-exclusive logging mechanism to reduce the log contention by exploiting parallelism.

FastStore Structure. As shown in Figure 8, FastStore stores incoming KVs into multiple thread-exclusive logs on PM and maintains a volatile index in DRAM. For small fixed-sized KV (e.g., 8+8 bytes), FastStore stores all KV data in both index and PM-logs for crash recovery. This approach is similar to LSM-tree's memtable and WAL, thus avoiding slow PM access when querying buffered KVs. For variable-sized KVs, FluidKV applies a key-value separation mechanism [37], storing a full-length KV in a PM-log while only recording its PM address as the value of the key in the in-memory index. In addition to the KV length and data, the log record also includes a 1-bit *Valid* flag to indicate whether the KV is deleted, and a 31-bit *log sequence number (LSN)* to record the written order of records for crash consistency. For PM consistency and cache locality, all records are designed to be easily aligned to the size of 64 bytes. To save PM space, when there are log records smaller than 32 bytes, multiple consecutive small records are allowed to be packed into one cacheline.

Volatile range index. Although many hybrid PM-DRAM indexes [8, 35, 41] store leaf nodes in PM to keep persistency without logging, their performances are still low because of the small random PM accesses induced. So, FastStore uses a fast DRAM-only range index, leaving the responsibility of persistence to the IO-friendly logging mechanism. Because the latency of current PM is an order of magnitude higher than that of DRAM, the DRAM-only index is unlikely to become a performance bottleneck of FastStore. Therefore, FastStore can in principle use all kinds of existing volatile range indexes. However, for performance purposes, the index must meet requirements for high concurrency and range query performance. For concurrency, indexing with fine-grained locks or optimistic concurrency control are used to avoid sudden performance degradation due to thread contention. Range query performance is important not only for standard scan operations, but also for the fact that FluidKV requires range scans of the whole index when flushing FastStore data to BufferStore (§4.4 and §5.4).

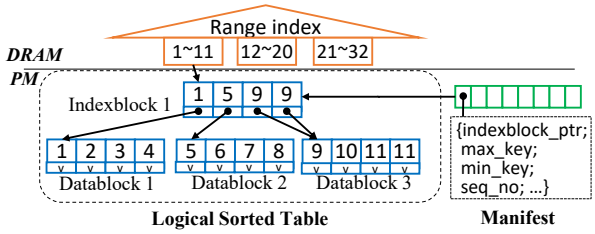


Figure 9: Data structures in StableStore.

Thread-exclusive logging. As mentioned in §2.2.2, the root cause of poor write parallelism of LSM-tree is the contention among multiple user threads for the same shared log endings. To improve the concurrency of log writes for *GO1*, we propose a thread-exclusive logging mechanism. Instead of sharing a single log, FastStore allocates an exclusive per-thread log for each user thread to mitigate I/O contention. This enables FastStore to fully utilize the I/O parallelism of PM.

Coarse-grained allocator. FluidKV employs a coarse-grained PM allocator to allocate/recycle log space for each write thread. The allocator partitions the PM space into fixed-sized chunks (e.g., 4 MB) and maintains the allocation states of the chunks with a persistent bitmap. When a thread writes new data to the PM, the allocator exclusively assigns a free PM chunk to the corresponding thread, thus avoiding write contention between multiple threads. This mechanism facilitates not only the parallel logging in FastStore, but also the PM-block writing in BufferStore and StableStore.

4.3 StableStore: Memory-Efficient Indexing

StableStore is designed to achieve a competitive read performance at a low memory footprint. StableStore is designed as a PM+DRAM hybrid structure, consisting of a volatile B+-tree and persistent sorted blocks, to index the largest portion of KVs in FluidKV. The small-scale upper-level B+-tree provides sufficient parallelism and the large-scale sorted blocks effectively reduce the memory footprint of upper-level indexes with data locality. Meanwhile, StableStore adjusts the size of persistent blocks based on I/O affinity to further optimize read performance.

Logical Sorted Table. As shown in Figure 9, the volatile B+-tree builds indexes for persistent objects named Logical Sorted Tables

Table 1: Read latency (us) of PM with different I/O sizes.

I/O size	64B	128B	256B	512B	1024B	2048B	4096B
4 PMs	0.687	0.727	0.731	0.768	0.932	1.292	1.598
6 PMs	0.655	0.688	0.695	0.741	0.897	1.256	1.529

(LST). Each LST consists of an index block and the multiple data blocks it indexes. Each data block stores a set of sorted KVs, and the key and value of an index entry in the index block are the minimum key of its indexed data blocks and the address of the data block, respectively. When key-value separation is enabled, the value for a key in the data block is a pointer to the corresponding PM-log record. Empty entries at the end of an index block are filled with the last valid entry in the block to facilitate binary search (the same goes for a data block). The maximal number of KVs stored in each LST is the product of the number of entries in an index block and the number of entries in a data block. For example, if we use 512-byte-sized blocks, i.e., one block stores 32 entries (8+8 bytes). Then the DRAM consumption of StableStore's volatile indexes is reduced to 1/1024 that of a KV-grained index like FastStore.

I/O friendly PM block. As mentioned in §2.2.1, while most persistent indexes leverage the byte-addressability of PM to perform fine-grained (e.g., 8-byte pointer) accesses, StableStore chooses 512 bytes as the size of both index and data blocks because of the I/O affinity of PM. As shown in Table 1, the random access latency of PM remains roughly the same in the I/O size range of 64B to 512B, and then rises significantly with increasing I/O size, due to the 256-byte XPBuffer inside the PM [23, 50]. Therefore, when querying the index of StableStore, at most two PM accesses with minimum read latency are required in the critical path.

In addition to the stable read latency and low memory footprint, the hybrid structure of StableStore also simplifies crash recovery. Rather than rebuilding a huge KV-grained index, StableStore only needs to recover the volatile B+-tree with LST metadata persisted in Manifest (detailed in §5.5), while the LSTs are persistently stored on the PM and do not need to be recovered.

4.4 BufferStore: Dynamic Data Migration

BufferStore between FastStore and StableStore is responsible for seamlessly fast migrating the KV data from memory-intensive FastStore into memory-efficient StableStore, enabling FluidKV to achieve the advantages of high concurrent write, stable read latency, and low memory footprint simultaneously.

In BufferStore, key-value data are indexed by multiple *buffer-trees* structurally identical to StableStore. When FastStore reaches its capacity (determined by memory constraints), the volatile index will be converted into a buffer-tree to quickly release memory resources by the *flush* operation. Then one or more buffer-trees are sorted and merged into StableStore when the number of buffer-trees reaches a certain threshold, through the *compaction* operation, which is a time-consuming process. The detailed workflows of flush and compaction are described in §5.4. The flush and compaction operations are performed concurrently and do not block the front-end processing of user requests, fully leveraging the parallelism of multi-core CPUs and ultra-fast storage to accelerate the data flowing from FastStore to StableStore.

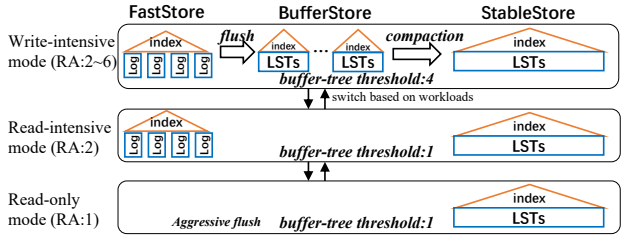


Figure 10: Load-awareness by adjusting the threshold for the number of buffer-trees. *RA denotes read amplification.*

Dynamic load-awareness. Rather than maintaining KV data distributed across multiple levels with exponentially increasing data capacity like LSM-tree, which helps reduce write amplification [*GO3*], BufferStore adopts a different data migration strategy at runtime, adapting to the workload dynamics. FluidKV calculates the read/write ratio (R:W) of the workload by profiling requests from user threads. As shown in Figure 10, under *read-intensive* workloads (e.g., R:W > 1), FluidKV triggers flush and compaction more aggressively to reduce the number of indexes and thus alleviate read amplification. Any buffer-tree in BufferStore will be opportunistically merged into StableStore via compaction (i.e., threshold=1). Moreover, under *read-only* workloads, FluidKV even ignores the threshold on the FastStore capacity to trigger flush for merging FastStore data to the later stages. On the contrary, under *write-intensive* workloads, FluidKV temporarily tolerates more buffer-trees in BufferStore to improve write performance, and triggers compaction when the number of buffer-trees reaches a high threshold (e.g., 4). This is because the write amplification of compaction is determined by the capacity ratio of BufferStore and StableStore. Therefore, enlarging the BufferStore capacity reduces write amplification at the cost of read amplification. In summary, FluidKV dynamically balances read and write performances by adjusting the relative spaces and processing capacities of its three stages.

5 IMPLEMENTATION

In this section, we provide the prototype implementation details of key operations in FluidKV. We implemented a FluidKV prototype with over 5,000 lines of original code and employed libraries such as Masstree and RocksDB thread pool.

5.1 Volatile Index

In the FluidKV prototype, we implement the volatile indexes of FastStore, BufferStore, and StableStore based on Masstree [38], a volatile B+-tree variant[2]. Our reasons for choosing Masstree are as follows. First, B+-tree is optimized for the scan operation, which facilitates iterating through all KVs of FastStore during flush operations. Second, Masstree offers a high level of concurrency and performance. Furthermore, the Masstree code is easy to modify, and its slab-based memory allocator helps us further control the DRAM usage. We reimplement the memory recycling mechanism in the destructor of Masstree to fast recycle all of its memory footprint

[2]Masstree is a trie where each node is a B+-tree, inheriting the advantages of both B+-tree and trie. In this paper, we consider it to be a B+-tree because with 8-byte keys it degrades to only one trie node, i.e., a B+-tree

Algorithm 1 Put operation (KV-separation is enabled).

```
1: function Put(k, v)
2:     lsn ← counters[hash(k) mod 256]++                    ▷ Get LSN
3:     log_ptr ← WriteLogRecord(k, v, lsn)
4:     index.PutValidate(k, lsn, log_ptr)
5:     return
6: function Masstree::PutValidate(k,lsn,log_ptr)
7:     p ← tree.FindAndLock(k) ▷ Find the position for the inserted key and
       lock the node
8:     if the target key already exists then
9:         value ← p.value
10:        if value.LSN > lsn then
11:            Unlock()
12:            return
13:    p.value ← {Valid : 1, LSN : lsn, LogPtr : log_ptr}
14:    Unlock()
15:    return
```

in batch, since FluidKV deletes the entire index frequently during flush and compaction. Also, we add a validation mechanism for consistency (see §5.2).

We also evaluated Bwtree [34], a lock-free B+-tree, and HOT [4], a concurrent trie, but did not use them. Bwtree underperforms Masstree in both read and write [51]. HOT achieves better read performance but slightly worse write and scan performance than Masstree. However, since it does not recycle memory during delete operations, it is not efficient for StableStore which frequently deletes the index entries due to compactions.

5.2 Write

Put operation. The *Put* operation is shown in Algorithm 1. First, an LSN is generated by a global incremental counter (line 2) to distinguish the global order of log records in multiple per-thread logs during crash recovery. Since sharing a single global counter causes significant synchronization overheads of concurrent user threads, each *Put* request uses one of multiple (e.g., 256) separate counters by hashing its key. Therefore, the order of the log records for the same key can still be distinguished by the LSNs. Second, FluidKV builds a log record with the key-value pair and LSN and persists it into the per-thread log (line 3). Finally, the volatile index is updated using the log record address and the LSN as value (line 4).

Because log writing and index updating are not locked, multi-threaded updates for the same key may cause the value in the index to be inconsistent with the latest log. To solve this problem without adding an inefficient coarse-grained lock, we add a validation mechanism in the write operation of Masstree. The writing process of Masstree involves first searching the target node that needs to be updated, then locking that node (line 7) and updating it (line 13), and finally unlocking it (line 14). When Masstree finds the target key in the first step, the proposed validation will read the value of the target key to get its LSN and compare it with the LSN of the record to be written. If the value is newer, the node will not be updated and will be returned (lines 8~12). This validation works for most concurrent indexes because of their similar update processes and induces no overhead when the target key does not exist in the index. Accordingly, FastStore ensures the consistency between volatile index and persistent logs while guaranteeing linearizability under high concurrency.

Algorithm 2 Get operation (KV-separation is enabled).

```
1: function Get(key)
2:     ▷ Get from active and immutable FastStores                ◁
3:     for all mem_index from FastStores do
4:         value ← mem_index.Get(key)
5:         if value ≠ NULL then
6:             if value.invalid() then
7:                 return NOT_FOUND
8:             return ReadLogForValue(key, value.log_ptr);
9:     ▷ Get from trees in BufferStore                           ◁
10:    for all mem_index ∈ trees from BufferStore do
11:        LST_id ← mem_index.scan(key, 1)          ▷ '1' is the scan size
12:        KV ← SearchKVFromLST(key, LST_id)
13:        if KV ≠ NULL then
14:            return ReadLogForValue(key, KV.value)
15:    ▷ Get from StableStore                                    ◁
16:    LST_id ← StableStore.mem_index.scan(key, 1)
17:    if KV ≠ NULL then
18:        return ReadLogForValue(key, KV.value)
19:    return NOT_FOUND
```

5.3 Read

Get operation. In a get operation (as shown in Algorithm 2), the user thread may access the active FastStore, the immutable FastStore that is being flushed, buffer-trees in BufferStore (from newest to oldest), and StableStore in order. In FastStore, the existence of the target key can be determined only by the index. In buffer-trees or StableStore, FluidKV first scans the in-memory index for an LST and then checks if the target key is in the LST. Only when the target key is found at a certain stage can the search be finished. At this point, if the found key is valid, FluidKV reads the address of its corresponding value to return the value data; if it is invalid, FluidKV notifies that the key does not exist.

Scan operation. The *Scan* operation of FluidKV is implemented in a similar way to compaction, i.e., iterating the minimum element on all stages in the target key-range with a priority queue. In the current implementation, we use LSN and *seq_no*(introduced in §5.5) as a timestamp to build a consistent snapshot for a scan. The obsolete index deleting steps of flush and compaction are postponed to keep the snapshots that are being scanned.

5.4 Flush and Compaction

The workflows of *Flush* and *Compaction* are shown in Algorithm 3 and performed with a dedicated background thread respectively. Both of them only read the read-only structures (e.g., immutable index and LSTs) from the previous stage and update the next stage, thus ensuring consistency and correctness.

Flush. To avoid contention between flush and front-end writes, FluidKV allows two FastStores simultaneously during a flush operation (immutable Flatstore for flush and active Flatstore for writes). FluidKV first creates a new active FastStore structure including volatile index and thread-exclusive logs for the subsequent writes (line 2~4). The background *Flush* thread switches FastStore by modifying a global atomic semaphore which indicates the active FastStore. The user threads check the semaphore before each write to get the index and PM logs to write. After waiting for a timeout (e.g., 100ms), the *Flush* thread scans the index of the immutable FastStore for all of its KV data to generate LSTs and builds a new corresponding buffer-tree. Finally, when the buffer-tree is ready and can be read by user threads, FluidKV deletes the index of the old FastStore

Algorithm 3 Flush and compaction operations

```
1: function FLUSH
2:     new_index ← new range index (e.g., Masstree)
3:     old_index ← FastStore.mem_index
4:     FastStore.mem_index ← new_index
5:     Wait()          ▷ Wait for user threads to finish operations on the old index
6:     ▷ Covert FastStore index into LSTs ◁
7:     Tree ← new range index (e.g., Masstree)
8:     for all k, v ∈ old_index do                          ▷ Build volatile index
9:         LSTBuilder.AddEntry(k, v)
10:        if new LST is generated then
11:            LSTMeta ← LSTBuilder.GetLST()
12:            persist LSTMeta in Manifest
13:            Tree.Put(LST.min_key, LST_Meta)
14:    Add Tree into BufferStore
15:    Delete old_index
16:    return
17: function COMPACTION
18:    ▷ Pick compaction ◁
19:    for all tree_n ∈ BufferStore do
20:        for all LSTMeta ∈ tree_n do
21:            inputs[n].add(LSTMeta)
22:    inputs[tree_num + 1] ← all LSTMeta of overlapped LSTs in StableStore
23:    ▷ Run compaction ◁
24:    Merge-sort the KVs from inputs to generate new LSTs in parallel.
25:    outputs ← all LSTMeta of new LSTs
26:    ▷ Clean compaction ◁
27:    for all LSTMeta ∈ output do
28:        Persist LSTMeta in Manifest
29:        StableStore.mem_index.put(LSTMeta.firstkey, LSTMeta)
30:    for all LSTMeta ∈ inputs.back() do
31:        Delete LSTMeta from StableStore.mem_index
32:        Free the PM space of index block and data blocks with LSTMeta
33:    Delete obsolete index trees and LSTs from BufferStore
34:    return
```

to free its occupied memory space. With the key-value separation enabled, the PM-logs will remain to store variable-length values.

Compaction. A *Compaction* operation consists of three consecutive steps, i.e., pick compaction, run compaction, and clean compaction.

In the *pick compaction* step (line 19~22), FluidKV first scans all buffer-trees to read the metadata of their associated LSTs and then scans the StableStore index to determine LSTs that have overlapped key ranges with BufferStore's LSTs.

Then in the *run compaction* step (line 24~25), FluidKV merge-sorts all input LSTs into new LSTs and builds the corresponding metadata. Specifically, in our implementation, we use a priority queue as a min-heap to perform this merge-sort. It is worth mentioning that when FluidKV finds LSTs not overlapped with other input LSTs in the merge-sorting step, the LSTs need not be changed and are inserted into StableStore to reduce write amplification [49]. Since the efficiency of compaction will be lower than flush when StableStore becomes larger, we employ a parallel compaction mechanism to fully leverage PM bandwidth and CPU cores to accelerate compactions. FluidKV partitions the input LSTs based on the key range and assigns a thread to each partition to perform merge-sorts in parallel. In our prototype, the default number of partitions is 8.

Finally, the *clean compaction* step (line 27~33) first inserts output LSTs into the StableStore index after persisting them in Manifest, then deletes the obsolete LST data and metadata from StabeStore, and finally deletes BufferStore and frees its DRAM/PM space. Because we use a concurrent B+-tree (e.g., Masstree) as the StableStore index, the index update process also does not require locking.

5.5 Crash Recovery and Consistency

As mentioned in §4.2 and 5.2, the volatile index of FastStore can be recovered with thread-exclusive logs. Because each log record has a globally unique LSN, FluidKV first merges all the logs in the LSN order and then replays them during the recovery. For the crash recovery of volatile indexes in the last two stages, FluidKV stores the metadata of LSTs in a separate PM space called Manifest (LST metadata of BufferStore and StableStore are stored separately). Therefore, the corresponding volatile tree indexes are reconstructed according to the metadata in Manifest. As shown in Figure 9, the metadata of an LST contains the following main fields: *indexblock_ptr* points to the index block; *min_key* and *max_key* indicate the key range; *seq_no* is a self-incrementing version number which is used to implement a basic Multi-Version Concurrency Control (MVCC) for crash consistency. Specifically, in BufferStore, LSTs of the same buffer-tree share the same *seq_no*. FluidKV persists the valid *seq_no* ranges in the manifest (updated after each flush and compaction). During crash recovery, the outdated or overrun LSTs can be cleaned up based on the *seq_no*.

During a flush, all associated per-thread logs need to be deleted atomically for otherwise it will cause inconsistencies during crashes (only appear when KV separation is disabled). To solve this problem, FluidKV records the addresses of associated log chunks on PM before log recycling and clears them at the end of flush. During recovery, the PM chunks need to be recycled again based on the persisted addresses.

Although flush or compaction needs to change a lot of metadata of LSTs, any temporary intermediate state can only inroduce redundant data in two stages without the risk of missing data, because the operations always write new data before deleting obsolete data. Thus, read operations can be efficiently concurrent with background operations while ensuring consistency.

6 EVALUATION

6.1 Experiment Setup

Test platform. All experiments are conducted on a Linux 5.1.0 machine with an Intel Xeon Gold 5218 CPU (32 cores, 2.3GHz) and 64GB DDR4 memory. The experiments are performed on 6×128GB Intel Optane DC PM 100s configured in AppDirect Mode. We use our modified version of PiBench [33] to test and statistically measure the performance of various KVSs. PiBench is a benchmarking framework that targets PM-based indexes and has been widely used in prior studies [17–19]. We extended it to support more KVSs.

Baselines. We compared FluidKV against state-of-the-art PM-aware KVSs including Fast&Fair B+-tree (*FFTree*) [20], *PacTree* [31], *NBTree* [63], *LB+-Tree* [35], *DPTree* [65] and *ListDB* [30]. Since the baselines. with the exception of ListDB, are only indexes rather than full-feature KVSs (only support 8+8 byte items), we integrate the indexes with Flatstore[9] to support full KVS functionalities such as variable-length values and recovery by storing variable-sized values in PM-logs. Flatstore enhances the overall performance with parallel PM-logs and I/O batching mechanisms so that it does not become a bottleneck in the KVSs. For reference, we also evaluate Flatstore with *Masstree* [38], serving as a performance upper bound of PM-aware KVS by using a DRAM-only index instead of a persistent index. FFTree, NBTree, and LB+-tree are persistent B+-Tree

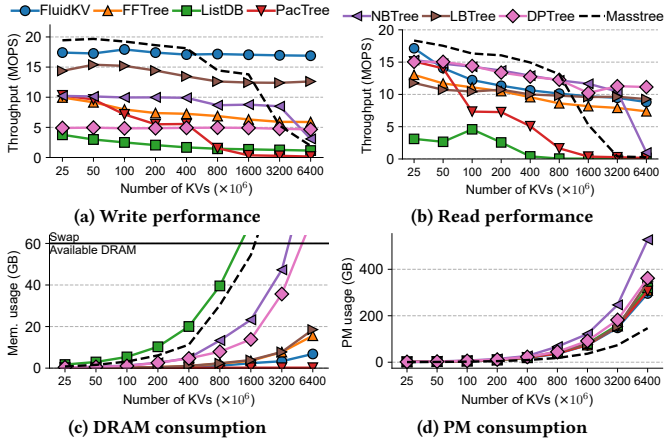


Figure 11: KVS performance with varying dataset sizes.

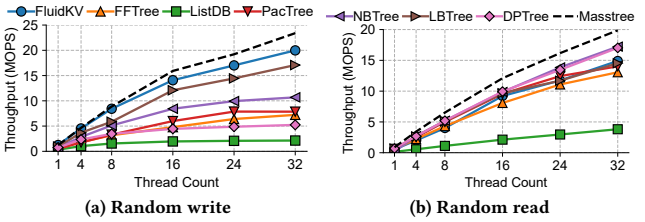


Figure 12: Write/read throughput scalability (200M dataset).

variants. PacTree is based on a persistent radix tree and B+-tree-like linked leaf nodes. DPTree and ListDB are both based on multi-stage indexes. DPTree uses volatile B+-trees as the index of the first two stages and a volatile trie with persistent leaf nodes as the last stage index. ListDB is composed of persistent skiplists in each stage.

Workloads and FluidKV configurations. To focus on the indexing mechanism rather than logging, all workloads use 8+8 byte KV size unless mentioned otherwise. To be fair with the baselines that use Flatstore, FluidKV enables key-value separation so that all requests also need to read/write the PM log. By default, the capacity of FastStore is of 40M records and the threshold for the number of buffer-trees to trigger compaction is 4, but it can be adjusted to 1 dynamically under read-heavy workloads.

6.2 Micro-Benchmarks

6.2.1 Data Volume Scalability. To verify that FluidKV achieves our three performance goals as mentioned in § 3, we perform 10M read and write operations with 24 parallel user threads after loading different sizes of datasets. We measure the request throughputs and DRAM/PM consumptions to evaluate the performance of different KVSs in terms of write, read, and memory efficiency.

Write. As shown in Figure 11a, FluidKV exhibits a consistently high write performance under various data volumes, just slightly lower than the ideal DRAM-only Masstree at small/middle-scale datasets. For large-scale datasets, FluidKV outperforms all KVSs, 30% higher than the second highest, LBTree. This is because FastStore can quickly persist KVs and the asynchronous compaction and flush operations do not affect front-end writes. Whereas, the single-stage indexing KVSs suffer from significant performance degradation as the size of the index increases. The performance of Masstree and NBTree drops sharply when the data volume exceeds 800M and 3200M respectively, because they use up all the DRAM and start using swap space.

Read. Figure 11b shows that FluidKV's read performance is still competitive with other single-stage indexing KVSs. ListDB and PacTree have particularly significant performance degradation at large data volumes. In multi-stage indexing KVSs, ListDB performs worst due to read amplification and low parallelism of skiplists, while DPTree performs slightly better than FluidKV by up to 20%. This is because DPTree tends to buffer KVs in the first few stages, making full use of the DRAM-only indexes, rather than merging data into the persistent last stage proactively as FluidKV. Therefore, its read performance is close to that of Masstree under small-scale datasets but its write performance is only 25% of FluidKV due to the write stalls during migrations.

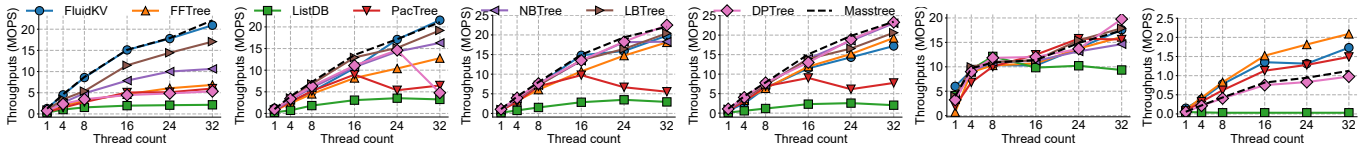
DRAM consumptions. Figure 11c illustrates that Masstree, ListDB, NBTree, and DPTree have much larger memory footprints than other baselines and finally run out of DRAM when data volume increases. The reasons for their huge DRAM consumption are different. Masstree and DPTree use large DRAM to store the KV-grained index. NBTree stores inner nodes and metadata of leaf nodes on DRAM. ListDB uses a lazy memory recycling technique that cannot release memory immediately after a flush. The DRAM consumptions of the remaining KVSs are acceptable, among which FluidKV is the second lowest, less than 10% of the data volume. LBTree and FFTree as hybrid indexes have a DRAM footprint of about 15% of the data volume, while PacTree as a PM-only index utilizes almost no memory.

PM consumptions. As shown in Figure 11d, FluidKV and most baselines have similar PM consumptions, about 2-3× of the data amount. This is because both the PM-logs and the persistent index contain all the KV records (key-value in log and key-pointer in index) and several additional fields such as record lengths. Masstree without persistent index cuts PM consumptions in half compared to other KVSs. NBTree consumes 30% more PM space with large data volumes possibly due to its inefficient PM-space recycling.

6.2.2 Parallel Scalability. To compare the performance scalability of FluidKV with the baselines with sufficient DRAM, we perform 200M random writes and reads on the KVSs respectively with a varying number of user threads.

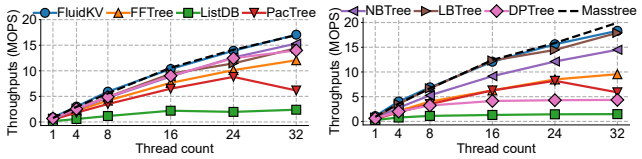
As shown in Figure 12a, FluidKV achieves the best write scalability among the KVSs using persistent indexes. With 32 threads, the write throughput of FluidKV is 1.2×~9× that of the baselines. This manifests that FluidKV's FastStore absorbs write-intensive workloads with highly parallelized index and thread-exclusive logging.

Figure 12b shows that the read throughput of FluidKV is similar to most single-stage KVSs and up to 3.9× higher than multi-stage ListDB. Although the multi-stage design causes more overhead querying multiple stages, FluidKV can dynamically reduce the number of stages in FastStore and BufferStore by performing aggressive flush and compaction under read-intensive workloads. Overall, FluidKV's read performance, while 10% lower than NBTree and DPTree,



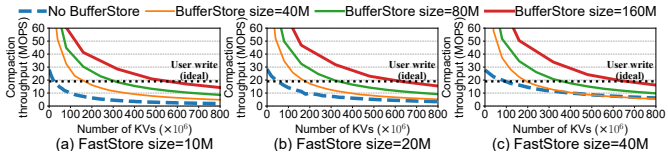
(a) YCSB Load (100% write, uniform) (b) YCSB A (50% write, 50% read, Zipfian) (c) YCSB B (5% write, 95% read, Zipfian) (d) YCSB C (100% read, Zipfian) (e) YCSB D (5% write, 95% read, latest) (f) YCSB E (5% write, 95% scan, Zipfian)

Figure 13: Performance under YCSB workloads. The proportion and distribution of the workloads are shown in parentheses. The skewness factor of the Zipfian distribution is 0.99.



(a) Cluster 27 (15% write, 85% read) (b) Cluster 31 (94% write, 6% read)

Figure 14: Performance under Twitter cluster workloads.



(a) FastStore size=10M (b) FastStore size=20M (c) FastStore size=40M

Figure 15: Compaction efficiency with different FastStore and BufferStore configurations.

is 1.9× and 3.8× higher than their write performance, respectively, demonstrating a good trade-off between read and write.

6.3 Macro-Benchmarks

We evaluate the performance of each KVS under synthetic workloads generated by YCSB as shown in Figure 13. In each group of experiments, we load 200M KVs and perform 10M operations.

Under **write-intensive workloads (Load and A)**, as illustrated in Figure 13a and 13b, FluidKV outperforms all baselines in throughput significantly, 1.3x~9.5x under Load and 1.1x~6.6x under A with 32 threads. This shows that highly concurrent indexing and parallelized logging in FluidKV's FastStore are highly capable of handling write requests.

Under **read-intensive workloads (B and C)**, as shown in Figure 13c and 13d, FluidKV, as a multi-stage indexing KVS, demonstrates comparable performance (0.8~2.2×) to the single-stage baselines and superior performance (up to 8×) over the multi-stage ListDB, because of the read-optimized design of StableStore and the dynamic load balancing mechanism to reduce the indexes across its stages under read workloads.

Under **hot-data search workload D**, as shown in Figure 13e, multi-stage FluidKV, DPTree, and ListDB can keep the latest-written hot data in the top-most stage with KV-grained index, thus obtaining performance as good as the single-stage designs. Among the multi-stage KVSs, the performance of FluidKV, using the faster Masstree index in FastStore, is significantly higher than that of ListDB with limited parallelism. By caching more latest data in DRAM at the expense of write performance, DPTree achieves 12% higher read performance than FluidKV and even outperforms Masstree which indexes all data with a DRAM-only index.

Under **scan workload E.**, Figure 13f shows the scan performance of FluidKV and the baselines except for ListDB and NBTree without support for scan operation. Note that Masstree does not represent the ideal performance under workload E since its scan implementation is suboptimal. Because of the multi-stage designs, FluidKV needs to search on all stages to perform a scan operation, so its performance is lower than that of single-stage FFTree. However, FluidKV's scan performance is still acceptable and scalable.

We also measure the throughput on two realistic **Twitter cluster workloads**[58] with different read-write ratios. As Figure 14 shows, with 32 threads, FluidKV outperforms the baselines by 1.1×~7× and 1.05×~12× for the read-heavy and write-heavy workloads, respectively. These results are largely consistent with the YCSB results.

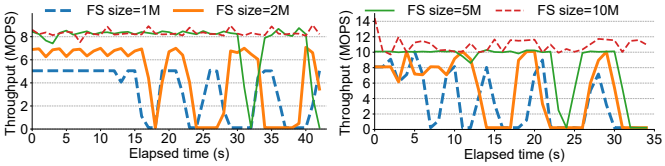
6.4 Recovery

We evaluate the recovery time of FluidKV after loading 20M and 200M KVs with a single thread respectively. It takes 1.5 and 1.8 seconds, respectively, to scan the persistent logs and Manifest data, and rebuild the volatile indexes in the three stages. More than 80% of the recovery time comes from FastStore due to the KV-grained volatile index. Because FastStore has a typically low capacity limit, the recovery time of FluidKV is less affected by the amount of data. For reference, ListDB takes 1.8 seconds to recover 200M KVs, while PacTree's recovery time is less than 0.5 seconds because it is a PM-only index.

6.5 Sensitivity Study

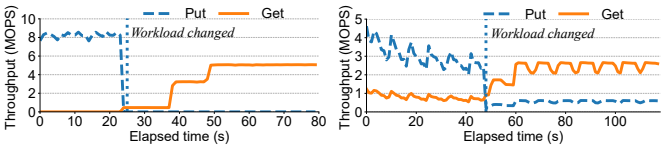
6.5.1 FluidKV trade-offs. FluidKV ensures read/write/DRAM efficiency simultaneously with multi-stage indexing and fast data migration. While trade-offs among these three dimensions remain, they have shifted from user requests to background flush and compaction operations. When the background merging (*write*) is slower than the user writes, the *DRAM* footprint continuously increases because of the increase in FastStore size and the number of buffer-trees in BufferStore, thus leading to high *read* amplification. Therefore, the size of FastStore and BufferStore is the key to trading off read and memory efficiency for write performance.

Figure 15 shows how FluidKV adjusts this trade-off. Note that since the throughput of flush is higher than user writes (stable at 20 MOPS), we only show the impact of compaction. First, the efficiency of compaction decreases with the increase in data volume because the write amplification heightens as StableStore enlarges.



(a) 8 user threads (b) 16 user threads

Figure 16: FluidKV write capability with limited FastStore capacities (1M, 2M, 5M, and 10M KVs, respectively).

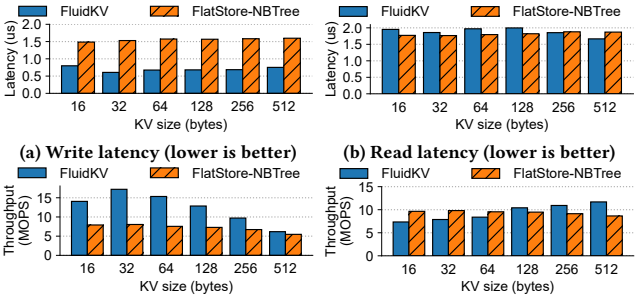


(a) W:R changes from 1:0 to 0:1 (b) W:R changes from 4:1 to 1:4

Figure 17: FluidKV performance through workload changes.

Table 2: Random read latency (us) of StableStore.

Data block \ Index block	256B	512B	1024B
256B	1.254	1.246	1.352
512B	1.217	1.227	1.294
1024B	1.327	1.346	1.302



(a) Write latency (lower is better) (b) Read latency (lower is better)

(c) Write throughput (higher is better) (d) Read throughput (higher is better)

Figure 18: Write/read performance with different KV sizes.

When the compaction throughput falls below the throughput of user writes (e.g., when data volume reaches 700M), it is hard to maintain the long-term stability of an ideal write performance under intensive workloads. Second, with the same FastStore size, doubling the BufferStore size can approximately double the compaction throughput, keeping the ideal write performance under write-intensive workloads for a longer time, but leading to higher read amplification due to more buffer-trees in BufferStore. Fortunately, the read amplification can be compensated by increasing the capacity limit of FastStore at the expense of a corresponding memory consumption. Similar results in all three figures show that the FastStore size barely affects the compaction efficiency. Therefore, we recommend configuring a larger FastStore to trade off between read and write performances when DRAM is sufficient, or dynamically increase the capacity of FastStore and BufferStore with the increasing data volume.

In addition, we evaluate the efficiency of compacting data directly from FastStore to StableStore without BufferStore. The results show that the configuration without BufferStore is the worst in all configurations due to its hight write amplification.

DRAM/write trade-off: FastStore size. We evaluate the write throughput of FluidKV under write workloads with a limited capacity of FastStore. The number of buffer-trees is fixed at 4 to ensure an almost constant read performance. As shown in Figure 16, FluidKV maintains stable high write throughputs for 16, 18, and 32 seconds (under 8 threads), 7, 14, and 24 seconds (under 16 threads) at FastStore capacities of 1M, 2M, and 5M KVs, respectively. The performance is more stable when FastStore size is larger. In the figure, the slight performance fluctuations come from the flush operations while the larger dips are due to BufferStore being full. Because the capacity of BufferStore also depends on the capacity of FastStore (i.e., the size of buffer-tree), a larger FastStore provides better buffering for stable performance. Note that under high-intensity writes, even though the throughput fluctuates sometimes, it still returns to a normal performance level after the flush and compaction operations are completed. These results indicate that under workloads with fixed intensity or limited burst time, suitable FastStore size can achieve a sensible balance between stable write performance and reasonable memory footprint.

Write/read trade-off: dynamic stage-merging. Figure 17 demonstrates FluidKV's ability to be aware of and thus able to dynamically adjust according to the workload (8 user threads are used). As shown in Figure 17a, we executed 200M write requests and immediately converted the workload to read-only. When FluidKV detects the workload change, it accelerates stage merging by aggressively triggering flush and compaction to improve read performance. Under a mixed workload (80% write, 20% read) in Figure 17b, the initial buffer-tree threshold to trigger compaction is 4 to accommodate the major write requests. When the proportion of read requests becomes higher (20% write, 80% read), FluidKV adjusts the threshold to 1, reducing read amplification to better serve the read requests. These dynamic threshold adjustments flexibly trade off between read and write performances according to workload characteristics.

Read/DRAM trade-off: LST block size. To validate the efficiency of the 512-byte PM block size, we load 100M 8+8 byte KVs into StableStores with different sizes of index and data block respectively, and evaluate the random read latency of StableStore as shown in Table 2. For reference, the read latency of Masstree is 1.1us. The results show that the LST with 512-byte index block and 512-byte data block, which are the default configurations of FluidKV, achieves a good trade-off, trading an 11% read latency penalty for a 1024× (mentioned in §4.3) DRAM footprint reduction.

6.5.2 Different KV size. Although variable-sized KV is not our focus, we also evaluate the single-thread latencies and 16-thread throughputs of FluidKV and NBTree under workloads with different KV sizes (8-byte key with variable-sized value). The dataset size for all workloads is 3 GB. Note that variable-sized KVs are supported by the Flatstore implementation and irrelevant to the indexes, and the indexes on Flatstore behave similarly to NBTree. We do not evaluate ListDB that does not support variable-sized KVs.

As shown in Figure 18, due to the high bandwidth of PM, the latencies of both KVSs are slightly impacted by the KV length. FluidKV exhibits similar read latency to and 50%~60% less write latency than Flatstore. The write throughputs of FluidKV and Flatstore decrease with increasing KV size because of the larger PM I/Os. FluidKV's write throughput with 16-byte KVs is less than that with 32-byte KVs. This is because 16-byte KV causes 32-byte logging I/O, which is PM-unfriendly due to the misalignment with cacheline. Flatstore as a single-stage indexing KVS has a stable read throughput. In contrast, since there are fewer KVs under the same size dataset when KV size increases, more reads hit FluidKV FastStore thus improving the read throughput. In summary. the results indicate that FluidKV's optimizations remain effective for large-sized KVs.

7 DISCUSSION AND FUTURE WORK

Transaction support is also an important feature required for KVSs [47]. Although FluidKV implements a basic MVCC to ensure concurrency and crash consistency of read/write/flush/compaction operations, it needs more modifications to support transactions. For example, we need to use a globally consistent timestamp as the LSN of the log record and give the same LSN to log records from the same transaction. Also, the uncommitted records should not be flushed to BufferStore to avoid losing the LSN. Note that transaction support does not affect the fairness of evaluation because the baselines also do not support transactions.

CXL-based SSD. Compared to Intel Optane PM which was discontinued in 2022, the latest CXL-based memory-semantic SSDs [25, 60] have higher bandwidth and capacity at a lower cost. As the storage capacity increases, DRAM-only indexes incur a larger DRAM footprint for larger datasets. In contrast, the hybrid-index architecture of FluidKV effectively constrains the DRAM footprint without significant performance degradation. Because the CXL protocol is based on PCIe with a longer I/O path than Optane, our optimizations aimed at reducing PM accesses will become potentially more important for and beneficial to the CXL-based SSDs. Even so, FluidKV still needs to be adjusted and optimized for the new devices. For example, considering the characteristics of flash media, the sizes of the index and data block need to be increased (e.g., 4 KB). Also, storing large KV pairs in the data blocks instead of using KV separation also helps reduce random small reads, which are inefficient on the flash devices.

8 RELATED WORK

Single-stage indexing KVS for PM. As mentioned in §2.2.1, single-stage indexing KVS with DRAM-only indexes such as Flatstore [9] and KVell [32], aim for extreme performance at the expense of a huge memory footprint. Viper [3] and Halo [18] both use volatile hash indexes to achieve higher performance, but sacrifice range query functionality. Prism [46] builds KVS on PM with PacTree and employs caches on DRAM to accelerate read operations. While the performance and memory footprint can be statically balanced by using PM-only [20, 31, 39] or hybrid indexes [35, 65], FluidKV can achieve a dynamic balance to cope with increasing data volumes.

Multi-stage indexing KVS for PM. SLM-DB [26], NoveLSM [28], and MatrixKV [61] build additional indexes or buffers on PM to accelerate SSD-based LSM-tree. ChameleonDB [64] builds LSM-tree in PM with PM-friendly I/Os. Different from FluidKV, ChameleonDB uses hash-based sharding which deprives it of the range query capability and builds in-memory auxiliary indexes for data on PM to reduce I/O amplification, which causes a high memory footprint. ListDB [30] uses persistent skiplists to build LSM-tree on PM and employs NUMA-aware optimizations to improve scalability across multiple NUMA nodes, but is limited by its complex indexing on PM. MioDB [13] also employs persistent skiplists for good tail latency and does not achieve a good write throughput scalability, limited by the LSM-tree structure. FluidKV proposes a StableStore structure that is more suitable for PM and a highly concurrent FastStore with optimizations for parallelism to achieve higher scalability and flexibility, achieving significant performance improvement beyond traditional LSM-tree.

Dynamic index transition. Monkey [10] and Dostoevsky [11] explore the dynamic tuning for read-write trade-offs of traditional LSM-tree, e.g., switching between tiering and leveling structures. Idreos et al. [21] analyze the performance characteristics of B+-tree and LSM-tree, and show the potential of transitions between the indexes with a theoretical model. FluidKV is the first to design and implement a practical system to combine the benefits of single-stage and multi-stage indexes through fast stage-merging by utilizing ultra-fast storage.

Key-value separation. WiscKey [37] first proposes key-value separation to solve the high write-amplification problem caused by large values in the LSM-tree. HashKV [7] and NovKV [45] further solve the garbage collection problem of key-value separation by methods such as hot-cold separation. Pacman [48] optimizes the efficiency of garbage collection on PM through techniques such as reverse indexing. FluidKV also uses the key-value separation to store long KVs. Because garbage collection optimization is not our focus, FluidKV leverages these existing techniques to implement and optimize garbage collection.

9 CONCLUSION

FluidKV proposes a new multi-stage KVS architecture for ultra-fast storage, which achieves high performance and low memory footprint by exploiting the high processing capabilities and parallelism of modern computer hardware efficiently. It dynamically and seamlessly migrates data across three stages, including a parallelized FastStore for fast persistence, a transitional BufferStore to control memory footprint in time, and a StableStore providing memory-efficient indexing. Our evaluation shows that FluidKV achieves higher performance while maintaining a lower memory footprint than the state-of-the-art PM-aware KVSs. We believe that FluidKV's design principles and key techniques can be applied to other ultra-fast storage devices such as CXL-based SSDs.

ACKNOWLEDGMENTS

This work was supported by NSFC No.62172175 and No.61821003, Key Research and Development Project of Hubei No.2022BAA042, and the US National Science Foundation grant CNS-2008835 and CCF-2226117.

REFERENCES

[1] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna T. Malladi, and Yang-Seok Ki. 2022. Enabling CXL Memory Expansion for In-Memory Database Management Systems. In *International Conference on Management of Data, DaMoN 2022, Philadelphia, PA, USA, 13 June 2022*. 8:1–8:5.

[2] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, Alexander Driskill-Smith, and Mohamad Krounbi. 2013. Spin-transfer torque magnetic random access memory (STT-MRAM). *ACM J. Emerg. Technol. Comput. Syst.* 9, 2 (2013), 13:1–13:35.

[3] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. 2021. Viper: An Efficient Hybrid PMem-DRAM Key-Value Store. *Proc. VLDB Endow.* 14, 9 (2021), 1544–1556.

[4] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. 521–534.

[5] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA, 209–223.

[6] Yunpeng Chai, Yanfeng Chai, Xin Wang, Haocheng Wei, Ning Bao, and Yushi Liang. 2019. LDC: A Lower-Level Driven Compaction Method to Optimize SSD-Oriented Key-Value Stores. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. 722–733.

[7] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. 2018. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. 1007–1019.

[8] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. 2020. μTree: a Persistent B+-Tree with Low Tail Latency. *Proc. VLDB Endow.* 13, 11 (2020), 2634–2648.

[9] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*. 1077–1091.

[10] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. 79–94.

[11] Niv Dayan and Stratos Idreos. 2018. 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 Conference 2018, Houston, TX, USA, June 10-15, 2018*. 505–520.

[12] Zhuohui Duan, Jiabo Yao, Haikun Liu, Xiaofei Liao, Hai Jin, and Yu Zhang. 2023. Revisiting Log-Structured Merging for KV Stores in Hybrid Memory Systems. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. 674–687.

[13] Zhuohui Duan, Jiabo Yao, Haikun Liu, Xiaofei Liao, Hai Jin, and Yu Zhang. 2023. Revisiting Log-Structured Merging for KV Stores in Hybrid Memory Systems. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. 674–687.

[14] Facebook. 2022. RocksDB. https://rocksdb.org/ (last aceessed at 2-17-2024).

[15] Google. 2014. LevelDB. https://github.com/google/leveldb (last aceessed at 2-17-2024).

[16] Shukai Han, Dejun Jiang, and Jin Xiong. 2020. SplitKV: Splitting IO Paths for Different Sized Key-Value Items with Advanced Storage Devices. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*.

[17] Yuliang He, Duo Lu, Kaisong Huang, and Tianzheng Wang. 2022. Evaluating Persistent Memory Range Indexes: Part Two. *Proc. VLDB Endow.* 15, 11 (2022), 2477–2490.

[18] Daokun Hu, Zhiwen Chen, Wenkui Che, Jianhua Sun, and Hao Chen. 2022. Halo: A Hybrid PMem-DRAM Persistent Hash Index with Fast Recovery. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. 1049–1063.

[19] Daokun Hu, Zhiwen Chen, Jianbing Wu, Jianhua Sun, and Hao Chen. 2021. Persistent Memory Hash Indexes: An Experimental Evaluation. *Proc. VLDB Endow.* 14, 5 (2021), 785–798.

[20] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies, FAST 2018, Oakland, CA, USA, February 12-15, 2018*. 187–200.

[21] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, and Zichen Zhu. 2019. Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*.

[22] Intel. 2021. Pmem-RocksDB. https://github.com/pmem/pmem-rocksdb (last aceessed at 2-17-2024).

[23] Intel. 2022. Intel® Optane™ Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[24] Brian Myungjune Jung. 2022. Controller Design Considerations for Samsung's Memory-Semantic SSD. Flash Memory Summit 2022.

[25] Myoungsoo Jung. 2022. Hello bytes, bye blocks: PCIe storage meets compute express link for memory expansion (CXL-SSD). In *HotStorage '22: 14th ACM Workshop on Hot Topics in Storage and File Systems, Virtual Event, June 27 - 28, 2022*. 45–51.

[26] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA, 191–205.

[27] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA, 993–1005.

[28] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA, 993–1005.

[29] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald G. Dreslinski. 2021. Improving Performance of Flash Based Key-Value Stores Using Storage Class Memory as a Volatile Memory Extension. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*. 821–837.

[30] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young ri Choi, Alan Sussman, and Beomseok Nam. 2022. ListDB: Union of Write-Ahead Logs and Persistent SkipLists for Incremental Checkpointing on Persistent Memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA, 161–177.

[31] Wook-Hee Kim, Madhava Krishnan Ramanathan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*. 424–439.

[32] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. 447–461.

[33] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating Persistent Memory Range Indexes. *Proc. VLDB Endow.* 13, 4 (2019), 574–587.

[34] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. 302–313.

[35] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+-Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *Proc. VLDB Endow.* 13, 7 (2020), 1078–1090.

[36] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *Proc. VLDB Endow.* 13, 8 (2020), 1147–1161.

[37] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA, 133–148.

[38] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*. 183–196.

[39] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*. 31–44.

[40] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385.

[41] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. 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 Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. 371–386.

[42] Rekha Pitchumani. 2022. Next-Gen System Architectures with Memory-Semantic SSDs. Flash Memory Summit 2022.

[43] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles,*

Shanghai, China, October 28-31, 2017. 497–514.

[44] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2021. Constructing and Analyzing the LSM Compaction Design Space. *Proc. VLDB Endow.* 14, 11 (2021), 2216–2229.

[45] Chen Shen, Youyou Lu, Fei Li, Weidong Liu, and Jiwu Shu. 2020. NovKV: Efficient Garbage Collection for Key-Value Separated LSM-Stores. In *36th Symposium on Mass Storage Systems and Technologies, MSST 2020, Santa Clara, CA, USA, Oct 29-30, 2020*. 38–50.

[46] Yongju Song, Wook-Hee Kim, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. Prism: Optimizing Key-Value Store for Modern Heterogeneous Storage Devices. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. 588–602.

[47] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. Cobra: Making Transactional Key-Value Stores Verifiably Serializable. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 63–80.

[48] Jing Wang, Youyou Lu, Qing Wang, Minhui Xie, Keji Huang, and Jiwu Shu. 2022. Pacman: An Efficient Compaction Approach for Log-Structured Key-Value Store on Persistent Memory. In *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*. 773–788.

[49] Xiaoliang Wang, Peiquan Jin, Bei Hua, Hai Long, and Wei Huang. 2022. Reducing Write Amplification of LSM-Tree with Block-Grained Compaction. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. 3119–3131.

[50] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2021. Characterizing and Modeling Nonvolatile Memory Systems. *IEEE Micro* 41, 3 (2021), 63–70.

[51] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. 473–488.

[52] H.-S. Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P. Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E. Goodson. 2010. Phase Change Memory. *Proc. IEEE* 98, 12 (2010), 2201–2227.

[53] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*. 349–362.

[54] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. 2022. Characterizing the performance of intel optane persistent memory: a close look at its on-DIMM buffering. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*. 488–505.

[55] Baoyue Yan, Xuntao Cheng, Bo Jiang, Shibin Chen, Canfang Shang, Jianying Wang, Kenry Huang, Xinjun Yang, Wei Cao, and Feifei Li. 2021. Revisiting the Design of LSM-tree Based OLTP Storage Engine with Persistent Memory. *Proc. VLDB Endow.* 14, 10 (2021), 1872–1885.

[56] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*. 169–182.

[57] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*. 167–181.

[58] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. 191–208.

[59] J. Joshua Yang and R. Stanley Williams. 2013. Memristive devices in computing system: Promises and challenges. *ACM J. Emerg. Technol. Comput. Syst.* 9, 2 (2013), 11:1–11:20.

[60] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhyung Park, Jin yong Choi, Eyee Hyun Nam, Eunji Lee, Sungjin Lee, and Bryan S. Kim. 2023. Overcoming the Memory Wall with CXL-Enabled SSDs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA, 601–617.

[61] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. 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)*. 17–31.

[62] Jinghuan Yu, Sam H. Noh, Young ri Choi, and Chun Jason Xue. 2023. ADOC: Automatically Harmonizing Dataflow Between Components in Log-Structured Key-Value Stores for Improved Performance. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. Santa Clara, CA, 65–80.

[63] Bowen Zhang, Shengan Zheng, Zhenlin Qi, and Linpeng Huang. 2022. NBTree: a Lock-free PM-friendly Persistent B+-Tree for eADR-enabled PM Systems. *Proc. VLDB Endow.* 15, 6 (2022), 1187–1200.

[64] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. 2021. ChameleonDB: a key-value store for optane persistent memory. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*. 194–209.

[65] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. 2019. DPTree: Differential Indexing for Persistent Memory. *Proc. VLDB Endow.* 13, 4 (2019), 421–434.