Co-design of B+-tree Index with Emerging Zone Interfaces for Small-sized Key-value Pairs

JINLEI HU, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
BO CHEN, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
HONG JIANG, The University of Texas at Arlington, Arlington, United States
MIAOSONG ZHANG, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
JING HU, Wuhan National Laboratory of Optoelectronics, Huazhong University of Science and Technology, Wuhan, China
JIANXI CHEN*, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
DAN FENG*, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China

The host-side and append-only zone interface offers new opportunities for existing key-value stores (KVSs), especially in reducing the flash-layer write amplification. While existing works focused on leveraging the zone interface for LSM-Tree-based KVSs, the potential advantages for B^+-Tree-based KVSs remain under-explored. Through in-depth experimental observations, we identify three key opportunities for B^+-Tree-based KVSs: superior read performance, flash-friendly append operations, and the ability to directly manage flash media. However, existing B^+-Tree-based KVSs designed for block SSDs rely on inefficient file system layers to adapt to the idiosyncratic zone interface, and achieve reduced write amplification by seriously sacrificing read performance. We proposed a write-optimized B^+-Tree-based key-value store ZKV, designed to minimize write amplification through the zone interface without sacrificing read performance. ZKV employs a three-level buffer structure to enable its core index structure Z^+-Tree to be zone-interface efficient, including a tree-level merged filter, a leaf-level adaptive delta buffer, and an efficient zone-level management module ZFlusher. Through the three-level buffer structure, Z^+-Tree effectively leverages the zone interface by converting small-size random write operations into flash-friendly large-block append operations. Our evaluations on a real ZNS SSD device demonstrate that ZKV achieves up to 3.12x/2.53x higher insert/read throughput than the current buffered-B^+-Tree-based KVSs under YCSB workloads. Under three additional

*Corresponding authors:Dan Feng; Jianxi Chen.

Authors' Contact Information: Jinlei Hu, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei, China; e-mail: hujinlei@hust.edu.cn; Bo Chen, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China; e-mail: BrianChan@hust.edu.cn; Hong Jiang, The University of Texas at Arlington, Arlington, Texas, United States; e-mail: hong.jiang@uta.edu; Miaosong Zhang, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei, China; e-mail: zhangms@hust.edu.cn; Jing Hu, Wuhan National Laboratory of Optoelectronics, Huazhong University of Science and Technology, Wuhan, HuBei, China; e-mail: hujingreal@hust.edu.cn; Jianxi Chen, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei, China; e-mail: chenjx@hust.edu.cn; Dan Feng, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei, China; e-mail: dfeng@hust.edu.cn.

ACM 1544-3973/2025/11-ART
https://doi.org/10.1145/3778171

real-world workloads, ZKV matches the write performance of LSM-Tree-based RocksDB while delivering the read performance of B$^+$-Tree-based WiredTiger.

CCS Concepts: • **Hardware → Emerging interfaces**; • **Information systems → Key-value stores**; **Flash memory**.

Additional Key Words and Phrases: ZNS SSDs,B$^+$-Tree,write amplification,buffer,key-value stores

1 Introduction

Block-interface Solid-State Drives (block SSDs), while widely used, have long been hindered by the so-called block interface tax [23]. This term refers to the significant performance and resource overhead caused by the mismatch between the internal flash erase mechanism and the traditional disk-oriented block interface. This mismatch leads to unpredictable and debilitating garbage collection (GC) operations, increased DRAM costs due to the flash translation layer (FTL), and the need for over-provisioned space. As a recent alternative to SSDs, the emerging Zoned Namespace interface for SSDs (ZNS SSDs) [8], offers a promise to rid the block SSDs of their block interface tax, by mapping the flash physical space to multiple append-only *zones*. As a result, ZNS SSDs have received significant attention from storage vendors and data centers [5, 10, 18, 29, 35, 53, 81], especially for its potential in eliminating the flash-layer write amplification.

Flash-based key-value stores (KVSs) are widely used in industrial storage systems to provide high-quality service [9, 14, 43, 44, 79]. The B$^+$-Tree-based KVS is known for its superior and stable read performance, but with notorious write amplification (WA) on flash devices due to the *small-grained* node's *random* and *in-place updates* [24, 26, 41, 42, 44, 58, 62, 72]. Numerous studies have shown that the zone interface can help LSM-Tree-based KVSs improve write performance [16, 23, 34, 39, 40, 48, 51, 56, 73]. However, to the best of our knowledge, no studies have been reported on reducing the write amplification of the B$^+$-Tree-based KVSs on real ZNS SSDs despite the potential benefits. Based on observations obtained from our in-depth analysis (detailed in Section 2), we believe that one of the most beneficial opportunities of the zone interface lies in reducing flash-layer write amplification for B$^+$-Tree-based KVSs. Compared with LSM-Tree-based KVSs, B$^+$-Tree-based KVSs can provide higher read performance and more stable read tail latency, especially in small-size key-value pairs (kv-pairs) scenarios (O1). Our preliminary experiments also show that if B$^+$-Tree-based KVSs can fully utilize the zone interface by converting random updates into sequential writes, it will improve write bandwidth and reduce write amplification on flash media (O2). Furthermore, B$^+$-Tree-based KVSs can directly manage flash media through the host-side zone interface, thus avoiding file system overhead (O3).

However, we find that existing B$^+$-Tree-based KVSs designed for conventional block SSDs fall far short of fully reaping the potential benefits offered by the zone interface. *1) Degraded read performance on the index layer.* Applying a buffer structure to merge kv-pairs is a classic idea for the B$^+$-Tree index to reduce write amplification. We argue that the reason why existing buffered-B$^+$-Tree index, including tree-level buffered FD-Tree [45], node-level buffered B$^\epsilon$-Tree [21, 24], and kv-pair-level buffered Bw-Tree [44, 58, 69, 72], exhibit degraded read performance is because of their high access overheads for extra read-unfriendly buffered structures. Thus, they lose the most important capability of the B$^+$-Tree index, that is, ensuring excellent read performance. *2) Idiosyncratic restrictions on the flash layer.* We are not aware of any existing solutions for effectively executing B$^+$-Tree operations on ZNS SSDs, which must convert the update-in-place tree node operations to append-only zone operations. A potential solution involves using a host-side mapping table to map the append-only zones to conventional pages, but this approach demands significant DRAM resources [2]. A second potential approach is to use zone-adopted file systems, such as F2FS [3] and BtrFs [1], as a middle layer for B$^+$-Tree-based KVSs on ZNS SSDs. However, this results in additional metadata write amplification and underutilizes the bandwidth of ZNS SSDs. Furthermore, both approaches render the zone interface's active management features ineffective.

Motivated by our in-depth experimental analysis and observed limitations of current solutions, we propose a novel ZKV designed for ZNS SSDs. ZKV takes a software-hardware co-design approach by devising an enabling

three-level buffer structure for B^+-Tree index on the zone interface, which works hand-in-glove to reduce write amplification while ensuring excellent read performance. At the heart of ZKV is its core index Z^+-Tree, a heterogeneous B^+-Tree index that places the update-in-place internal nodes in memory and append-only leaf nodes in ZNS SSDs. At the tree level, the merged filter first aggregates and sorts kv-pairs in memory. It then separates frequently updated *hot* pairs from less-frequently updated *cold* pairs, retaining the former in memory while filtering out and merging the latter into the Z^+-Tree via a batch array. At the leaf-node level, a limited number of adaptive delta buffers (ADB) are used to further collect kv-pairs, with a leaf node linking to at most one ADB. Working with the coarse-grained batch array, each ADB structure automatically resizes its buffer size according to the sequentiality of kv-pairs inside a batch, further increasing the number of pairs written to flash each time. At the zone level, the ZFlusher combines a write sequencer with a read-only cache to ensure the zone's strict write order and provide a high read-hit ratio. ZFlusher utilizes a ring buffer to combine multiple 4KB leaf data pages into a larger zone-friendly block append operation. Each open zone is also assigned its own ZFlusher instance via an exclusive mechanism, preventing the limited inter-zone concurrency from becoming a bottleneck.

We implement a prototype of ZKV on a commercial ZNS SSD device [12], and evaluate the performance via widely-used YCSB workloads [25]. ZKV shows a 3.12x/2.53x higher insert/read throughput than the current buffered-B^+-Tree-based KVSs. With three additional realistic workloads, ZKV is compared against two representative industrial KV storage engines on ZNS SSDs, the LSM-Tree-based RocksDB [23], and the B^+-Tree-based WiredTiger [3], demonstrating its attainment of the best of two worlds.

In summary, this paper makes the following contributions:

• We conduct an in-depth analysis of experimental observations to draw insights into the potential benefits of building B^+-Tree-based KVSs on zone interfaces.

• We determine that existing B^+-Tree-based KVS designed for block SSDs cannot fully utilize the zone interface because of the degraded read performance and relying on the extra file system layer.

• We propose a write-optimized B^+-Tree-based ZKV designed for ZNS SSDs. ZKV takes a software-hardware co-design approach by devising a three-level buffered structure for B^+-Tree index based on zone interfaces to reduce write amplification while ensuring excellent read performance.

• Under comprehensive synthetic and realistic workloads, we extensively evaluate the efficiency and effectiveness of ZKV, against existing B^+-Tree-based KVSs and two industrial KV storage engines on a commercial ZNS SSD.

2 Background and Motivation

2.1 Zoned Namespace Solid-State Drives.

While conventional **Solid-State Drives (SSDs)** are widely used in storage systems [26, 62], the mismatch between SSD's internal flash media's page write and block erase characteristics causes them to suffer from an unavoidable *block interface tax* [8, 23] that not only degrades performance but also increases SSD cost. The **Zoned Namespace (ZNS)** is a new NVMe storage interface for SSDs that abstracts flash-based space as append-only *zones* to avoid the block interface tax and shifts the data management inside a zone to the host [7]. Compared with SSDs, the host-side zone interface of ZNS SSDs eliminates write amplification caused by GC operations, resulting in more predictable write bandwidth and reducing the need for additional over-provisioned (OP) flash space. Furthermore, the coarse zone granularity decreases the FTL memory requirements and alleviates a significant constraint on increasing SSD capacity, facilitating the proliferation of low-endurance quad-level cell (QLC) SSDs.

A ZNS SSD tracks per-zone writes via write pointers as shown in Figure 1, so any write to a zone must specify the same offset the write pointer indicates. The Zone Size of ZNS SSDs is greater than the writeable zone capacity, which aligns with the power-of-two industry norm introduced with SMR HDDs. To match the offset of a host-issued write with the on-device write pointer of a zone, the host can only issue one request to each zone

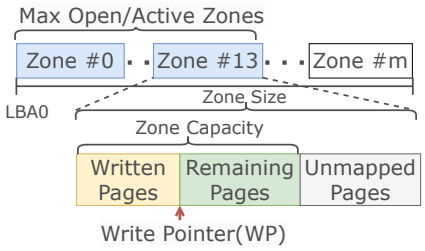

Fig. 1. The zone interface has limited intra-zone concurrency, meaning that the write offset within a zone must completely match the write pointer, and its inter-zone concurrency is upper bound by the maximum number of open active zones.

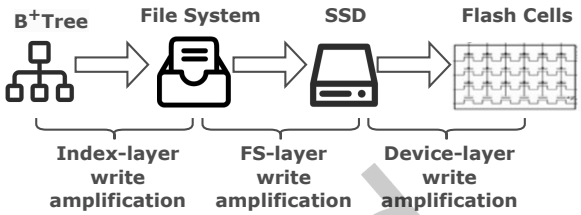

Fig. 2. The three-layer IO stack for B$^+$-Tree-based KVSs. B$^+$-Tree-based KVSs will cause write amplification at the flash layer due to small-size and random writes, while incurring additional write amplification at the FS layer by file system metadata.

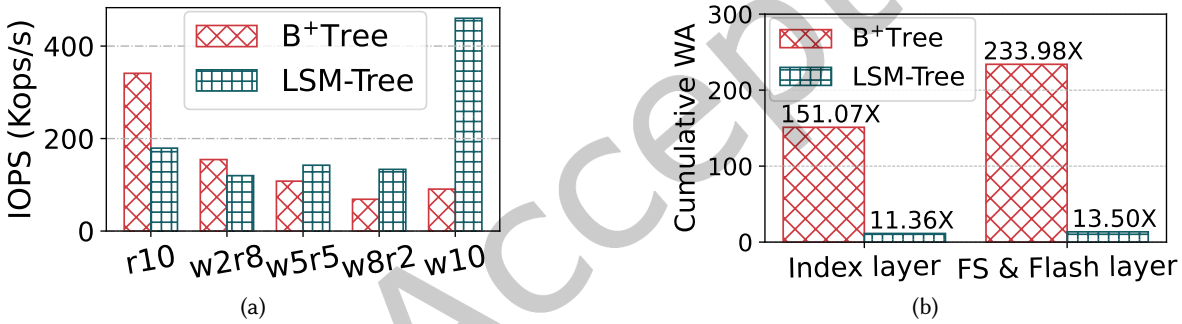

Fig. 3. Comparison of B$^+$-Tree-based KV (WiredTiger) and LSM-Tree-based KV (RocksDB) in two metrics: (a) *Throughput* on different ratio of read(r) and write(w) operation (e.g., r(w)10 = 100%read(write)), (b) *Cumulative Write Amplification* on the index, file systems (FS) and flash layer, Both KVs are running on a real ZNS SSD configured in §4.1.

at a time, thereby limiting intra-zone parallelism. Besides, the internal resource limitations of ZNS SSDs also restrict the maximum number of zones simultaneously open for writing. For instance, the large-zone ZNS SSD device, such as WD DC ZN540, can only open 14 zones for writing simultaneously [12, 27, 66].

2.2 The B$^+$-Tree-based Key-value Stores.

Flash-based key-value stores (KVSs) play a key role in industrial storage systems to provide high-quality service. The read-optimized B$^+$-Tree-based KVSs [14, 43, 44] and write-optimized LSM-Tree (Log-Structured Merge tree) based KVSs [9, 67, 79] are the most deployed among these KVSs. Numerous existing studies have demonstrated that the zone interface of ZNS SSDs can help LSM-Tree-based systems such as RocksDB [9] reduce write amplification [16, 23, 34, 39, 40, 48, 51, 53, 56, 73]. However, to the best of our knowledge, there are no reported studies on reducing write amplification of B$^+$-Tree-based KVSs on real ZNS SSDs. Moreover, our preliminary experimental analysis finds that existing B$^+$-Tree-based KVSs designed for block SSDs fall far short of fully benefiting from the append-only zone interface's advantages. This analysis led us to the following hypothesis: ***the zone interface provides new opportunities for not only LSM-Tree-based KVSs but also B$^+$-Tree-based KVS.*** We run the B$^+$-Tree-based KVS and the LSM-Tree-based KVS on a real ZNS SSD and compare them under the three-layer flash IO stack as shown in Figure 2: index layer, file system (FS) layer, and flash (device) layer. We obtained the following observations to prove our hypothesis.

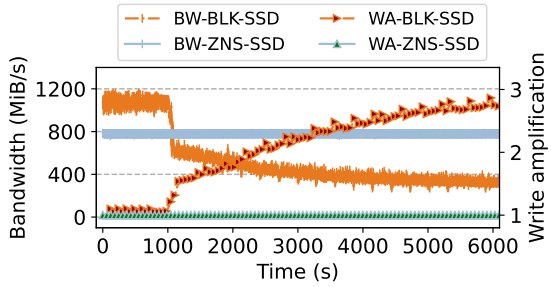

Fig. 4. ZNS SSDs eliminate the block interface tax of block SSDs (BLK-SSD): no flash-layer write amplification (WA) and more stable write bandwidth (BW).

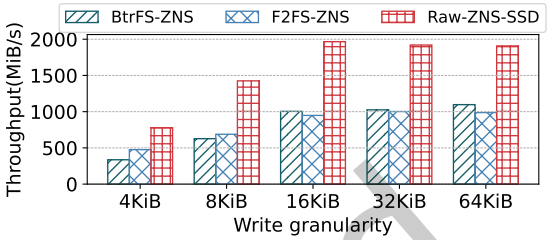

Fig. 5. Current file systems like F2FS-ZNS can manage the zone interface for B$^+$-Tree-based KVSs, but cannot fully utilize its write bandwidth.

B$^+$-Tree indexes exhibit higher and more stable read performance than LSM-Tree indexes. **(Observations #1 (O1))**. LSM-Tree uses a zone-friendly append mechanism when flushing Sorted String Table (SST) files. LSM-Tree ensures order only within individual SST file, while these SST files may be out of order among themselves. To maintain a partially ordered structure, the LSM-Tree asynchronously merges out-of-order SST files level by level in the background, which may result in lower write performance during the merge phase than B$^+$-Tree [26, 52]. As a result, the LSM-Tree's most significant drawback is causing data to be spread across multiple levels, thereby compromising the stability of read operations. For instance, in a typical 3-leveled LSM-Tree, locating a key-value pair (kv-pair) requires first searching the in-memory Memtable, then traversing the out-of-order L0 level, followed by multiple binary searches through the L1 to L2 levels. This results in less stable read performance, especially for operations like scans that require traversal. Many works try to improve the read performance of LSM-Tree-based KVSs by adding secondary indexes [75] or additional caches [70, 78], but this will incur a lot of memory overhead and its effectiveness is limited by LSM-Tree's inherent structural constraints. As illustrated in Figure 3, while the LSM-Tree-based KVS offers superior write performance, its read performance lags considerably behind that of the B$^+$-Tree-based KVS.

B$^+$-Tree-based KVSs can utilize append-only zone interfaces to improve write throughput. **(Observation #2 (O2))** While B$^+$-Tree offers superior read performance (Figure 3a), it induces much higher write amplification than LSM-Tree, on both the index layer and the flash layer (Figure 3b). We argue that the root cause lies in its global order guarantee, which ensures that all the kv-pairs are sorted in leaf nodes and form a hierarchical tree structure. The global order guarantee incurs many *random*, *small-sized*, and *update-in-place* writes, resulting in degraded write performance on flash media [26, 62]. Furthermore, high-cost flash-unfriendly writes induce a large amount of write amplification caused by flash-layer GC operation. As shown in Figure 4, both block SSDs and ZNS SSDs are in trimmed condition before the test, meaning that all of the flash blocks are erased, and no GC operation will be triggered before the free blocks are exhausted. The FIO test tool [37] performs 4KB random write and sequential write loads on BLK-SSD and ZNS-SSD respectively. The former simulates random in-place updates of leaf nodes through the conventional block interface, which causes SSDs to suffer severely from write amplification, leading to unstable and deteriorating write performance. The latter simulates the sequential append operation of leaf nodes through the zone interface, achieving higher and more stable write bandwidth.

B$^+$-Tree-based KVSs should directly manage zone interfaces bypassing the FS-layer. (Observation #3 (O3)) Although the B$^+$-Tree-based KVSs write to flash memory conveniently through the existing file system, they suffer from additional write overhead. The B$^+$-Tree-based KVSs designed for block SSDs, such as WiredTiger based on the POSIX file-system (FS) API, can run directly on a log-structured file system adopted for ZNS SSDs, such as BtrFS-ZNS [1] and F2FS-ZNS [3]. However, we find that the log-structured file system induces extra write amplification and under-utilizes the bandwidth of ZNS SSDs. Moreover, log-structured file systems also prevent

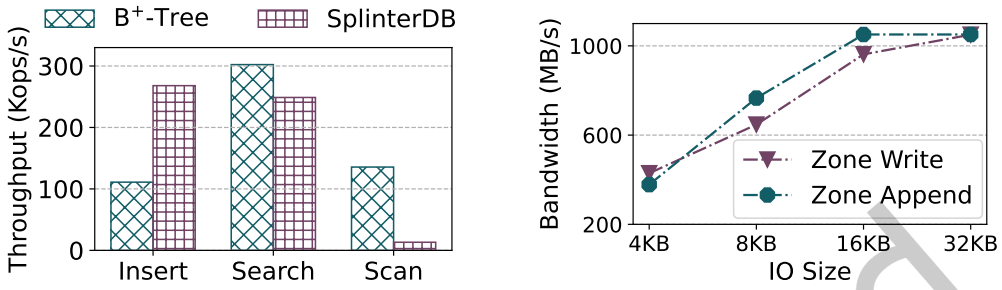

Fig. 6. (a) The B$^\epsilon$-Tree-based SplinterDB significantly improves write performance but causes read performance degradation. (b) The asynchronous zone append command has no obvious bandwidth advantage over the synchronous zone write command when the write granularity is greater than 16KB. The Append/Write commands use 32 queue depth/32 threads respectively, and both write to 14 zones simultaneously.

KVSs from being aware of and taking full advantage of the zone interface's features. As shown in Figure 5, we tested the bandwidth of F2FS-ZNS and BtrFS-ZNS on a ZNS SSD device through the FIO [20] tool at different write granularities. The results indicate that both Btrfs-ZNS and F2FS-ZNS achieves only about 50% of the raw device bandwidth, mainly due to the additional overhead introduced by FS-layer overheads, such as syncing metadata and not fully utilizing the zones' parallel bandwidth [63].

2.3 Motivation and Challenges

The main takeaway from the above observations [$O1 - O3$] is that ***for B$^+$-Tree-based KVS on ZNS-SSDs, a co-design of a write-optimized B$^+$-Tree index structure with the emerging zone interface is necessary to reduce write amplification holistically while ensuring excellent read performance***. Specifically, the main goal of this paper is to design an optimized B$^+$-Tree-based KVS for ZNS SSDs, called ZKV, achieving the best of two worlds: the excellent write performance of LSM-Tree and the superior read performance of B$^+$-Tree, by addressing the following design challenges.

Challenge #1 (C1): Avoiding degraded read performance due to multiple buffer structures at the index layer. B$^+$-Tree-based KVSs cause write amplification on flash media due to the *granularity mismatch* between the kv-pair size and the operational size inside flash media. Applying and merging a buffer structure for the B$^+$-Tree index is a classic idea for reducing write amplification. Buffered B$^+$-Tree can be roughly divided into three categories based on their merge granularity: tree level, single-node level, and kv-pair level. We argue that *the existing buffered B$^+$-Tree index reduces write amplification but at the expense of read performance.*

FD-Tree [45] merges an in-memory B$^+$-Tree directly into a multilevel structure on flash memory at the tree level, requiring hierarchical accesses similar to LSM-Tree for read operations. The FD-Tree also involves expensive merge operations: when the tree-level buffer exceeds its capacity after a single insertion, and the whole FD-Tree has to be entirely rewritten. This process may result in an unacceptable response time. The node-merge-level B$^\epsilon$-Tree [21] buffers kv-pair for each node (including internal nodes), but introduces additional traversal tail latency. The SplinterDB based on B$^\epsilon$-Tree [24] adds a quotient filter to each buffer structure, which takes up a lot of memory, and its scan performance is much lower than that of the classic B$^+$-Tree. For example, as illustrated in Figure 6a, while SplinterDB improve write performance by up to 2.41x, they experience a 18.6%/90.3% decline in point/range search performance. Bw-Tree [44, 58, 69, 72] links each new kv-pair modification to the existing node as a delta record. The longer the unordered and dispersive delta record list is, the more pronounced the write amplification factor will be. However, a leaf node may consist of a backward chain of updates with new delta records in main memory and older delta records on flash media. Thus, a search operation of Bw-Tree may

cause multiple accesses to flash media, causing severe degradation in read performance. Furthermore, existing buffered B$^+$-Tree indexes rely on an additional FS layer to interact with the idiosyncratic restrictions of zone interfaces, leading to additional performance loss and the inability to operate the data layout directly.

Challenge #2 (C2): Meeting the idiosyncratic restrictions of zone interfaces while achieving high utilization at the flash layer. ZKV attempts to directly manage the host-side zone interface for the B$^+$-Tree structure, removing the extra overhead of the middle file system layer. So, ZKV needs to cope with the following zone interface limitations:

1) Sequential Write Constraints: B$^+$-Tree-based KVSs can easily convert random and in-place updates into zone-friendly sequential append operations for nodes through some methods like the copy-on-write mechanism. Current B$^+$-Tree-based KVSs use a buffer pool to operate memory nodes, thereby improving SSD utilization of SSDs [6, 14]. Things get complicated when considering evicting and flushing a node from a buffer pool. KVSs on ZNS SSDs must ensure the strict write order: a zone can only be written to the *wp* offset, meaning that the page ID allocated first must be written first. This requires a sequential order guarantee in the buffer pool, which leads to the infeasible LRU-like postponed write buffer mechanism. For example, the allocated pages in the write buffer are *{3, 4, 5}*, but page *5* is evicted first, which will not match the current *wp* offset *3*, thus causing system IO errors.

2) Limited Inter-Zone Concurrency: B$^+$-Tree indexes rely on high concurrency to ensure efficient writing of small kv-pairs. However, the internal resource limitations of ZNS SSDs restrict the maximum number of zones simultaneously open for writing. For instance, the WD DC ZN540, can only open 14 zones for writing simultaneously. Although the zone append command can support high queue depths for I/O, its bandwidth is not higher than zone write and brings more complex changes to the concurrency mechanism of the B$^+$-Tree index [68, 71], as shown in Figure 6b. Although the zone append command can support high I/O queue depths, its bandwidth is not higher than zone write with the high queue depth (32) and the 16K granularity as shown in Figure 6b. This phenomenon is also found in other works [68, 71]. Furthermore, the disordered and asynchronous append commands will brings more complex changes to the concurrency mechanism of the B$^+$-Tree index. ZKV adopts the write commands to fully utilize the ZNS SSD's write bandwidth through a lightweight concurrency mechanism and three-level buffer mechanism.

3) Read IOPS vs Write Bandwidth. Recent work recommends using 4KB pages to obtain the highest random read IOPS and lowest latency for NVMe SSD [30, 32]. However, the 4KB write bandwidth will not be able to achieve the maximum bandwidth as shown in Figure 5 and 6b. Although a larger write granularity can obtain a larger write bandwidth, it will also bring greater read amplification and thus reduce the overall read throughput.

3 ZKV Design

ZKV's main design goal is to fully utilize the zone interface's potential to reduce the B$^+$-Tree index's flash-layer write amplification caused by small kv-pair writes, while retaining its read performance advantage. Building on insights from existing buffering mechanisms, ZKV introduces a novel three-level buffering architecture to these design goals. ZKV considers the following three design decisions (abbreviated as **D**) for a novel B$^+$-Tree-based KVS on ZNS SSDs.

D1. We found that a small number of leaf nodes in the B$^+$-Tree have higher update counts, while most leaf nodes have smaller update counts [47]. Motivated by this, ZKV includes a merge filter that separates frequently and infrequently updated nodes, keeping hot data in memory to reduce flash write overhead.

D2. ZKV addresses these limitations through a memory-resident adaptive delta buffer that dynamically aggregates scattered kv-pairs at the node level and tries to evict high-utilization buffer structures as much as possible. This design maintains efficient read performance while operating under fixed memory constraints, and it automatically prioritizes buffer allocation to frequently updated (hot) nodes, optimizing memory usage and ensuring low flash-level write amplification.

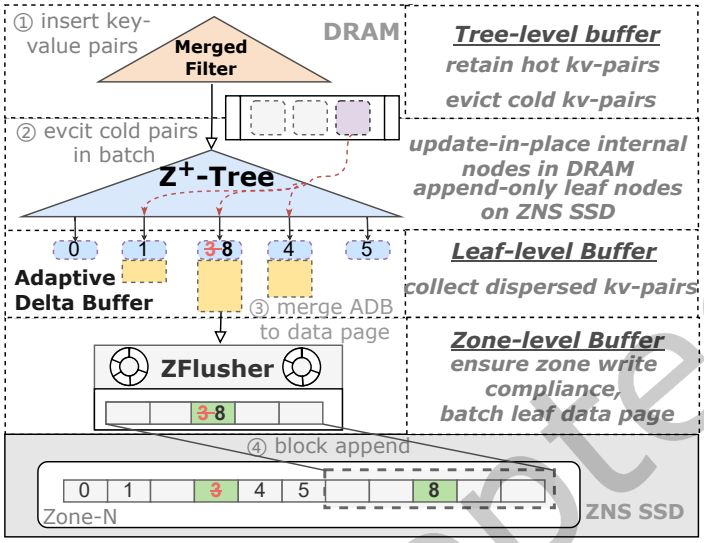

Fig. 7. As the core index of ZKV, the heterogeneous Z+-Tree uses a three-level buffer structure to enable append-friendliness while preserving B+-Tree's read superiority. The blue dotted and gray solid rectangles represent Z+-Tree leaf-node metadata and data pages respectively.

D3. ZKV utilize the ZFlusher module ensures compliance with the sequential write constraints of the ZNS interface by consolidating small pages into large blocks before writing, thereby optimizing both write performance and zone utilization.

3.1 Overview of ZKV

ZKV's Workflow. Figure 7 shows an architecture overview and high-level workflow of ZKV, where its core index Z+-Tree utilizes a three-level buffer structure to reduce write amplification by separating hot and cold kv-pairs, sorting and batching dispersed kv-pairs, and addressing the idiosyncratic restrictions of the zone interface. In ZKV, kv-pairs are first aggregated and sorted in a small-size merged filter ① that keeps hot kv-pairs in memory and evicts and inserts cold kv-pairs into the Z+-Tree at the granularity of a batch array. While inserting batched kv-pairs can reduce the traversal overhead ②, kv-pairs in batch may scatter to many leaf nodes of Z+-Tree. To prevent serious write amplification caused by a leaf node directly flushing its received kv-pairs, Z+-Tree employs a pool of adaptive delta buffers (ADBs), of which each is linked to at most one such leaf node, to buffer and batch dispersed kv-pairs belonging to this leaf node in each batch insertion. A linked ADB automatically expands its capacity depending on the number of buffered kv-pairs to avoid wasting memory space. When an ADB buffers more than 16 kv-pairs, a leaf node merges it with the data page into a new page, as writing 16 kv-pairs for a 4KB page is sufficiently efficient for reducing write amplification based on our empirical analysis ③. When merging an ADB with its linked leaf node, ZFlusher ensures the write order guarantee of concurrent write operations and maximizes the zone interface concurrency ④. ZFlusher will batch multiple 4KB leaf data pages into larger blocks, ensuring low read and write amplification while fully utilizing the large-granularity write bandwidth on ZNS SSD. ZKV's search process only needs to access the ordered merged filter and Z+-Tree twice, and a leaf node of the Z+-Tree is linked to at most one ADB. Although Z+-Tree in its current form is a proof-of-concept prototype based on ZNS SSD, it nonetheless provides basic functions of KV storage systems: point and range queries, variable-length value pairs, and basic crash recovery.

Z+-Tree. As shown in Figure 7, the heterogeneous Z+-Tree utilizes the log-structured (append-only) leaf nodes to shift from traditional in-place update operations to append-only operations well-suited for the zone interface. Both the internal nodes and the leaf node's metadata reside in memory. The metadata contains an 8-byte *pid*

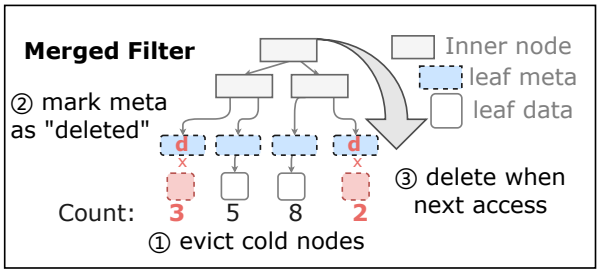

Fig. 8. The merged filter retains hot kv-pairs while evicting cold kv-pairs at the leaf-node granularity through an LFU strategy. The deletion of cold leaf nodes is postponed to the next access to avoid upwardly cascading modifications.

representing its global logical page offset in the ZNS SSDs, in units of 4KB pages. To achieve higher IOPS, Z$^+$-Tree uses a 4KB leaf node to align with the flash page size [30, 32]. ZFlusher will automatically batch multiple pages to increase write bandwidth and reduce read and write amplification. Z$^+$-Tree uses 8-byte keys and values by default, and the total memory capacity of the internal nodes will not exceed 0.5% of the total capacity of leaf nodes in flash memory. For large values, ZKV generates a log record and returns its global offset in ZNS SSD as the value to be inserted, further reducing the memory usage of internal nodes. Z$^+$-Tree is adopted from the widely-used pure-in-memory B$^+$-Tree (olc-btree) index with the optimistic concurrent locking mechanism [4]. Its read operations traverse leaf nodes without locks, while write operations access nodes exclusively through the optimistic lock-coupling (olc) mechanism. To simplify the concurrency mechanism, olc-btree does not contain the classic sibling pointer, thus also reducing the write amplification on SSDs.

3.2 Merged Filter

The merged filter identifies and separates frequently updated *hot* kv-pairs from less-frequently updated *cold* kv-pairs by keeping the former in memory while efficiently merging the latter into Z$^+$-Tree in *batches* rather than merging kv-pairs one by one. Specifically, the merged filter serves two primary functions: (1) aggregating and sorting dispersed kv-pairs through a tree structure (*"merged"* semantic) and (2) swiftly evicting and inserting cold kv-pairs into Z$^+$-Tree in a *batch array*, explained next, (*"filter"* semantic). We use the pure-in-memory olc-btree to implement the merged filter for simplicity, providing efficient read performance [72], high concurrency [64], and a unified interface with Z$^+$-Tree. As the merged filter's main function is to sort and separate hot and cold kv-pairs and only needs to occupy tens of MB memory, implementing it as a more complex radix tree or red-black tree will not bring obvious added performance benefits. Compared to the tree-level FD-Tree, a merge filter can separate frequently and infrequently updated nodes, keeping hot data in memory to avoid expensive merge operations and provide a higher read hit ratio.

Node eviction. The eviction mechanism of the merged filter aims to efficiently pick out the coldest leaf nodes and minimize the blocking of the foreground threads. The merged filter maintains an atomic 8-byte variable *valid_nodes* to record the number of valid leaf nodes. Since the *valid_nodes* variable is only updated when splitting or deleting a leaf node, its maintenance overhead is negligible. Each leaf node maintains an access *counter*, incremented by one for each access and halved after each eviction process for dynamic load changes. When *valid_nodes* exceeds a predefined threshold, the merged filter will be temporarily locked to execute an LFU-like eviction process, in which one of the foreground threads will traverse all leaf nodes to find the coldest leaf nodes by adding the pointer to each leaf node to a heap structure. The heap will pass the pointer arrays of cold leaf nodes based on its *counter*, called *batch array*, to the Z$^+$-Tree. Finally, the merged filter will quickly delete evicted leaf nodes through a *shadow deletion* technique, explained next.

Shadow deletion. Deleting a node in the classic B$^+$-Tree structure may trigger a high-overhead structural modification operation, seriously hindering concurrency and becoming a performance bottleneck [33, 77]. To

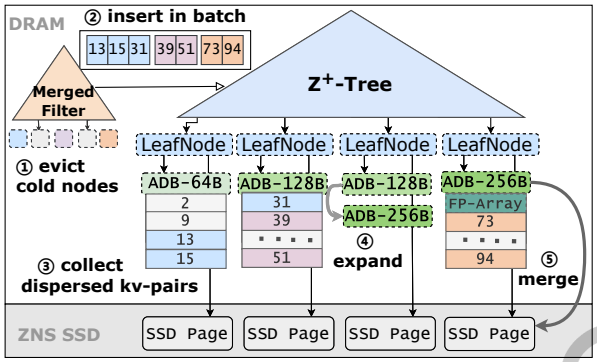

Fig. 9. An adpdative delta buffer (ADB) further buffers and groups dispersed kv-pairs belonging to a leaf node. This is because kv-pairs in a batch may be scattered among multiple leaf nodes, resulting in severe write amplification.

avoid the bottleneck, we adopt a *shadow deletion* technique for fast node eviction as shown in Figure 8. Each leaf node in the merged filter is logically separated into two parts, the metadata and the key-value data; both reside in memory, which is different from the Z$^+$-Tree's leaf nodes. When being deleted from the merged filter, the leaf node will directly set the pointer to the key-value data as null and move the data to form the *batch array*, without cascading upward to delete the pointer to it from its parent node (①). Thus, a deletion will only require a modification inside a leaf node. However, if the leaf-node metadata is retained, the merged filter will obtain an extremely sparse distribution of the stored kv-pairs among these "shadow deleted" and valid leaf nodes (②). Therefore, after clearing the key-value data pointer, the merged filter will specially mark the leaf node that has been "shadow deleted". When a "shadow deleted" leaf node is accessed next time, its parent node will delete the pointer, which points to the "shadow deleted" leaf metadata, to make the merged filter compacted (③). In addition, we notice that the merged filter's height usually is three. To avoid the root node becoming a concurrency bottleneck, the deletion and merging of internal nodes will not lead to regenerating a new root node.

3.3 Adaptive delta buffer.

As shown in Figure 9, Z$^+$-Tree first reduces the insertion overhead by inserting cold kv-pairs in the form of a batch array (① and ②). Although a batch evicted from the merged filter contains less-frequently updated kv-pairs, these dispersed kv-pairs may span many leaf nodes of Z$^+$-Tree and thus cause high write amplification. For example, a batch may correspond to dozens of leaf nodes in Z$^+$-Tree, so each leaf node may only receive a few kv-pairs. Fixed-size buffering structures, such as those used in B$^\epsilon$-Tree [21, 45], often result in inefficient memory usage, while KV-pair-level linked buffers, as in Bw-Tree [72], can introduce additional search overhead. To address this problem, Z$^+$-Tree uses an adaptive delta buffer (ADB) to buffer and group dispersed pairs belonging to the same leaf node (③) to reduce write amplification. As the number of buffered kv-pairs increases, a small-sized ADB attempts to transition to a larger structure (④). When ADB's memory usage exceeds the set threshold, the Z$^+$-Tree will try to merge an ADB-256 structure into its data page as much as possible (⑤).

Cooperative batch insertion. With a buffer pool running out of available space, an insertion into the B$^+$-Tree index with a single kv-pair will cause a buffered leaf node to be flushed, resulting in high write amplification. An insertion to the Z$^+$-Tree is at a batch granularity rather than in single kv-pair granularity to avoid such overhead. Through an eviction operation, the merged filter evicts some less-frequently updated (cold) leaf nodes via an *batch array*. A cold leaf node evicted from a merged filter is a batch in the batch array. Since the kv-pairs in a merged filter's leaf node are ordered, every batch is sorted based on its first key in the batch array. Z$^+$-Tree takes out a batch from the batch array and inserts it into its leaf nodes as continuously as possible. The exclusive lock of the merged filter is released after generating the *batch array*. The newly arrived write threads will repeatedly and

cooperatively execute the following procedure in sequence until all the batches in the batch array are inserted into the Z$^+$-Tree. First, the foreground thread retrieves a batch from the *batch array*, then finds the corresponding leaf node in the Z$^+$-Tree for its first kv-pair (*lower*) of the batch and calculates the last pair (*upper*) that can be inserted continuously within a traversal. Second, Z$^+$-Tree inserts all the pairs from the *lower* to the *upper* at once. Third, the current thread continues to look for the next Z$^+$-Tree's leaf node to execute the above operations until all kv-pairs in the current batch have been inserted. Fourth, after merging the current batch into the append-only leaf nodes in the Z$^+$-Tree, the foreground thread will continue its previous interrupted operations. As for the search operation, it will traverse the merged filter, the *batch array*, and finally, the Z$^+$-Tree in sequence. The search overhead is quite low since the merged filter and the *batch array* are relatively small-sized and sorted in memory.

Structure of adaptive delta buffer. ADB aims to batch enough kv-pairs for each write operation on a leaf node to reduce write amplification sufficiently. We find that a considerable number of leaf nodes need to write several kv-pairs during each batch-insertion operation under uniform workloads. Hence, Z$^+$-Tree utilizes three different-size ADB structures, ADB-64B/128B/256B buffering 4/8/16 kv-pairs, respectively. Depending on the number of dispersed kv-pairs collected from a batch-insertion operation, a leaf node will create and link an appropriate-sized ADB structure. For example, when a leaf node needs to insert seven pairs, it can directly initialize an ADB-256B. If more than 16 kv-pairs from a leaf node need to be inserted at a time, Z$^+$-Tree will not link a new ADB structure but directly merge the kv-pairs in the flash memory. Because batching 16 kv-pairs is able to reduce write amplification significantly. ADB also adopts the following optimization to avoid the loss of read performance. Each leaf node will link to only one ADB structure at most to avoid high-overhead pointer chasing in the linked list. ADB's valid slots are also recorded in the leaf node metadata. As a result, each ADB structure only contains kv-pairs without other extra metadata information. The kv-pairs in ADB are not sorted because a binary search for a dozen kv-pairs is insufficient to compensate for the extra overhead of sorted writes. In addition, ADB structures are all cache-line-aligned, so traversing them only requires one additional cache line miss in most cases. To avoid the additional memory access overhead, the 4-cache-line-sized ADB-256 uses SIMD-accelerated fingerprints (FP-Arrray) to filter out negative search operations like persistent indexes [57]. The FP-Array occupies 15*1 bytes. Each byte represents a fingerprint generated from each key's lowest 1-byte hash value. For small-sized ADB-64/128, traversal search is faster than binary search and only requires one extra cache line miss in most cases. For large-sized ADB-256, most of the high-cost negative searches have been filtered out, and a search with matching fingerprint hits can jump directly to the corresponding kv-pair.

Auxiliary ADB pool. When ADB's memory usage exceeds the set threshold, the Z$^+$-Tree will try to merge an ADB-256 structure into its flash page as much as possible. Every ADB structure is created and recycled from an auxiliary *ADB pool*. The ADB pool maintains three linked lists for three different-sized ADB structures, list-256/128/64. The metadata of a newly created ADB structure's leaf node will be inserted at the end of the corresponding linked list. Since the linked list stores the metadata of the leaf nodes, a leaf node can naturally merge its ADB structure to flash pages and protect the critical area by reusing the leaf node's lock mechanism. To evict an ADB structure, the ADB pool only needs to find the first head item from the list-256, list-128, and list-64. The auxiliary ADB pool for ADB structures only requires two efficient operations: an insertion to the tail of the linked list and a deletion from the head of the linked list. Furthermore, The ADB pool is divided into M instances (32 in our evaluation) according to the page_id to avoid high contention in the critical area. It is worth noting that the Z$^+$-Tree will merge ADB-256 as much as possible, so that each flash page write can process a dozen kv-pairs on average, significantly reducing the write amplification of B$^+$-Tree on flash memory.

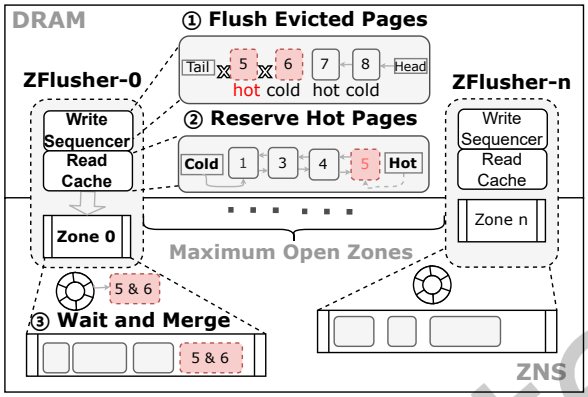

Fig. 10. ZFlusher ensures strict write order within a zone and high inter-zone concurrency, while further batching multiple leaf data pages into a larger-granularity append operation.

3.4 ZFlusher

A leaf node of Z$^+$-Tree merges its ADB structure with the data page through the log-structured merge mechanism, generating a new data page that needs to be appended to the new offset of ZNS SSDs. Every append operation on the zone interface requires strict sequential write order, high concurrency guarantee, and high write bandwidth utilization. Further, a batch insertion of Z$^+$-Tree will generate thousands of pages waiting to be appended, increasing the write overhead. ZFlusher fully utilizes the zone interface to provide efficient append operations for the Z$^+$-Tree interface. As shown in Figure 10, ZFlusher combines a write sequencer with a read-only cache, ensuring zone's strict write order and providing a high read-hit ratio by caching frequently-accessed pages (① and ②). At the zone level, ZFlusher will utilize a ring buffer to batch multiple 4KB leaf data pages into a larger zone-friendly block append operation (③). Each open zone is also assigned its own ZFlusher instance via an exclusive mechanism, preventing the limited inter-zone concurrency from becoming a bottleneck.

Write Sequencer. A write sequencer in a ZFlusher maintains the write order in a zone. A write sequencer uses the first-in-first-out (FIFO) algorithm to ensure the strict write-order guarantee within a zone. The FIFO mechanism is implemented on the linked list, supporting high efficiency for transferring the page data. When Z$^+$-Tree creates a new leaf node or updates a leaf node, it first allocates a new cache page from a write sequencer. The cache page will increase the *wp_* offset in its corresponding zone object and return it as the global logical page offset (*page_id*) to the leaf node. This operation ensures that the new flash page is pre-allocated and waits for the subsequent flush operation. Then, the write sequencer will evict the cache page with the longest residence time according to the FIFO mechanism, which ensures that the evicted page offset is aligned with the zone's current *wp_*.

Read Cache. Combined with the merged filter and ADB structures, most leaf nodes flushed to the ZFlusher are cold, with a lower probability of being modified again, alleviating the hit-rate reduction problem from the classical FIFO algorithm to some extent. However, the write sequencer still shows a low hit rate for frequently accessed (read) pages. ZFlusher uses a read cache to retain frequently accessed hot pages evicted from a write sequencer. A write sequencer tracks a counter for each cache page. When the counter exceeds a predefined threshold, the page is not invalidated after flushing but directly migrated to the read cache. In our experiments, setting the threshold to 1 can achieve good results. When a new insert request comes in, it retrieves or creates a cache page by accessing three objects in turn: the write sequencer, the read cache, and the zone object. If it hits the write sequencer, a request can retrieve a flash page immediately and increase the counter. Otherwise, it will read the page from the read cache or ZNS SSDs based on the *page_id*. We found that the classic LRU

algorithm has a high lock competition overhead when multiple threads access it. Therefore, read cache adopts the state-of-the-art clock-based algorithm SIEVE [80] to achieve a higher hit rate with superior scalability.

Ring Merger. Recent research shows that using 4KB leaf nodes in B$^+$-Tree can minimize read/write amplification while maximizing the IOPS of NVME flash devices [30]. However, Figure 5 shows that small-sized zone writes, especially 4KB, can not fully utilize the zone interface bandwidth. ZFlusher uses a ring merger to cache the pointer of an evicted page in a write sequencer and greedily check whether the page has been written. If there are consecutive pages that can be flushed, they will be merged into a new group of consecutive pages (set to 4 pages in our evaluation) and flushed to the flash memory through a *zone write()* operation. Compared with directly using larger-sized pages, the ring merger reduces write and read amplification for kv-pairs and improves the effective bandwidth utilization of ZNS SSD.

3.5 Other functions.

Write ahead log area and MetaLog for consistency. ZKV adopts the classic Write-Ahead Log (WAL) mechanism to ensure the consistency of inserted kv-pairs [15]. *1) Crash Consistency:* Before inserting a kv-pair into the merged filter, it is first written to the Write-Ahead Log Area (WALA). To minimize performance overhead, WALA is divided into multiple independent instances (e.g., 48 in our evaluation). Each instance contains a lock-based ring buffer queue to enhance flushing efficiency and provide low overhead for insertion. When a ring buffer queue reaches a threshold (e.g., 4KB), it is temporarily blocked and flushes the buffered kv-pairs as WAL files into flash media, following practices similar to RocksDB. The WAL file is written to a separate device following common practices [59]. Except in the variable length value scenario, it directly occupies several zone interfaces on the same ZNS SSD device. *2) Fast Recovery:* To support fast recovery, ZKV generates periodic snapshots of the internal structure of Z$^+$-Tree [24], referred to as the MetaLog. The MetaLog is created and flushed during a specified period of batch merging, thereby reducing interference with foreground operations. Each MetaLog is associated with a self-version number, which is inspired by the basic Multi-Version Concurrency Control (MVCC). During recovery, ZKV can quickly reconstruct the system state using the most recent MetaLog and WAL entries generated after the corresponding version. ZKV replays kv-pairs from the WAL files in order, starting from the corresponding version number. Because the number of key-value entries after the latest MetaLog is limited, recovery is significantly accelerated. Similar to RocksDB's MemTable, ZKV may lose unflushed WAL entries if a crash occurs before flushing. However, this loss is minimal, as the WAL is flushed every 4KB. If the upper-layer application requires strict data persistence, Z$^+$-Tree can force WAL to be flushed for each insertion. However, this will lead to unoptimizable write amplification on the WAL.

Variable length values. Z$^+$-Tree supports variable-length values in a simple and conventional manner [6]. For large-sized values, an 8-byte global logical address (*offset*) on the ZNS SSD is pre-allocated, and this *offset* is inserted into the Z$^+$-Tree as the "value". Specifically, the WAL reuses part of the *opened* zone through the ZFlusher module to flush kv-pairs to flash media and return the *offset*. Therefore, the raw value either exists in the ZFlusher or has been flushed to the ZNS SSDs. This method ensures efficient write operations, but introduces additional overhead for the read operation. To optimize read performance, Z$^+$-Tree can be adopted to embed large-sized values directly into leaf nodes.

3.6 Overheads Analysis.

We have added a formal complexity comparison in Table 1, focusing on I/O costs. The analysis demonstrates that ZKV (Z$^+$-Tree in the table) achieves lower average insertion and read costs compared to other structures. Our analysis assumes that the performance of B$^+$-Tree variants is primarily determined by the number of I/O operations required for each query or update. To enable a fair comparison within a unified framework, we make several simplifying assumptions: (1) the tree stores N fixed-size kv-pairs, (2) each node can hold up to B entries, and (3) each node access corresponds to a single I/O operation (e.g., a 4KB random read on flash storage).

Table 1. The I/O cost comparison of four B$^+$-Tree indexes.

	Avearge Read Cost	Average Write Cost
B$^+$-Tree	$\log_b m$	$\log_b m$
B$^\epsilon$-Tree	$\log_b m/\epsilon$	$\log_b m/\epsilon * B^{(1-\epsilon)}$
Bw-Tree	$\log_b m + L$	$\log_b m/L$
Z$^+$-Tree	$\log_b m$	$\log_b m/K$

Unlike the classic B$^+$-Tree, B$^\epsilon$-Tree partitions each node into two parts: a buffer that temporarily holds incoming updates, and a pivot area that stores the actual kv-pairs. During each flush operation, updates are propagated from a parent node's buffer to its child's buffer. A tuning parameter, $\epsilon (0 < \epsilon < 1)$, determines the space allocation within internal nodes: approximately B entries for pivots and the remaining $B - B^e$ entries for buffering. This reduction in node fanout from B to B^e increases the height of the tree by a factor of $1/\epsilon$, resulting in up to twice as many I/Os for queries when $1/\epsilon = 0.5$. It shows that B$^\epsilon$-Tree improves write performance by reducing write amplification through node-level buffering, but incurs higher read latency for random accesses. The Bw-Tree mitigates write amplification by appending updates as chained deltas. While this avoids in-place updates, it introduces additional read amplification, as queries must traverse a chain of length L, potentially adding up to L extra I/Os. In contrast, Z$^+$-Tree employs a three-level buffering strategy to significantly reduce write amplification. By aggregating random kv-pairs and writing them sequentially, Z$^+$-Tree lowers write amplification to $\log_b m/(K)$, where $K > 1$ depends on workload characteristics and buffer size. In our experiments, K is typically around 30. Importantly, Z$^+$-Tree preserves the classic B$^+$-Tree 's search complexity, introducing no additional I/Os for reads, thereby maintaining high read efficiency.

3.7 Discussion and Limitations

Zone migration. The current implementation of ZKV aims to reduce the B$^+$-Tree index's flash-layer write amplification caused by small kv-pairs while retaining the read performance advantages. In a production environment, KVSs may need to periodically reclaim the invalid zone space, thus requiring a zone migration mechanism. In the future, we will work on placing the leaf nodes with similar life cycles inside the same zone to reduce the migration overhead.

Small Zone VS Large Zone. Currently, flash hardware vendors have implemented two different-sized zone interfaces. ZNS SSDs with small-sized zones, such as Samsung's PM1731a, contain 40304 96MB zones and 384 maximum number of active zones [10]. A small zone fails to be mapped to flash blocks residing on all dies, causing underutilized die-level parallelism [65]. Conversely, a Large zone SSD can map to almost all dies. ZKV fully utilizes the internal flash concurrency through the lightweight ZFlusher. ZKV can work well on small-zone ZNS SSDs by adding more ZFlusher instances.

Other features of ZNS SSDs. The ZNS SSD device we experiment with has a conventional namespace, which provides a small area for random writes. Our zone space is approximately 4TB, while the conventional namespace is only 4GB. Before the real device becomes available, ZB$^+$-Tree [49] tries to place nodes in the conventional namespace as much as possible to block the cascade update of COW-B$^+$Tree on ZNS SSD. However, the real conventional namespace of ZNS SSDs is only 1/1000 of the zone namespace and has worse write performance, dramatically lowering its performance. The ZNS standard also defines a zone random write area (ZRWA) [17], which adds a window area that can be randomly written after each *wp* pointer. ZRWA allows an asynchronous form of in-place writing, which may create more challenges for B$^+$-Tree's concurrency mechanisms.

Generality of ZKV's Design. ZKV mainly uses a three-level buffer structure to fully convert the random updates of the B$^+$-Tree index into flash-friendly sequential append operations and fully utilizes the current append-only zone interface. In addition to ZNS SSDs, other forms of flash products improve the performance of

flash-based storage systems in different ways. Flexible Direct Placement reduces the GC overhead of flash devices by adding a hint field to existing NVMe commands [13]. CXL-SSD provides higher throughput by combining DRAM with flash media based on the Compute Express Link protocol [19]. Computational SSDs can provide functions such as transparent compression to reduce the write amplification of flash-based storage systems [62]. The core idea of ZKV, converting random updates into flash-friendly appends, is orthogonal to these flash devices. Our main design, including the merged filter and ADB pool, can also be applied to other SSDs.

4 Evaluation

We evaluate ZKV to answer the following questions. *Q1*: How efficient is ZKV compared against existing buffered B$^+$-Tree-based KVSs on ZNS SSDs? *Q2*: How does each enabling technique contribute to the overall performance of ZKV? *Q3*: How competitive is Z$^+$-Tree to production-grade storage engines driven by real-world workloads?

4.1 Experimental Setup

Hardware Environment. Our experiments are performed on an Ubuntu 22.04 server (kernel version 6.6.20) equipped with two Intel Xeon Gold 6430 CPUs with 64 logical cores. The system has 128 GB of 2666MHz DDR4 DRAM (2×32 GB DIMMS per socket). We test on the commercial *Western Digital DC ZN540 4TB* [12].

Baselines. We compare ZKV against current buffered B$^+$-Tree indexes on ZNS SSDs: **B$^+$-Tree**: We implemented a conventional B$^+$-Tree-based KVS, which is implemented on the widely-used olc-BTree [4] with the classic LRU-based buffer pool to reduce write amplification. Its internal nodes are also placed in memory like Z$^+$-Tree. **BD-Tree**: To study the effect of Bw-Tree's KV-pair-level delta records, we apply the delta records mechanism on our implemented B$^+$-Tree index and call it BD-Tree. the BD-Tree we implemented has higher read and write performance than the original Bw-Tree design. The optimistic locking mechanism is shown to have higher performance than Bw-Tree's original lock-free mechanism [72]. All delta records of BD-Tree are placed in memory to avoid accessing flash media and remove additional high-cost mapping tables. The delta record pool is also divided into 64 instances to avoid the GC operation being bottleneck. The length of the delta record list for a leaf node is set to 32 as previous works suggested [72]. **SplinterDB**: SplinterDB is the state-of-the-art B$^\epsilon$-Tree-based KVS [24] and is open-sourced at [11]. We use the default configuration as its origin paper and README. All of the above are running on ZNS SSDs with F2FS-ZNS file systems. Unless otherwise specified, all KVSs' available memory size is set to 400MB, around 12.5% of the dataset size similar to other works. Based on our evaluation, the merged filter, ADB pool, and ZFlusher sizes for ZKV are set to 50MB, 150MB, and 200MB, respectively. Our test found that it can fully utilize the write bandwidth without affecting the read performance. We also compare the performance of ZKV to Production-grade storage engines such LSM-Tree-based RocksDB-ZENFS [9] and B$^+$-Tree-based WiredTiger [14]. We disable irrelevant functions such as write-ahead-log to demonstrate the index performance. For a fair comparison with Z$^+$-Tree, RocksDB-ZENFS sets the write cache to 200MB, with a 200MB LRU-based read cache. WiredTiger also sets the cache size to 400MB. All other parameters are set to default. For a fair comparison, all the minimum granularity are set to 4KB.

Table 2. The prefix characters W, R, U, and S before the numbers indicate the operations *insert, read, update and scan*, respectively.

Benchmarks	**Workloads**				
	a	b	c	d	e
YCSB	W100	W80 R20	W20 R80	R100	S95 W5
Skew	U100	U50 R50	U5 R95	R100	S95 U5

Workloads. Our evaluation uses workloads generated by the widely used YCSB benchmark [25], as listed in Table 2. YCSB-Skewed generates a skewed load (80% hotspot fraction), while YCSB-Uniform is evenly distributed. The columns and rows of the table can be combined to indicate specific workloads, such as YCSB-a and Skew-a

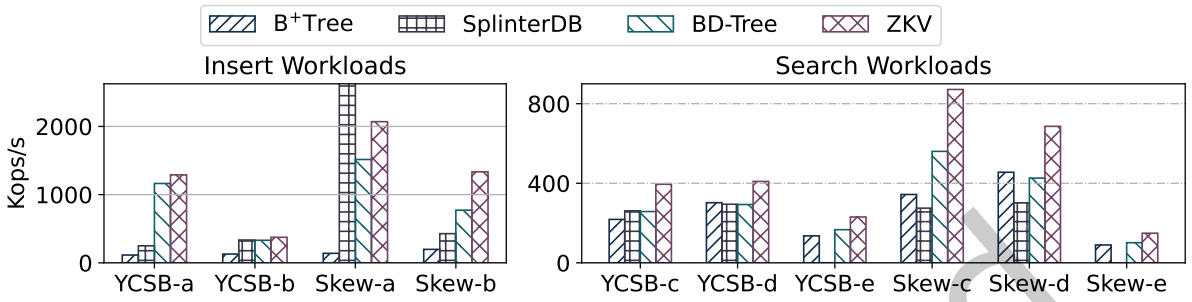

Fig. 11. The YCSB Uniform (YCSB) and YCSB-Skewed (Skew) benchmarks with B+-Tree-based KVSs.

for the two workloads in *column a*. Unless otherwise stated, we use the following method for evaluation. For all YCSB workloads, we first load 200 million kv-pairs into the B+-Tree index and then execute 200 million operations to perform the run-phase evaluation. For the classic B+-Tree in our evaluation, inserting 200M kv-pairs will generate approximately 1,300GB of write traffic on ZNS SSDs. To further evaluate the performance of Z+-Tree, we use three realistic read-only datasets obtained from SOSD [54]. These datasets include (1) AMZN containing book popularity data from Amazon, (2) WIKI containing timestamps of edits from Wikipedia, and (3) FACEBOOK containing randomly sampled Facebook user IDs. Three data sets contain a maximum of 100M kv-pairs. All the datasets use 8B keys and 8B values to verify the write amplification optimization. In the variable-length kv-pairs scenario, we insert the pointer to the value as "8B-value", ensuring uniform indexing performance.

4.2 Overall Performance

Exp#1 YCSB Benchmarks. All the workloads in Figure 11 are evaluated on 56 threads. Under the YCSB-a workload, ZKV's insert performance is the best of all B+-Tree indexes and outperforms BD-Tree by 1.15x, owing to its ability to aggregate kv-pairs through the three-level buffer structures. Under the read-write hybrid workloads YCSB-b, all of B+-Tree-based KVSs suffer from severe concurrent write conflicts. Under Skew-a loads, Z+-Tree achieves much higher throughput improvement over other B+-Tree indexes due to much less write traffic on ZNS SSDs except for SplinterDB. This is because it directly aggregates and flushes the memtable without triggering a compaction operation. Under Skew-b loads, ZKV achieved a better performance of up to 3.12x compared with SplinterDB. For point-search YCSB-d and scan YCSB-e workloads, ZKV is 1.35x/1.69x than that of the classic B+-Tree, thanks to its avoidance of the file system overheads and utilizing efficient read-cache. Under skewed search workloads, ZKV performs better than B+-Tree and BD-Tree by up to 2.53x, through a three-level buffer structure to more fully aggregate kv-pairs. The classic B+-Tree shows very little improvements because of the poor effects of their conventional buffer pool with an LRU-like cache policy. In our scan query with a maximum length of 256, SplinterDB is only about a few thousand IOPS, much lower than other B+-Tree-based KVSs. This is mainly because of its size-tiering's inherent disadvantage, which means that scans must search every branch along the root-to-leaf path to the starting key. The scan throughput of all indexes decreases under the skewed YCSB-e benchmark mainly because of the lock contention.

4.3 Effects of All Methods

Exp#2 Tail latency analysis. Write and read latency are evaluated on the insert-only YCSB-a workload and the read-only YCSB-d workload, respectively, and both are running on 56 threads. ZKV demonstrates up to two orders of magnitude lower 99^{th} percentile tail latency than other B+-Tree-based KVSs in Figure 12-a. However, ZKV exhibits a slightly higher 99.9^{th} percentile write tail latency due to the strict write order inside a zone. And ZKV shows 3.48x lower 99.99^{th} write tail latency than SplinterDB, which is defined as the baseline tail latency

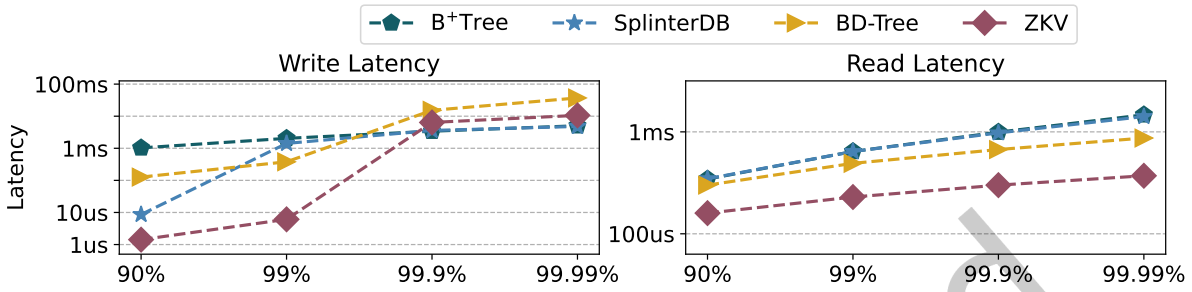

Fig. 12. (a).Write (b). Read latency in different percentiles.

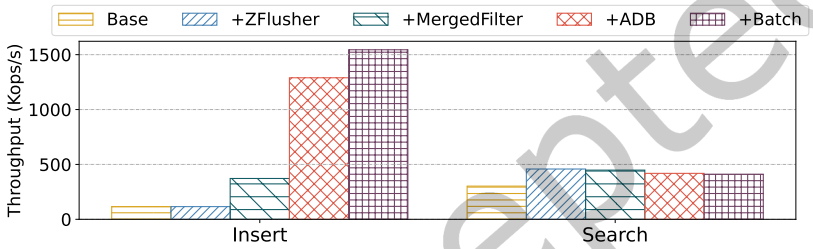

Fig. 13. Break Down of Optimizations.

divided by the ZKV's tail latency. The slightly higher 99.9^{th} percentile write tail latency for Z$^+$-Tree is acceptable, given the overall performance improvement. The write fluctuation inside the flash memory mainly affects this part of the long tail delay. Figure 12-b shows that ZKV reduces 99.99^{th} read tail latency by up to 3.96x. This is because ZKV removes the additional file system overhead.

Exp#3 Read/Write amplification. We measure the amount of read/write bytes per operation on the flash under the YCSB-a and YCSB-d workload. As Table 3 shows, the bytes written by ZKV are much smaller than those of B$^+$-Tree and BD-Tree, because the three-level buffer structure fully reduces write amplification and avoids additional writes to the file system. SplitnerDB uses a flush mechanism similar to LSM-Tree, so although it has lower write amplification, it causes more severe read amplification and makes scan operation very slow. It is worth noting that the BD-Tree we implemented avoids accessing delta records from flash memory, making its read bytes is similar to the classic B$^+$-Tree.

Table 3. IO amplification measured with *iostat* tool.

	B$^+$-Tree	SplinterDB	BD-Tree	ZKV
Write bytes/op	**6503.99**	132.19	664.57	222.27
Read bytes/op	3686.55	**5179.91**	3729.61	3724.86

Exp#4 Performance Break Down. Figure 13 shows the performance improvements attributed to each enabling technique or optimization in the Z$^+$-Tree index. It is evaluated on the 100M YCSB-a and YCSB-d workloads. *Base* is the base implementation of the B$^+$-Tree which runs on the F2FS-ZNS. As the figure shows, *+ZFlusher* improves the read throughput over the base by 55%, avoiding the additional FS layer access overheads. *+MergedFilter* buffers small-write kv-pairs to reduce write traffic, improving write performance by up to 3.21x. *+ADB* further improves write performance by up to 3.72x because it can fully aggregate kv-pairs to reduce write amplification. *+BatchMerge* inserts an average of 150 kv-pairs in a batch, which needs less flush operations and improves the write throughput of *+ADB* by 1.19x. With the introduction of the merged filter and ADB, read performance dropped by about 7.3%, but this will be exchanged for a significant improvement in write performance.

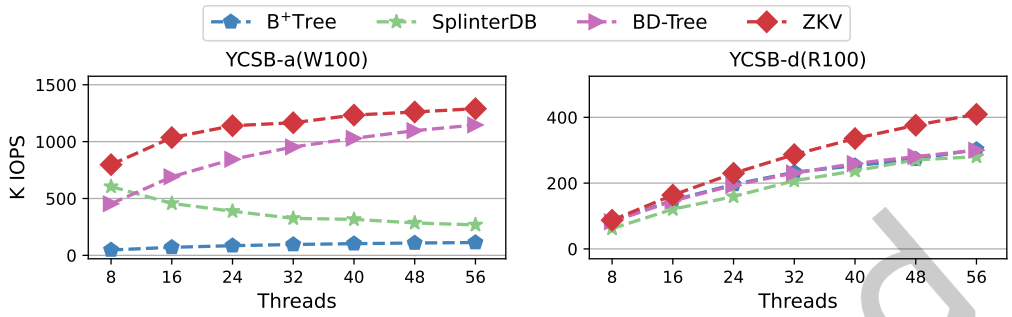

Fig. 14. Sensitive study on different threads.

4.4 Sensitivity Study

Exp#5 Concurrency Scaling. As shown in Figure 14, the insertion throughput of ZKV outperforms the B$^+$-Tree up to 11.4x and other two KVs up to 4.81x under 56 threads. The main performance gain is from its three-level buffer structure, significantly reducing write amplification on zone interfaces. SplinterDB has serious concurrency issues; as the number of threads increases, its write performance gradually decreases. In workload (YCSB-d), the performance of all B$^+$-Tree-based KVSs is scaled, but ZKV can still outperform others by up to 1.46x. This is mainly due to reducing the FS layer overheads and the read-efficient merged filter and ADB structures.

Exp#6 Buffersize. As shown in Figure 15, we gradually increased the ADB pool size from 50MB to 200MB with a total buffer size of 400MB. The write IOPS increased from 202.74 Kops/s to 1293.85 Kops, and the write amplification also stabilized at 221.32 bytes per insertion when ADB pool size is larger than 200MB. This change does not impact the random read performance in the YCSB-Uniform scenario. We selected an ADB pool size of 150MB to achieve both good read and write performance.

Exp#7 Memory Budget. We have added performance evaluations of ZKV under varying memory budgets. The three-leveled buffer architecture of ZKV, comprising the Merged Filter, ADB, and ZFlusher, adopts a fixed memory allocation ratio of 1:3:4 consistent with previous experiments. As shown in Figure 16, the total memory budget is increased incrementally from 200 MB to 1000 MB. Write performance peaks at 600 MB and gradually stabilizes around 1.5 million operations per second, approaching the throughput of the in-memory version. During this process, write amplification decreases significantly, from 491.74 bytes per insert at 200 MB to 187.26 bytes per insert at 600 MB. This improvement stems from enhanced buffering capacity; however, further memory increases yield diminishing returns, as in-memory operations begin to dominate the performance bottleneck. Interestingly, read throughput under random point queries improves only modestly, rising from 390.62 KOPS to 438.12 KOPS, a 12.2% increase as memory grows. This limited gain is expected, as most queries still incur high-latency flash random reads, which are less sensitive to increased memory availability.

Exp#8 Variable-length value. As shown in Figure 17, ZKV's throughput decreases from 1,280.64 Kops/s to 950.83 KOPS/s as the value size increases. This decline is primarily attributed to increased write-ahead log (WAL) overhead: larger kv-pairs introduce more internal fragmentation, raising the average write size per insertion from 201.73 to 325.66 bytes. While using a coarser WAL cache granularity could alleviate this issue, it may increase the risk of data loss during a crash. Despite the increase in value size from 8 to 512 bytes, ZKV maintains stable point and range query performance, approximately 200K and 120K IOPS, respectively (see Figure 17). This is achieved by separating variable-length values from the leaf nodes, allowing ZKV to preserve a uniform query path. Consequently, reading a variable-length value incurs only one additional flash access, avoiding a significant impact on overall read performance.

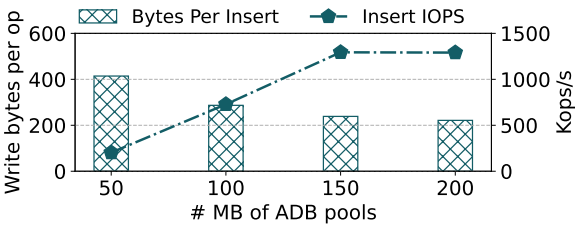

Fig. 15. The sensitive test on various size of ADB size.

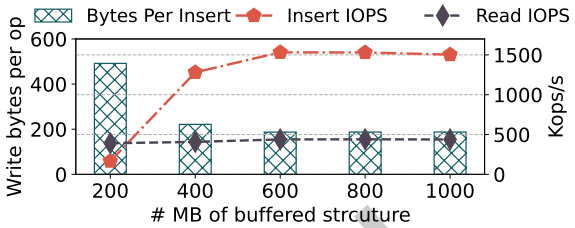

Fig. 16. The sensitive test on various sizes of three-leveled buffered structure.

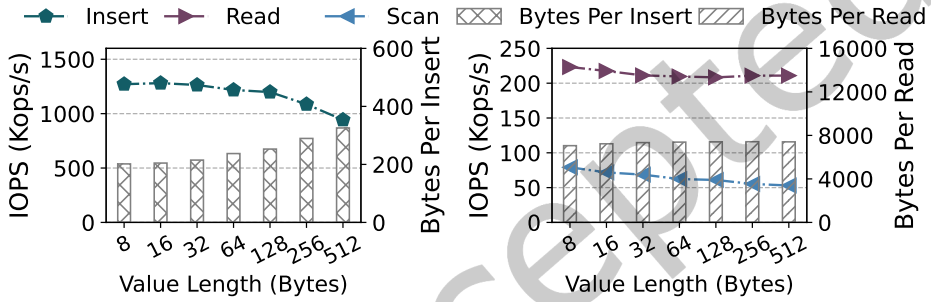

Fig. 17. The sensitive test on various value lengths.

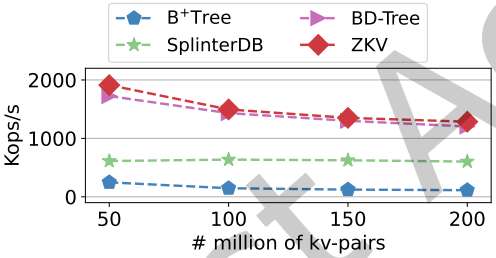

Fig. 18. Sensitive study on data size.

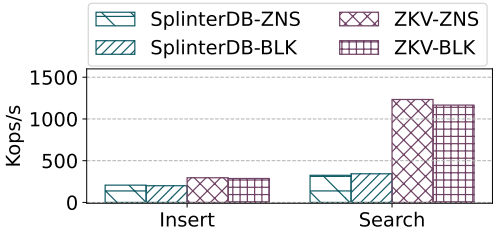

Fig. 19. Throughput on block SSD device.

Table 4. Sensitive study on different leaf node sizes.

	4KB	8KB	16KB	32KB
Write KIOPS/s	1,530.81	1,477.43	1,109.09	723.55
Read KIOPS/s	419.25	319.00	221.59	104.37
Write bytes/op	188.3	316.16	639.63	1,269.31
Read bytes/op	3,681.10	7,119.45	14,474.09	29,168.23

Exp#9 Datasize. Figure 18 shows the insert throughput of ZKV converges to 1257.30 Kops/s with workload size grows, while BD-Tree/SplinterDB converges to 1154.81/602.86 Kops/s. All KVSs show stable insertion performance, but ZKV shows a slightly higher performance than others.

Exp#10 NodeSize. As shown in Table 4, the amplification increases linearly with the leaf data page size, resulting in a linear decrease in both of write and read throughput. Although the variable-length value of kv-pairs is not our main focus, we evaluate its performance impact on the index. The insertion performance is not sensitive to this parameter because Z$^+$-Tree can fully utilize the zone interface.

Exp#11 Block SSD. We use a 4TB SN640 SSD as the block SSD, which was developed on similar hardware to the ZN540 SSD. ZKV-BLK and SplinterDB are evaluated on F2FS on SN640 with 56 threads. When running on the block-interface-based SN640 SSD, ZKV uses a conventional LRU-based buffer pool rather than the ZFlusher

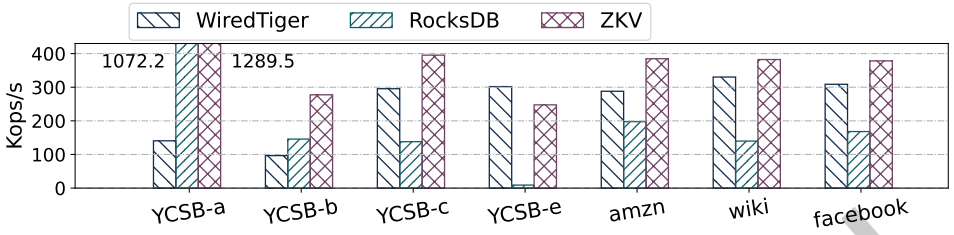

Fig. 20. The throughput of ZKV compared to two KV store engines under YCSB and three real-world workloads.

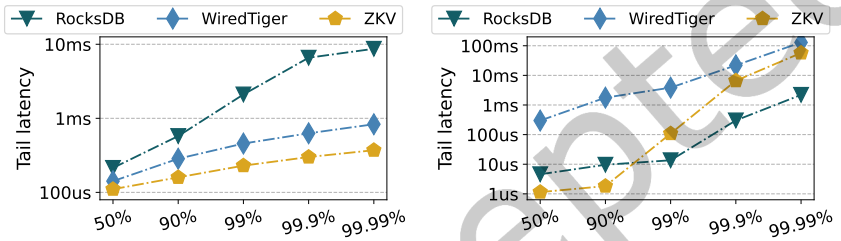

Fig. 21. (a) read latency under read-Only (YCSB-d); (b) Write latency under write-heavy (YCSB-b) workloads

component. As shown in the Figure 19, ZKV-BLK outperforms SplinterDB on the SN640 device, achieving 1.43× higher read throughput and 3.40× higher write throughput. The improvement in read performance is primarily attributed to ZKV's three-tiered buffering design, which preserves the efficient read characteristics of B^+-Tree structures. In contrast, SplinterDB incurs greater read overhead due to its LSM-tree-inspired design, which requires multiple read operations during query processing. Although the SN640 offers significantly higher 4KB random read and write IOPS than the ZNS 540 SSD, ZKV still demonstrates slightly greater performance gains on the ZNS device. This highlights the effectiveness of ZKV's design in fully leveraging the characteristics of ZNS SSDs.

4.5 Real-World System Evaluation

Exp#12 Real-Applications with Realistic Datasets. To further assess the performance of ZKV, we compare it against two production-grade KV storage engines, LSM-Tree-based RocksDB and B^+-Tree-based WiredTiger. As shown in the Figure 20, ZKV outperforms WiredTiger by 6.80× in write performance and is slightly higher than RocksDB in YCSB-a workloads. Under mixed loads, ZKV delivers higher throughput by effectively combining the strengths of both indexing approaches. RocksDB, lacking a global order guarantee, exhibits notably poor search performance, especially in the scan-dominated YCSB-e workload. In contrast, ZKV demonstrates comparable read performance to WiredTiger and significantly outperforms RocksDB in three real-world workloads. We also evaluate the read and write latency under the two different workloads. All workloads are generated by YCSB, all of which are 4M 16-byte kv-pairs. As shown in Figure 21, the read latency of ZKV is slightly more stable than WiredTiger and is much lower than RocksDB. The 90^{th} tail write latency of ZKV is one to two orders of magnitude lower than that of RocksDB. The write tail latency of ZKV and WiredTiger after the 90^{th} percentile is two orders of magnitude higher than RocksDB due to the modify-after-read operation that needs to ensure global order guarantee. However, ZKV's write latency is still lower than WiredTiger's due to the elimination of additional overheads on the FS layer through a more efficient tree structure.

5 Related works

LSM-Tree-based KVSs on ZNS SSDs. The LSM-Tree index exhibits sequential write patterns, making it well-suited for ZNS SSDs. Consequently, the LSM-Tree based storage engines have attracted the attention of many research works [16, 23, 28, 40, 48, 51, 53, 56, 73, 74]. Among these works, ZENFS is the first to propose a lightweight file system for RocksDB, providing basic file interface semantics. Subsequent research has focused on further improving zone management, garbage collection, and other functions. In its current form, ZKV not only addresses the write amplification problem of B^+-Tree through the zone interface but also retains, and even enhances the superior read performance of B^+-Tree. ZKV goes further by removing the file system and writing directly to the raw ZNS SSDs device. The future direction of ZKV may include high-performance garbage collection between zones. Additionally, using an approach similar to DedupKV [36], which performs deduplication at the KVS layer to reduce write amplification, is not suitable for B^+-Tree –based KVs on ZNS SSDs. Since ZKV is primarily designed for small-sized kv-pairs, introducing deduplication would incur substantial metadata overhead and additional access latency. Moreover, deduplication further leads to higher GC overhead on ZNS SSDs.

Other Storage Systems on ZNS SSDs. There are currently other storage systems based on ZNS SSDs, such as swap systems [22], RAID systems [38, 68, 76], cloud local disks [81], storage optimization of F2FS file system [50, 60, 61]. Some research works focus on optimizing the internal mechanism of ZNS SSDs in a simulator environment, such as dynamically adjustable namespace based on zone [31], smaller zone size to reduce read and write conflict interference [55, 82]. These works mainly focus on making full use of the zone interface features to improve the existing storage performance, which is orthogonal to the host-side ZKV.

B^+-Tree-based KVSs on Hard Drive Disk. SW-B^+Tree [46] is a B^+-Tree variant designed to optimize write amplification on shingled magnetic recording (SMR) Hard Drive Disks (SMR-HDDs). It improves performance by isolating frequently updated nodes, such as leaf nodes and hot internal nodes, into separate SMR zones. This design minimizes data migration during zone recovery, thereby reducing write amplification. However, small-sized random kv-pairs still cause a large amount of write amplification in SW-B^+Tree, because a small-sized kv-pair will result in a 4KB write on average. XS-B^+Tree [47], built on the SW-B^+Tree design, targets the solid-state hybrid drive (SSHD), a SMR HDD paired with a small flash cache. It accelerates B^+-Tree search operations by storing frequently accessed but rarely updated "cold" nodes (i.e., internal nodes) in the high-speed flash cache. Different from XS-B^+Tree, ZKV can keep hot data in lower-latency memory through the merged filter mechanism and avoid excessive memory usage through efficient eviction. XS-B^+Tree also needs an intricate in-memory hash mapping table to keep the correctness of the B^+-Tree as its internal and external nodes are maintained in different storage spaces. In contrast to these SMR-optimized schemes, ZKV focuses on flash-based zoned storage by employing three levels of in-memory buffers. These buffers aggregate small kv-pairs into large, sequential blocks, improving write efficiency and reducing flash-level write amplification.

6 Conclusion

ZKV proposes an optimized Z^+-Tree for ZNS SSDs, which reduces write amplification while retaining stable read latency. Z^+-Tree utilizes a three-level buffer structure to convert small-size random write operations into flash-friendly large-block append operations. Our evaluation shows that ZKV significantly reduces the write amplification while providing more stable read latency than current buffered B^+-Tree indexes on ZNS SSDs, and obtains the best of two worlds compared against two industrial storage engines, LSM-Tree based RocksDB and B^+-Tree based WiredTiger.

References

[1] 2024. btrfs | Zoned Storage. https://zonedstorage.io/docs/filesystems/btrfs
[2] 2023. dm-zap. https://github.com/westerndigitalcorporation/dm-zap original-date: 2020-12-18T01:41:53Z.
[3] 2024. f2fs | Zoned Storage. https://zonedstorage.io/docs/filesystems/f2fs

[4] 2024. index-microbench/BTreeOLC at master · wangziqi2016/index-microbench. https://github.com/wangziqi2016/index-microbench/tree/master/BTreeOLC

[5] 2024. Inspur Launches a New Generation of Enterprise NVMe SSD. https://www.hpcwire.com/off-the-wire/inspur-launches-a-new-generation-of-enterprise-nvme-ssd/

[6] 2024. MySQL :: MySQL 8.0 Reference Manual :: 1 General Information. https://dev.mysql.com/doc/refman/8.0/en/introduction.html

[7] 2022. NVMe Zoned Namespaces (ZNS) Command Set Specification - NVM Express. https://nvmexpress.org/specification/nvme-zoned-namespaces-zns-command-set-specification/

[8] 2024. NVMe Zoned Namespaces (ZNS) Devices | Zoned Storage. https://zonedstorage.io/docs/introduction/zns

[9] 2024. RocksDB. https://github.com/facebook/rocksdb/wiki/

[10] 2024. Samsung Introduces Its First ZNS SSD With Maximized User Capacity and Enhanced Lifespan. https://semiconductor.samsung.com/news-events/news/samsung-introduces-its-first-zns-ssd-with-maximized-user-capacity-and-enhanced-lifespan

[11] 2024. vmware/splinterdb: High Performance Embedded Key-Value Store. https://github.com/vmware/splinterdb

[12] 2024. Western Digital Ultrastar DC ZN540. https://www.westerndigital.com/en-ae/products/internal-drives/ultrastar-dc-zn540-nvme-ssd. https://www.westerndigital.com/en-ae/products/internal-drives/ultrastar-dc-zn540-nvme-ssd

[13] 2025. What is the NVM Express® Flexible Data Placement (FDP)? | SNIA. https://www.snia.org/educational-library/what-nvm-express%C2%AE-flexible-data-placement-fdp-2023

[14] 2024. WiredTiger Storage Engine - MongoDB Manual v7.0. https://www.mongodb.com/docs/manual/core/wiredtiger/

[15] 2025. Write-ahead logging - Wikiwand. https://www.wikiwand.com/en/articles/Write-ahead%20logging

[16] 2023. ZenFS: RocksDB Storage Backend for ZNS SSDs and SMR HDDs. https://github.com/westerndigitalcorporation/zenfs original-date: 2021-02-23T09:21:43Z.

[17] 2024. Zoned Namespaces (ZNS) SSDs: Disrupting the Storage Industry | SNIA. https://www.snia.org/educational-library/zoned-namespaces-zns-ssds-disrupting-storage-industry-2020

[18] 2024. Zoned Storage SSDs. https://www.westerndigital.com/solutions/zns-ssd

[19] 2024. [Webinar] Memory-Semantic SSD™. https://semiconductor.samsung.com/us/news-events/tech-blog/webinar-memory-semantic-ssd

[20] Jens Axboe. 2022. Flexible I/O Tester. https://github.com/axboe/fio original-date: 2012-10-22T08:20:41Z.

[21] Michael A Bender, Br Adley C Kuszmaul, Donald E Porter, Jun Yuan, and Yang Zhan. [n. d.]. An Introduction to Be-trees and Write-Optimization. ([n. d.]).

[22] Shai Bergman, Niklas Cassel, Matias Bjørling, and Mark Silberstein. 2022. {ZNSwap}: {un-Block} your Swap. 1–18. https://www.usenix.org/conference/atc22/presentation/bergman

[23] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. 2021. {ZNS}: Avoiding the Block Interface Tax for Flash-based {SSDs}. 689–703. https://www.usenix.org/conference/atc21/presentation/bjorling

[24] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. 49–63. https://www.usenix.org/conference/atc20/presentation/conway GSCC: 0000069.

[25] Brian Cooper. 2022. YCSB. https://github.com/brianfrankcooper/YCSB original-date: 2010-04-19T20:52:11Z.

[26] Diego Didona, Nikolas Ioannou, Radu Stoica, and Kornilios Kourtis. 2020. Toward a better understanding and evaluation of tree structures on flash SSDs. 14, 3 (2020), 364–377.

[27] Krijn Doekemeijer, Nick Tehrany, Balakrishnan Chandrasekaran, Matias Bjørling, and Animesh Trivedi. 2023. Performance Characterization of NVMe Flash Devices with Zoned Namespaces (ZNS). In 2023 IEEE International Conference on Cluster Computing (CLUSTER) (2023-10). 118–131. doi:10.1109/CLUSTER52292.2023.00018 0 citations (Crossref) [2024-04-29] ISSN: 2168-9253.

[28] Krijn Doekemeijer, Nick Tehrany, Zebin Ren, and Animesh Trivedi. 2024. ZWAL: Rethinking Write-ahead Logs for ZNS SSDs with Zone Appends. (2024).

[29] Jin Yong Ha and Heon Young Yeom. 2023. zCeph: Achieving High Performance On Storage System Using Small Zoned ZNS SSD. In Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing (New York, NY, USA, 2023-06-07) (SAC '23). Association for Computing Machinery, 1342–1351. doi:10.1145/3555776.3577758 1 citations (Crossref) [2024-05-09].

[30] Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, and How to Exploit it: High-Performance I/O for High-Performance Storage Engines. In Proceedings of the VLDB Endowment, Vol. 16. 2090–2102. doi:10.14778/3598581.3598584 GSCC: 0000015 4 citations (Crossref) [2024-05-21].

[31] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. 2021. ZNS+: Advanced Zoned Namespace Interface for Supporting In-Storage Zone Compaction. 147–162. https://www.usenix.org/conference/osdi21/presentation/han

[32] Xiangpeng Hao and Badrish Chandramouli. 2024. Bf-Tree: A Modern Read-Write-Optimized Concurrent Larger-Than-Memory Range Index. Proc. VLDB Endow. 17, 11 (2024), 3442–3455. doi:10.14778/3681954.3682012 GSCC: 0000000 0 citations (Crossref) [2024-09-03].

[33] Dingze Hong, Jinlei Hu, Jianxi Chen, Dan Feng, and Jian Liu. 2024. Optimizing Structural Modification Operation for B+-Tree on Byte-Addressable Devices. In 2024 IEEE 42nd International Conference on Computer Design (ICCD). 231–238. doi:10.1109/ICCD63220.

2024.00043 ISSN: 2576-6996.

[34] Dong Huang, Dan Feng, Qiankun Liu, Bo Ding, Wei Zhao, Xueliang Wei, and Wei Tong. 2023. SplitZNS: Towards an Efficient LSM-Tree on Zoned Namespace SSDs. 20, 3 (2023), 45:1–45:26. doi:10.1145/3608476 0 citations (Crossref) [2023-10-09].

[35] Joo-Young Hwang, Seokhwan Kim, Daejun Park, Yong-Gil Song, Junyoung Han, Seunghyun Choi, Sangyeun Cho, and Youjip Won. 2024. {ZMS}: Zone Abstraction for Mobile Flash Storage. 173–189. https://www.usenix.org/conference/atc24/presentation/hwang

[36] Safdar Jamil, Awais Khan, Xubin He, and Youngjae Kim. 2025. DEDUPKV: A Space-Efficient and High-Performance Key-Value Store via Fine-Grained Deduplication. In Proceedings of the 39th ACM International Conference on Supercomputing (New York, NY, USA, 2025-08-22) (ICS '25). Association for Computing Machinery, 580–595. doi:10.1145/3721145.3730424

[37] Axboe Jens. 2025. 1. fio - Flexible I/O tester rev. 3.38 — fio 3.38-19-gd1eb documentation. https://fio.readthedocs.io/en/latest/fio_doc.html#

[38] Thomas Kim, Jekyeom Jeon, Nikhil Arora, Huaicheng Li, Michael Kaminsky, David G. Andersen, Gregory R. Ganger, George Amvrosiadis, and Matias Bjørling. 2023. RAIZN: Redundant Array of Independent Zoned Namespaces. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (New York, NY, USA, 2023-01-30) (ASPLOS 2023). Association for Computing Machinery, 660–673. doi:10.1145/3575693.3575746

[39] Hee-Rock Lee, Chang-Gyu Lee, Seungjin Lee, and Youngjae Kim. 2022. Compaction-aware zone allocation for LSM based key-value store on ZNS SSDs. In Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (New York, NY, USA, 2022-06-27) (HotStorage '22). Association for Computing Machinery, 93–99. doi:10.1145/3538643.3539743 20 citations (Crossref) [2024-07-17].

[40] Jongsung Lee, Donguk Kim, and Jae W. Lee. 2023. WALTZ: Leveraging Zone Append to Tighten the Tail Latency of LSM Tree on ZNS SSD. 16, 11 (2023), 2884–2896. doi:10.14778/3611479.3611495 0 citations (Crossref) [2023-08-31].

[41] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In 2018 IEEE 34th International Conference on Data Engineering (ICDE). 185–196. doi:10.1109/ICDE.2018.00026 ISSN: 2375-026X.

[42] 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. ACM, Huntsville Ontario Canada, 447–461. doi:10.1145/3341301.3359628 GSCC: 0000164.

[43] Justin Levandoski, David Lomet, and Sudipta Sengupta. 2013. LLAMA: a cache/storage subsystem for modern hardware. 6, 10 (2013), 877–888. doi:10.14778/2536206.2536215

[44] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In 2013 IEEE 29th International Conference on Data Engineering (ICDE) (2013-04). 302–313. doi:10.1109/ICDE.2013.6544834 ISSN: 1063-6382.

[45] Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, and Ke Yi. 2010. Tree indexing on solid state drives. 3, 1 (2010), 1195–1206. doi:10.14778/1920841.1920990 112 citations (Crossref) [2023-10-16].

[46] Yu-Pei Liang, Tseng-Yi Chen, Yuan-Hao Chang, Shuo-Han Chen, Kam-Yiu Lam, Wei-Hsin Li, and Wei-Kuan Shih. 2019. Enabling Sequential-write-constrained B+-tree Index Scheme to Upgrade Shingled Magnetic Recording Storage Performance. ACM Trans. Embed. Comput. Syst. 18, 5s (2019), 66:1–66:20. doi:10.1145/3358201

[47] Yu-Pei Liang, Tseng-Yi Chen, Ching-Ho Chi, Hsin-Wen Wei, and Wei-Kuan Shih. 2020. Enabling a B+-tree-based Data Management Scheme for Key-value Store over SMR-based SSHD. In 2020 57th ACM/IEEE Design Automation Conference (DAC). 1–6. doi:10.1109/DAC18072.2020.9218708 ISSN: 0738-100X.

[48] Biyong Liu, Yuan Xia, Xueliang Wei, and Wei Tong. 2023. LifetimeKV: Narrowing the Lifetime Gap of SSTs in LSMT-based KV Stores for ZNS SSDs. In 2023 IEEE 41st International Conference on Computer Design (ICCD). 300–307. doi:10.1109/ICCD58817.2023.00053

[49] Yang Liu and Peiquan Jin. 2023. ZB+-tree: A Novel ZNS SSD-Aware Index Structure. 60, 3 (2023), 509–524. doi:10.7544/issn1000-1239.202220502 Publisher: Journal of Computer Research and Development.

[50] LiuYachun, FengDan, ChenJianxi, HuJing, PengZhouxuan, and HuJinlei. 2025. ZNSFQ: an efficient and high-performance fair queue scheduling scheme for ZNS SSDs. ACM Transactions on Architecture and Code Optimization (Sept. 2025). doi:10.1145/3746230 GSCC: 0000000 Publisher: ACMPUB27New York, NY.

[51] Mingchen Lu, Peiquan Jin, Xiaoliang Wang, Yongping Luo, and Kuankuan Guo. 2023. ZoneKV: A Space-Efficient Key-Value Store for ZNS SSDs. In 2023 60th ACM/IEEE Design Automation Conference (DAC) (2023-07). 1–6. doi:10.1109/DAC56929.2023.10247926 1 citations (Crossref) [2024-05-09].

[52] Ziyi Lu, Qiang Cao, Hong Jiang, Yuxing Chen, Jie Yao, and Anqun Pan. 2024. FluidKV: Seamlessly Bridging the Gap between Indexing Performance and Memory-Footprint on Ultra-Fast Storage. Proc. VLDB Endow. 17, 6 (may 2024), 1377–1390. doi:10.14778/3648160.3648177

[53] Yanqi Lv, Peiquan Jin, Xiaoliang Wang, Ruicheng Liu, Liming Fang, Yuanjin Lin, and Kuankuan Guo. 2022. ZonedStore: A Concurrent ZNS-Aware Cache System for Cloud Data Storage. In 2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS) (2022-07). 1322–1325. doi:10.1109/ICDCS54860.2022.00148 0 citations (Crossref) [2023-07-04] ISSN: 2575-8411.

[54] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking learned indexes. Proc. VLDB Endow. 14, 1 (sep 2020), 1–13. doi:10.14778/3421424.3421425

[55] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. 2023. {eZNS}: An Elastic Zoned Namespace for Commodity {ZNS} {SSDs}. 461–477. https://www.usenix.org/conference/osdi23/presentation/min

[56] Gijun Oh, Junseok Yang, and Sungyong Ahn. 2021. Efficient Key-Value Data Placement for ZNS SSD. 11, 24 (2021), 11842. doi:10.3390/app112411842 3 citations (Crossref) [2023-07-04] Number: 24 Publisher: Multidisciplinary Digital Publishing Institute.

[57] 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 (New York, NY, USA, 2016-06-14) (SIGMOD '16). Association for Computing Machinery, 371–386. doi:10.1145/2882903.2915251

[58] Zhu Pang, Qingda Lu, Shuo Chen, Rui Wang, Yikang Xu, and Jiesheng Wu. 2021. ArkDB: A Key-Value Engine for Scalable Cloud Storage Services. In Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21). Association for Computing Machinery, New York, NY, USA, 2570–2583. doi:10.1145/3448016.3457553 GSCC: 0000012 2 citations (Crossref) [2024-03-27].

[59] Tarikul Islam Papon and Manos Athanassoulis. 2023. ACEing the Bufferpool Management Paradigm for Modern Storage Devices. In 2023 IEEE 39th International Conference on Data Engineering (ICDE) (2023-04). 1326–1339. doi:10.1109/ICDE55515.2023.00106 0 citations (Crossref) [2024-01-11] ISSN: 2375-026X.

[60] Wenjie Qi, Zhipeng Tan, Jicheng Shao, Lihua Yang, and Yang Xiao. 2022. InDeF: An Advanced Defragmenter Supporting Migration Offloading on ZNS SSD. In 2022 IEEE 40th International Conference on Computer Design (ICCD) (2022-10). 307–314. doi:10.1109/ICCD56317.2022.00052 2 citations (Crossref) [2024-05-09] ISSN: 2576-6996.

[61] Wenjie Qi, Zhipeng Tan, Ziyue Zhang, Jing Zhang, Chao Yu, Ying Yuan, and Shikai Tan. 2023. BlzFS: Crash Consistent Log-structured File System Based on Byte-loggable Zone for ZNS SSD. In 2023 IEEE 41st International Conference on Computer Design (ICCD) (2023-11). 206–213. doi:10.1109/ICCD58817.2023.00040 0 citations (Crossref) [2024-05-09] ISSN: 2576-6996.

[62] Yifan Qiao, Xubin Chen, Ning Zheng, Jiangpeng Li, Yang Liu, and Tong Zhang. 2022. Closing the B+-tree vs. {LSM-tree} Write Amplification Gap on Modern Storage Hardware with Built-in Transparent Compression. 69–82. https://www.usenix.org/conference/fast22/presentation/qiao

[63] Dongjoo Seo, Ping-Xiang Chen, Matias Bjørling, Huaicheng Li, and Nikil Dutt. 2023. Is Garbage Collection Overhead Gone? Case study of F2FS on ZNS SSDs. (2023).

[64] Ge Shi, Ziyi Yan, and Tianzheng Wang. 2023. OptiQL: Robust Optimistic Locking for Memory-Optimized Indexes. Proceedings of the ACM on Management of Data 1, 3 (2023), 216:1–216:26. doi:10.1145/3617336 GSCC: 0000004 0 citations (Crossref) [2024-01-11].

[65] Zhenhua Tan, Linbo Long, Jingcheng Shen, Congming Gao, Renping Liu, and Yi Jiang. 2024. Para-ZNS: Improving Small-Zone ZNS SSDs Parallelism Through Dynamic Zone Mapping. In 2024 Design, Automation & Test in Europe Conference & Exhibition (DATE). 1–6. doi:10.23919/DATE58400.2024.10546863 ISSN: 1558-1101.

[66] Nick Tehrany and Animesh Trivedi. 2022. Understanding NVMe Zoned Namespace (ZNS) Flash SSD Storage Devices. arXiv:2206.01547 [cs] doi:10.48550/arXiv.2206.01547

[67] Hao Wang, Jiaxin Ou, Ming Zhao, Sheng Qiu, Yizheng Jiao, Yi Wang, Qizhong Mao, Zhengyu Yang, Yang Liu, Jianshun Zhang, Jianyang Hu, Jingwei Zhang, Jinrui Liu, Jiaqiang Chen, Yong Shen, Lixun Cao, Heng Zhang, Hongde Li, Ming Li, Yue Ma, Lei Zhang, Jian Liu, Guanghui Zhang, Fei Liu, and Jianjun Chen. 2024. LavaStore: ByteDance's Purpose-Built, High-Performance, Cost-Effective Local Storage Engine for Cloud Services. Proc. VLDB Endow. 17, 12 (2024), 3799–3812. doi:10.14778/3685800.3685807

[68] Qiuping Wang and Patrick P. C. Lee. 2023. ZapRAID: Toward High-Performance RAID for ZNS SSDs via Zone Append. In Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems (New York, NY, USA, 2023-08-24) (APSys '23). Association for Computing Machinery, 24–29. doi:10.1145/3609510.3609810 0 citations (Crossref) [2024-03-05].

[69] Rui Wang, Xinjun Yang, Feifei Li, David B. Lomet, Xin Liu, Panfeng Zhou, Yongxiang Chen, David Zhang, Jingren Zhou, and Jiesheng Wu. 2024. Bwe-tree: An Evolution of Bw-tree on Fast Storage. In 2024 IEEE 40th International Conference on Data Engineering (ICDE). 5266–5279. doi:10.1109/ICDE60146.2024.00396 ISSN: 2375-026X.

[70] Xiaoliang Wang, Peiquan Jin, Yongping Luo, and Zhaole Chu. 2024. Range Cache: An Efficient Cache Component for Accelerating Range Queries on LSM - Based Key-Value Stores. In 2024 IEEE 40th International Conference on Data Engineering (ICDE). 488–500. doi:10.1109/ICDE60146.2024.00044 ISSN: 2375-026X.

[71] Yingjia Wang, You Zhou, Fei Wu, Jie Zhang, and Ming-Chang Yang. 2024. Land of Oz: Resolving Orderless Writes in Zoned Namespace SSDs. IEEE Trans. Comput. 73, 11 (Nov. 2024), 2520–2533. doi:10.1109/TC.2024.3441866 Conference Name: IEEE Transactions on Computers.

[72] 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 (Houston TX USA, 2018-05-27). ACM, 473–488. doi:10.1145/3183713.3196895

[73] Denghui Wu, Biyong Liu, Wei Zhao, and Wei Tong. 2022. ZNSKV: Reducing Data Migration in LSMT-Based KV Stores on ZNS SSDs. In 2022 IEEE 40th International Conference on Computer Design (ICCD) (2022-10). 411–414. doi:10.1109/ICCD56317.2022.00067 ISSN: 2576-6996.

[74] Denghui Wu, Biyong Liu, Wei Zhao, and Wei Tong. 2022. ZNSKV: Reducing Data Migration in LSMT-Based KV Stores on ZNS SSDs. In 2022 IEEE 40th International Conference on Computer Design (ICCD). 411–414. doi:10.1109/ICCD56317.2022.00067

[75] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. {LSM-trie}: An {LSM-tree-based} {Ultra-Large} {Key-Value} Store for Small Data Items. 71–82. https://www.usenix.org/conference/atc15/technical-session/presentation/wu

[76] Shushu Yi, Yanning Yang, Yunxiao Tang, Zixuan Zhou, Junzhe Li, Chen Yue, Myoungsoo Jung, and Jie Zhang. 2022. ScalaRAID: optimizing linux software RAID system for next-generation storage. In Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (New York, NY, USA, 2022-06-27) (HotStorage '22). Association for Computing Machinery, 119–125. doi:10.1145/3538643.3539740

[77] Bowen Zhang, Shengan Zheng, Zhenlin Qi, and Linpeng Huang. 2022. NBTree: a lock-free PM-friendly persistent B+-tree for eADR-enabled PM systems. Proceedings of the VLDB Endowment 15, 6 (2022), 1187–1200. doi:10.14778/3514061.3514066

[78] Jianshun Zhang, Fang Wang, and Chao Dong. 2022. HaLSM: A Hotspot-aware LSM-tree based Key-Value Storage Engine. In 2022 IEEE 40th International Conference on Computer Design (ICCD). 179–186. doi:10.1109/ICCD56317.2022.00035 ISSN: 2576-6996.

[79] Jianshun Zhang, Fang Wang, Sheng Qiu, Yi Wang, Jiaxin Ou, Junxun Huang, Baoquan Li, Peng Fang, and Dan Feng. 2024. Scavenger: Better Space-Time Trade-Offs for Key-Value Separated LSM-trees. In 2024 IEEE 40th International Conference on Data Engineering (ICDE) (2024-06-19). 4072–4085. doi:10.1109/ICDE60146.2024.00312 ISSN: 2375-026X.

[80] Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and K. V. Rashmi. 2024. {SIEVE} is Simpler than {LRU}: an Efficient {Turn-Key} Eviction Algorithm for Web Caches. 1229–1246. https://www.usenix.org/conference/nsdi24/presentation/zhang-yazhuo

[81] Yanbo Zhou, Erci Xu, Li Zhang, Kapil Karkra, Mariusz Barczak, Wayne Gao, Wojciech Malikowski, Mateusz Kozlowski, Łukasz Łasek, Ruiming Lu, Feng Yang, Lilong Huang, Xiaolu Zhang, Keqiang Niu, Jiaji Zhu, and Jiesheng Wu. 2024. CSAL: the Next-Gen Local Disks for the Cloud. In Proceedings of the Nineteenth European Conference on Computer Systems (New York, NY, USA, 2024-04-22) (EuroSys '24). Association for Computing Machinery, 608–623. doi:10.1145/3627703.3629566 0 citations (Crossref) [2024-05-30].

[82] Weilin Zhu and Wei Tong. 2023. Turn Waste Into Wealth: Alleviating Read/Write Interference in ZNS SSDs. In 2023 IEEE 41st International Conference on Computer Design (ICCD). 316–319. doi:10.1109/ICCD58817.2023.00055

Received 8 April 2025; revised 29 September 2025; accepted 11 November 2025