LodgeTree: A Last-Level Distributed and Surrogate Buffer Tree for Non-Volatile Memories

Chunfeng Du*¶, Shengzhe Wang*¶, Suzhen Wu*†✉, Hong Jiang‡,
Jiahong Chen*, Yingchao Ji*, Bo Mao*, Lvqing Yang*✉
*Xiamen Key Laboratory of Intelligent Storage and Computing, School of Informatics, Xiamen University, Xiamen, China
†Wuhan National Laboratory for Optoelectronics, Wuhan, China
‡Department of Computer Science and Engineering, The University of Texas at Arlington, Arlington, TX, USA

***Abstract*—Although the emergence of Non-Volatile Memory Techniques (NVMs) offers new insights into solving the problem of memory shortage, tree-based indexing systems face performance overhead when applied to the NVM due to the performance difference between DRAM and NVM. The utilization of buffer techniques exploiting DRAM can alleviate system performance overhead. However, when directly applied to the tree-based index system, it faces some challenges, such as high memory overhead, high flush overhead, and complicated log management. Meanwhile, NVM has some unique features, e.g., asymmetrical read/write, higher random read, and differential access granularity, that should also be carefully focused on in the design. Consequently, we propose LodgeTree, a last-level distributed and surrogate buffer tree design in B+-Tree-based index system. Specifically, LodgeTree fully utilizes the free space situation in the last level of internal nodes of B+ Tree to build a dynamic buffer and proposes three new techniques, including Leaf-count Flush, Hotness-Aware Multiply Split, and Partitioned Version Log, to reduce the memory space overhead, refresh overhead, and log management overhead, and to improve the system performance. Experimental evaluations show that LodgeTree can achieve up to 7.4× and 1.7× on average compared with other schemes in system throughput.**

I. Introduction

Emerging NVMs, with low latency, large capacity, and byte addressability, are consistently expected as alternatives to traditional volatile DRAM memory. Therefore, multiple existing studies call for the development of novel and efficient indexing data structures to effectively manage data in NVM-based applications [7], [11], [12], [20], [25], [29], [31]. Tree-based indexing is widely utilized in in-memory data centers and data-intensive applications due to its ability to deliver competitive read/write performance and support for range queries [4], [16], [17], [19], [24]. Additionally, because of the difference in the speed of write/read between DRAM and NVM, applying the tree-based index system directly to NVM devices will inevitably lead to performance degradation. Without loss of generality, We leverage Persistent Memory (PM) as a representative with NVM properties.

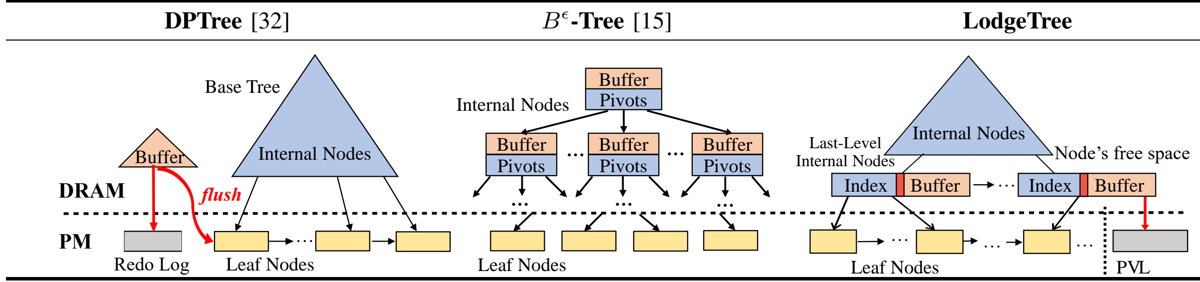
Recent studies have introduced various tree-based schemes tailored for PM, including FAST+FAIR [13], wB+Tree [8], and uTree [9]. However, these approaches treat PM as slower DRAM with non-volatile features, allowing incoming key-value pairs to be directly inserted into specific locations. Meanwhile, researchers have discovered that real PM exhibits specific characteristics in practice, e.g., high random read latency and larger access granularity than cacheline [6], [27], [28]. Furthermore, previous workload analysis have revealed that small writes dominate real-world workloads and exhibit strong locality [10]. The distinct characteristics of the device and data feature of applications have inspired researchers to redesign and improve their architecture. This effort is driven by the necessity to align with specific workload features and device capabilities. Current research emphasizes using buffer technology to enhance the performance of tree-based index systems [5], [15], [27], [32]. The buffer optimizes access patterns to minimize random read/write operations on PM devices while aligning with the CPU's cache line to support the larger access granularity inherent in PM systems.

The system-level buffer design usually temporarily allocates memory space to store data to enhance performance. Similarly, generic B-tree-based indexing systems utilize dedicated memory buffers to improve performance. Different buffer designs for the B-Tree-based index systems architecture are compared in Table I. For example, the DPTree [32] allocates a dedicated buffer space outside the base Tree. Incoming key-value pairs are first inserted into the buffer backed by a write-optimized redo log. The buffer is merged into the base tree when it reaches the size threshold. Moreover, previous studies have also proposed setups for constructing buffers for each internal node based on tree-based index structures. For instance, the traditional B^{ϵ}-Tree [5] utilizes the hierarchical structure of B-Tree and sets the buffers inside each internal node across multiple levels. Incoming key-value pairs are first inserted into the root node's buffer and then moved downward level by level with the flush operation.

This paper focuses on building ingenious buffer designs for B+-Tree tree structure-based NVM systems. While a single dedicated buffer, as seen in the traditional B+-tree index system (e.g., DPTree [32]), can effectively handle small writes and improve query performance, it also faces some challenges. First, the design of this buffer inherently increases additional memory overhead to the system. Second, it exhibits high flush

✉Corresponding Author: Suzhen Wu (suzhen@xmu.edu.cn) and Lvqing Yang (lqyang@xmu.edu.cn). ¶These authors have equal contributions.

TABLE I
COMPARISON OF DIFFERENT BUFFER DESIGNS FOR B-TREE-BASED INDEX SCHEMES.



overhead, potentially resulting in elevated tail latency and write stalls. Additionally, the multi-level buffer design, such as the traditional B^ϵ-Tree [5], usually assigns buffers to different levels of each internal node, allowing for fine-grained flush operations for each node. However, it also introduces complex Write-Ahead Logging (WAL) management and the potential for high cascade flushes.

Nevertheless, both approaches necessitate additional memory allocation to maintain the buffer design, which deviates from the original objective of leveraging NVM to expand the main memory capacity. Consequently, these two buffer-setting methods for NVMs within a B+ tree framework may not align with the intended purpose. Furthermore, our investigation revealed an untapped potential in the memory space allocated to the last-level internal nodes of a B+-Tree-based index.

To solve these challenges and better combine the advantages of the B+-tree-based structure's properties (i.e., internal node free space) and data characteristics, this paper proposed LodgeTree, a last-level internal-node distributed and dynamic surrogate buffer design for a B+-Tree-based system, shown in Section III. It can retain the advantages of the aforementioned two buffer designs and overcome their shortcomings by utilizing the unemployed space of last-level internal nodes. In summary, this paper makes the following contributions:

(1) The analysis of typical buffer design for the tree-based index system reveals some problems, such as memory space overhead, high tail latency, and complicated log management. Furthermore, experimental tests demonstrate that 25% - 58% of the memory space allocated to the internal nodes is unemployed in the B+-tree-based index, illustrated in Section II-C. Both inspire us to exploit the free space to construct the new buffer design for the B+-Tree-based index system.

(2) Taking the PM's features, typical buffer design, and the B+-tree internal node free memory space into consideration, this paper proposed the LodgeTree, illustrated in Section III, which mainly includes three critical techniques, e.g., *The Leaf Count-Based Flush*, *Hotness-Aware Multiple Split*, and *Partitioned Version Log*.

(3) Experimental evaluations conducted on the LodgeTree prototype demonstrate a remarkable enhancement in system throughput, with LodgeTree achieving a speedup of up to 7.4× and an average of 1.7×. Meanwhile, based on the NUMA architecture, experimental results demonstrate that our scheme outperforms existing schemes.

II. BACKGROUND AND MOTIVATION

A. Unique Characteristics of PM

Understanding the unique characteristics of NVM is crucial for designing effective storage systems. Although other NVM products exist, such as Spin Torque Transfer RAM (STT-RAM) and Resistive RAM (ReRAM) [1], [14], Intel Optane Persistent Memory (PM) remains the widely used commercial representative NVM. Besides its fundamental DRAM-like characteristics, the Optane PM also comprises some distinctive traits. First, the read/write granularity does not align with the typical cache line size of 64 bytes. A larger access granularity is needed to fully utilize PM bandwidth, prompting the exploration of buffer designs to facilitate the transfer of small accesses to a larger granularity. Second, research and white papers have demonstrated that PM exhibits higher media latency than DRAM, with random-read latency reaching 2.5 - 3 × that of DRAM [27], [28]. This disparity underscores the imperative of minimizing random access to PM. In a B+ tree with numerous leaf nodes, each access to a leaf node triggers a random read on PM. Consequently, each insertion into the B+ tree via point insertion incurs a costly random-read operation, culminating in elevated insert latency. This underscores the importance of considering buffer designs to mitigate the expensive overhead of random read/write. Certainly, with the development and advancement of future generations of the latest PM technologies, it is hopeful that general-purpose PM will ultimately be able to address the limitations currently faced by Optane. Our research primarily centers on Intel Optane PM [2] as a key platform for conducting various research explorations in this paper.

B. Buffer Designs for B-Tree-based Schemes

Integrating buffers is instrumental in enhancing the performance of B-tree-based indexing systems [5], [15], [32]. However, this also will encounter some challenges. The typical buffer design schemes can be broadly categorized into two main types in this paper, such as the *Single Buffer Mechanism (SBM)* and the *Multiple Buffer Mechanism (MBM)*.

1) Single Buffer Mechanism (SBM): The SBM's design usually leverages a single dedicated buffer (in DRAM) to speed up write requests with the DRAM-PM layout. Next, we take DPtree as an example to analyze the problems and challenges in detail. DPTree [32] designs two levels of B+-Tree. The first is the Buffer Tree in DRAM, which uses

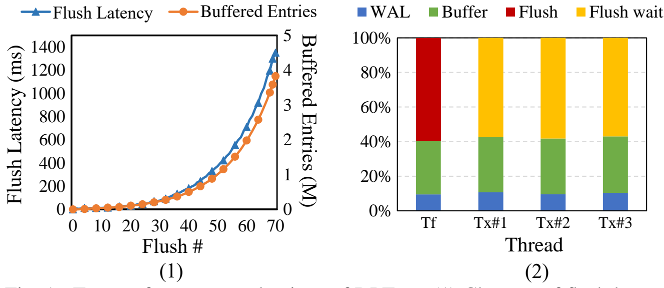


Fig. 1. Two performance evaluations of DPTree: (1) Changes of flush latency and number of buffered entries with increased flushes. (2) Breakdown of the insert time in DPTree with 1 flush thread (Tf) and 3 execution threads (Tx1, Tx2, and Tx3).

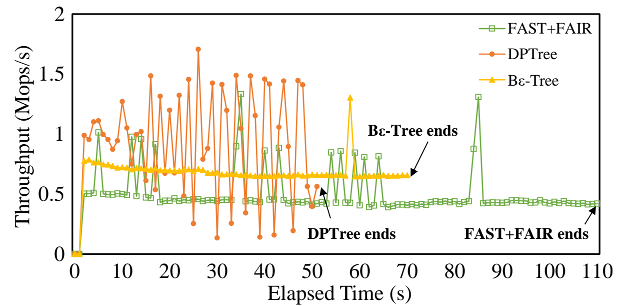


Fig. 2. Single-threaded write performance of DPTree (single-level centralized buffer), B^{ϵ}-Tree (multiple-level distributed buffer) and FAST+FAIR (point insertion) with 50M key-value pairs.

a write-optimized PM redo log to ensure crash consistency. When the Buffer Tree reaches a predefined threshold, DPTree will activate the background threads to flush the key-value pairs in the Buffer Tree's leaf node layer into the leaf nodes of the Base Tree in PM. Base Tree uses a DRAM-PM layout, with internal nodes in DRAM and leaf nodes in PM.

Employing dedicated memory space as a buffer can bring some performance gains. However, it also faces some challenges, as follows. (1) **High memory space overhead**: Based on the default setting of DPTree [32], with the continuous growth of the Buffer Tree, the occupied memory space gradually increases. Figure 1(1)(Y-axis on the right) shows that the number of buffered entries continues to increase with the number of flush times. Though the threshold in DPTree can limit the buffer size, the additional memory space consumed remains to be taken seriously. (2) **High flush overhead**: DPTree needs to flush the buffered data in the Buffer Tree to the leaf nodes of the Base Tree. As the Buffer Tree grows, the flush operation needs significant latency overhead and can lead to long tail latency. Figure 1(1)(Y-axis on the left) shows the latency of each flush during the process of inserting 50M key-value pairs. As the number of flush times increases and the Buffer Tree becomes larger, the latency of the flush operation gradually increases, from 9ms to an astonishing 1.4s. Meanwhile, it is also why DPTree's throughput fluctuates so widely, as shown in Figure 2.

Furthermore, substantial flushes overhead will block other threads in a multi-threaded environment, leading to more significant results, i.e., write stalls. Figure 1(2) shows a breakdown of the insert time into the four sections: write-ahead log (WAL), insert Buffer Tree (Buffer), flush (for flush threads), and wait for flush (flush wait, for execution threads), with 1 flush thread and 3 execution threads. It is important to note that the waiting time of execution threads accounted for about 60% of the total time, meaning that these threads spend the majority of time waiting for the flush to complete.

2) Multiple Buffer Mechanism (MBM): MBM can usually provide small buffers for multiple internal nodes in the B-tree-based index system. Next, we take the B^{ϵ}**-Tree** as an example to analyze these problems and challenges in detail. B^{ϵ}-Tree is designed for HDDs [5] and can be optimized for commodity SSDs [15]. It splits an internal node into two sections: the pivots part, which stores the index pointers, and the buffer part, which stores the buffered key-value pairs.

The MBM's design can provide finer-grained flush operation to avoid long flush latency, as shown in Figure 2. However, it also faces challenges. (1) **Complicated log management**: After flushing the buffer to the PM leaf nodes, the corresponding log space can be released, but in the case of B^{ϵ}-Tree, the distributed buffers in each internal node result in inconsistent log invalidation times. Moreover, the MBM forces the buffered data through multiple layers to reach the leaf nodes in PM, leading to longer WAL. (2) **High cascade flush overhead**: When B^{ϵ}-Tree flushes buffered data downwards, and the targeted lower node is also complete, B^{ϵ}-Tree will trigger a cascade flush. A key-value pair insertion may trigger multiple level nodes' flush and structural adjustment operations (e.g., node split), leading to long tail latency.

Furthermore, the concurrency splitting of multiple internal nodes also contributes to the extended tail latency. Given that PM leaf nodes are at the lowest level, they experience more frequent splits. When an internal node flushes data to numerous leaf nodes, it can lead to the concurrent splitting of multiple leaf nodes, resulting in prolonged tail latency. This concurrent splitting phenomenon adds to the overall latency, impacting system performance and responsiveness.

C. Motivation

The buffer design can help mitigate performance discrepancies in the Tree-based index system arising from the distinct characteristics of different devices. However, the Tree-based index NVM system must carefully consider challenges such as memory space overhead, flush overhead, and complex WAL management. Structural analysis and experimental results reveal that last-level internal nodes in B+-Tree-based indexing structures have a significant amount of empty slots, ranging from 25% to 58%, which aligns with previous studies [13]. Meanwhile, the existence of unoccupied slots also inspires us to take the free space into special consideration as a temporary buffer for leaf nodes. Recognizing the importance of buffer design in a B+-Tree-based NVM system, we proposed LodgeTree. This dynamic and distributed surrogate buffer design leverages the unused space in last-level internal nodes. By combining the performance characteristics of PM with the advantages of B+-Tree structure characteristics, the LodgeTree scheme can reduce flush overhead, mitigate complex log management, and thus improve system performance.

III. Design of LodgeTree

A. LodgeTree Overview

LodgeTree, designed to uphold the advantages of buffers in B+ tree-based indexing systems while minimizing memory overhead and write flush overhead, showcases an architectural overview in Figure 3. This overview exemplifies the creation of a single-level distributed buffer that cleverly utilizes the available free space in the last-level internal nodes. Initially, incoming key-value pairs find their way into the Partitioned Version Log to ensure crash consistency, illustrated in Section III-D. The key-value pairs seamlessly integrate into the buffer parts of the last-level internal nodes. When a buffer part of a last-level internal node reaches its capacity threshold, a Leaf Count-Based Flush mechanism will be activated, efficiently transferring the buffered key-value pairs to their respective leaf nodes in PM. This transfer process is elaborated upon in Section III-B, where crash-consistent merge and node split operations occur. Moreover, in cases where a last-level internal node becomes full, LodgeTree executes a Hotness-Aware Multiple Split operation, dividing the node into three parts without introducing significant latency. This operation is detailed in Section III-C. Additionally, LodgeTree implements a coarse-grained lock management system featuring a read/write lock for each last-level internal node during key-value pair insertion or search operations to ensure effective concurrency control. By acquiring the read/write lock in the write state, a thread effectively restricts other threads from accessing the last-level internal node and its associated leaf nodes. This strategic approach eliminates the necessity of locking leaf nodes during key-value pair flushing, thereby reducing lock overhead and enhancing system performance.

Node structures: In LodgeTree, the organization of last-level internal and leaf nodes is demonstrated in Figure 4. Each last-level internal node has a header and a predetermined number of key-value pairs. The header comprises some fields such as *buf_start*, *split_key* and *buf_ver*. *buf_start* indicates the starting position of the buffer part. The *split_key* variable stores the split key of the node when it was last split. The *buf_ver* array records the version numbers of key-value pairs in the buffer part in support of the partitioned version log technique. The split_key, inserted into the parent node of a newly split node, is used to index the newly created node from the split. Moreover, the *leaf_count* array facilitates the leaf-count-based flush technique. Within a last-level internal node, the key-value pairs are segregated into two parts: the **Index Part** and the **Buffer Part**. The former contains sorted keys and pointers that index the corresponding leaf nodes. In contrast, the latter is a temporary repository for unsorted key-value pairs, including a designated slot for pairs generated during leaf node splits. This organization ensures efficient management and retrieval of key-value pairs within LodgeTree's last-level internal nodes, optimizing performance and facilitating operations.

A leaf node in LodgeTree comprises a header containing essential metadata and multiple key-value slots. To minimize persistence overhead, LodgeTree adopts a strategy of leaving key-value pairs within the leaf node unsorted. Instead, it utilizes a bitmap in the header to ensure the crash consistency of the key-value slots, enhancing data integrity and reliability. Additionally, the header encompasses two sibling pointers, *next0* and *next1*, along with a *use_n* bit to determine the efficient utilization of sibling pointers during log-less split operations. This design choice is driven by the high cost associated with logging operations, as highlighted in prior research [13], [21]. By incorporating these features into the leaf node structure, LodgeTree optimizes performance, durability, and operational efficiency, offering a robust foundation for key-value storage and retrieval.

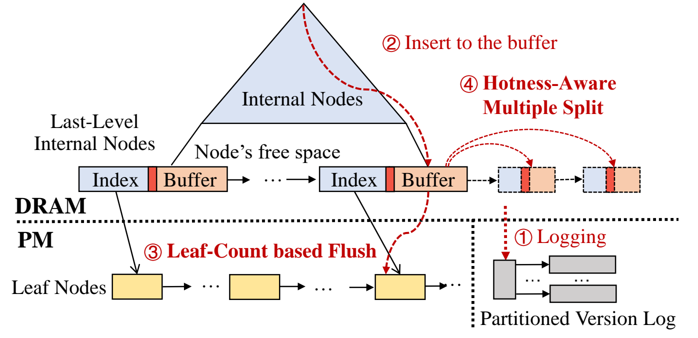


Fig. 3. Architecture overview of LodgeTree.

B. Leaf Count-Based Flush

The Leaf Count-Based Flush technique is designed to mitigate long-tail latency caused by concurrent leaf-node splits during the last-level internal node's flush process. It accomplishes this by limiting the number of PM leaf nodes accessed in parallel during the flush and orchestrating the flush execution to avoid triggering any leaf node splits. To implement this methodology effectively, every last-level internal node is associated with an array named *leaf_count*, comprising N 1-byte integers, as illustrated in Figure 5, where N signifies the total number of slots within the internal node. The *leaf_count* array is segmented into two distinct sections, aligning with the index and buffer parts of the internal node, with *buf_start* marking the boundary. Each integer records the count of key-value pairs housed in the respective leaf node within the index section. In contrast, in the buffer section, each integer precisely designates the specific leaf node to which the buffered key-value pair is assigned. This systematic organization and management of the *leaf_count* array are crucial for optimizing the flush process, improving performance, and ensuring the smooth operation of LodgeTree's storage mechanisms.

When a key-value pair is added to a last-level internal node, the Index Part undergoes key comparisons to determine the pair's destination node, identified as *leaf_pos*. Subsequently, the key-value pair is integrated into the Buffer Part, with the insertion position *buffer_pos* being recorded. Finally, the value at position *buffer_pos* within the *leaf_count* array is adjusted accordingly, setting *leaf_count*[*buffer_pos*] to *leaf_pos*. This sequential process ensures the accurate mapping and organization of key-value pairs within the data structure.

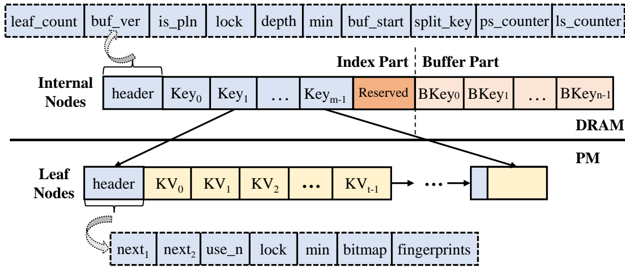


Fig. 4. Structures of last-level internal nodes and leaf nodes.

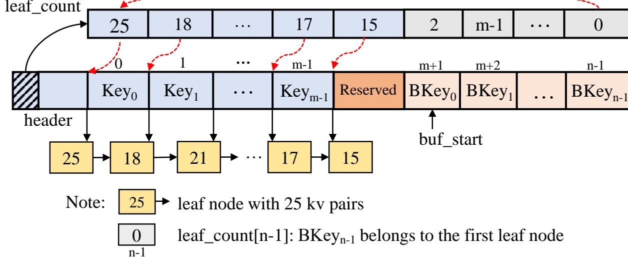
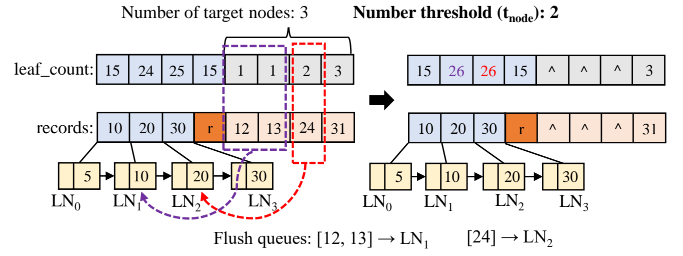


Fig. 5. The structure of leaf_count array of a last-level internal node.



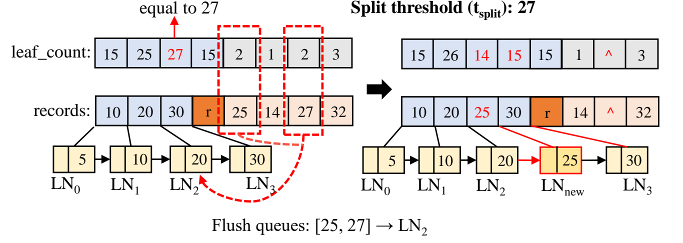


Fig. 6. Two cases of leaf count-based flushes.

The Leaf Count-Based Flush is performed when the Buffer Part is full. Two thresholds are used to control the flush process: t_{node}, which indicates the maximum number of leaf nodes that can be accessed for each flush, and t_{split}, which indicates the maximum number of key-value pairs a leaf node can store. The Buffer Part is traversed from left to right to group the buffered key-value pairs according to their targeted leaf nodes. With the help of the *leaf_count* array, the target node of each buffered key-value pair is recorded. During this process, there are two cases to stop grouping and start flushing.

First, the grouping process stops, and the flush process starts when the number of targeted leaf nodes reaches the threshold t_{node}, which means that this flush will access too many PM leaf nodes. As illustrated in Figure 6(1), t_{node} is set to 2, which means that one flush can only access two PM leaf nodes. However, in this case, buffered key-value pairs have a combined total of three targeted leaf nodes, which exceeds the t_{node} threshold. Consequently, only pairs with keys 12 and 13 can be flushed to leaf node LN_1 and pair with key 24 to leaf node LN_2, completing the flush operation.

Second, the grouping process stops, and the flush process starts if a leaf node is about to split during the Buffer Part traversal. As illustrated in Figure 6(2), where the node split threshold t_{split} is set to 27. The *leaf_count* value of the first buffered pair with key 25 is 3, which means that this buffered key-value pair's targeted leaf node is LN_2. But for LN_2, its *leaf_count* value is 27, reaching the t_{split} threshold, meaning that this leaf node is about to split. As a result, only the key-value pairs belonging to the leaf node LN_2 are flushed. That is, only pairs with keys 25 and 27 are flushed. Following the flush, the pair with key 25 and the pointer to the newly generated leaf node are inserted, and the values in *leaf_count* are updated.

When flushing key-value pairs to PM leaf nodes, crash-consistent merge and split operations are performed, leveraging the PM characteristic of 8-byte atomic write to ensure crash consistency. For leaf node merge, as a 512-byte leaf node contains fewer than 64 key-value slots, the bitmap is less than 8 bytes, allowing the entire bitmap update operation to be completed with a single PM atomic write. The merge process involves searching for key-value slots in the leaf node, writing the pairs to empty slots, persisting the pairs with a single CLFLUSH, and updating the bitmap by modifying the corresponding bits from ”0” to ”1” and then persisting the bitmap with CLFLUSH to complete the merge operation.

During leaf node splitting, a log-less mechanism is implemented, inspired by the designs in LB+Tree [21] and DPTree [32]. This mechanism utilizes two sibling pointers, *next0* and *next1*, along with the *use_n* bit to indicate the currently used sibling pointer. During the process of splitting, if *use_n* points to *next0*, *next1* will point to the new node. Once the new node is prepared, *use_n* is switched from *next0* to *next1*, signifying the completion of the node split. Since *use_n* is smaller than 8 bytes, an atomic write makes the switch operation crash consistent.

C. Hotness-Aware Multiple Split

The Hotness-Aware Multiple Split approach is devised to enhance buffer space utilization while minimizing extended tail latency. Key-value pairs are stored in free slots of last-level internal nodes, directly impacting buffer efficiency. However, increasing the number of split nodes may lead to more Structured Modification Operations (SMOs), triggering write bursts and causing long latency. To mitigate this issue, the approach permits last-level internal nodes to split into more new nodes during a write burst and fewer new nodes when the burst diminishes, without necessitating upper-level node splits. This balances buffer space and split latency effectively.

In LodgeTree, an ingenious approach is adopted to assess the system's activity level, leveraging two counters within each last-level internal node in conjunction with a global counter,

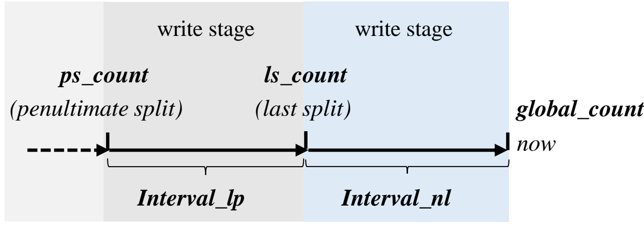


Fig. 7. Timeline of the three counters and two intervals.

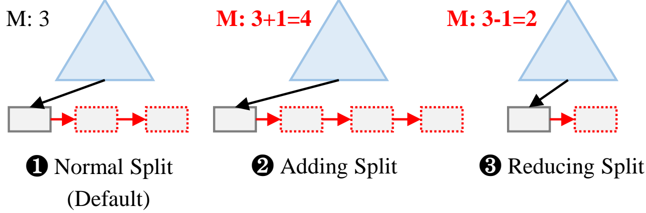


Fig. 8. Three types of last-level internal node split in hotness-aware multiple split.

as depicted in Figure 7. The initial counter, *ps_counter*, logs the timing of the penultimate split, while the subsequent counter, *ls_counter*, precisely captures the timing of the last split event. Complementing these internal node counters is the *global_counter*, which provides insights into the system's cumulative runtime. When a last-level internal node approaches the threshold for splitting, LodgeTree orchestrates a comparison between *interval_lp* (ls_count - ps_count) and *interval_nl* (global_counter - ls_count) to evaluate the pace of buffer depletion. Should *interval_lp* exceed *t* times *interval_nl*, indicating a notable surge in data volume, a write burst is identified. Conversely, if *interval_lp* is more than *t* times smaller than *interval_nl*, it signifies a cooling-off period for the node. Any other scenarios are categorized as stable periods. This dynamic comparative analysis empowers LodgeTree to fine-tune the number of new nodes a last-level internal node can split into, effectively tailoring the system's behavior to the prevailing workload conditions.

Now that LodgeTree can assess the hotness of a last-level internal node, it can execute various node split strategies. Figure 8 shows three types of last-level internal node splits in hotness-aware multiple split. Suppose a last-level internal node split *M* nodes last split. When a last-level internal node, which previously split into *M* nodes, experiences a write burst, LodgeTree will employ an **adding split**, causing the node to split into *M+1* new nodes. This action aims to introduce new nodes to distribute the hotspot data and increase the buffer size of each new node. Conversely, during a cooldown period, LodgeTree will implement a **reducing split**, causing the node to split into *M-1* new nodes to conserve DRAM space. In stable periods, LodgeTree will execute a **normal split**, leading to the node splitting into *M* nodes.

When a node encounters a write burst (*interval_lp* $>$ *t* $\times$ *interval_nl*), this node's split should avoid SMOs of upper-level nodes. This is because SMOs of upper-level nodes may incur extra long latency. To achieve this, a last-level internal node should also consider the space of upper-level nodes.

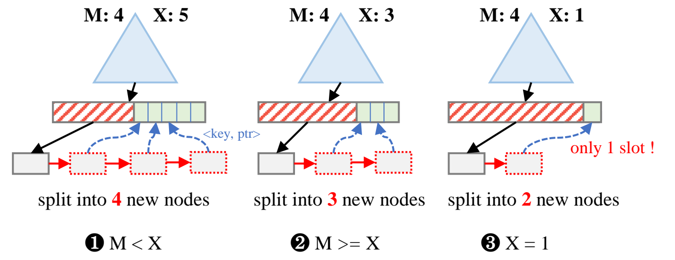


Fig. 9. Three situations of hotness-aware multiple split when considering space of upper-level nodes.

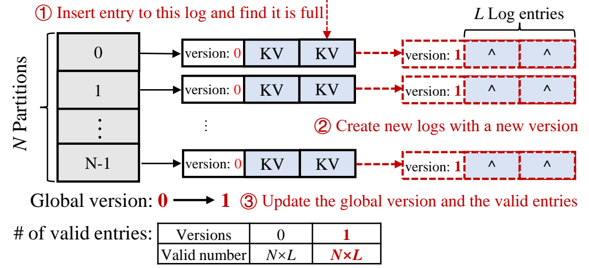


Fig. 10. The structure of the partitioned version log.

Suppose the upper-level node has X free slots, and the last-level internal node is about to split into M new nodes. To the end, the number of nodes split from this node N needs to meet:

$$N = \begin{cases} M, & M < X \\ X, & M \geq X \land X \neq 1 \\ 2, & X = 1 \end{cases} \quad (1)$$

Note that when a node is split into M new nodes, only *M-1* new $\langle key, ptr \rangle$ pairs will be inserted into the upper-level node. As shown in Figure 9, the meaning of Formula 1 is: when $X = 1$, the upper-level node must split. At this time, to reduce the waiting time for the remaining writes, this last-level internal node will only split into two new nodes in the end. When $X < M$, to avoid the upper-level node splitting, this last-level internal node will be split into X new nodes to make the upper-level node just not split.

D. Partitioned Version Log

The Partitioned Version Log (PVL) addresses the complexity of Write-Ahead Log (WAL) management in the distributed buffer of last-level internal nodes. Figure 10 illustrates its structure. The core idea of PVL is to allocate a specific number of log entries, track the count of invalid key-value pairs after each buffer flush, and free or reuse space when the count reaches zero. In the setup with N threads, N partitions are initially allocated, each with a linked list. Each node in the list represents a log with L log entries sharing the same version number. To facilitate log space recycling, the count of valid log entries for each version is recorded and decreases during buffer flushes. When a thread writes data to PVL, the entry is appended directly to its partition's current log. If the current log is full, PVL creates new logs with an incremented version number (e.g., by 1). Subsequently, PVL updates the global version and the count of valid entries, as shown in Figure 10.

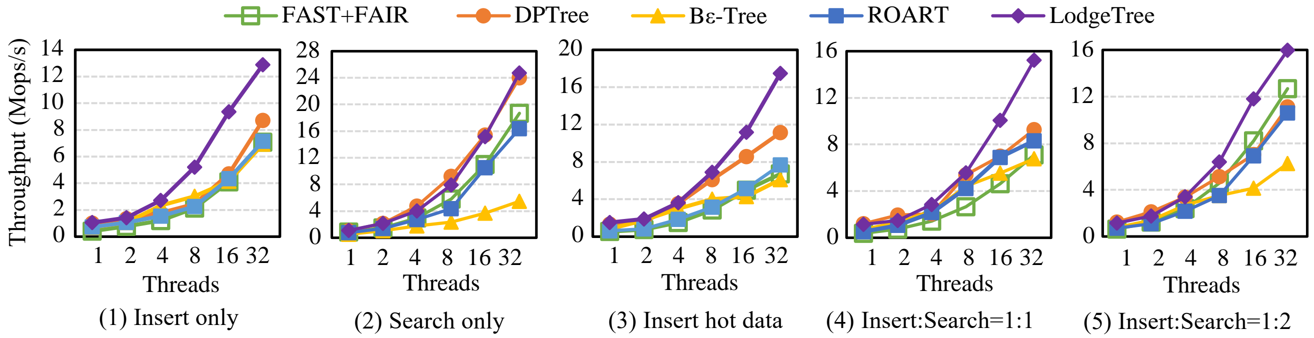


Fig. 11. The results of different schemes under different workloads.

After a key-value pair is inserted into PVL, its version number is recorded with *buf_ver* in the target last-level internal node's header. Meanwhile, the key-value pair is inserted into the corresponding buffer part. During the buffer flush, since each buffered key-value pair can find its corresponding version number in the *buf_ver*, the number of valid entries of the version number in *buf_ver* can be decreased. Suppose the number of valid entries of one version is reduced to 0. All key-value pairs with this version number are flushed to the PM leaf nodes, allowing this log space to be freed or reused.

IV. Performance Evaluation

A. Environment & Implementation

Experimental Environment: Experimental evaluations were performed on a Linux server with kernel version 5.4.0 and two 24 cores Intel Skylake-SP Xeon 5318 2.10GHz CPU with 22MB L3 cache. The server has 96GB of DDR4 DRAM and eight Intel Optane DC PMMs with 1024 GB total capacity. The Intel Optane DC PMMs are configured in the App Direct mode and mounted with the XFS-DAX file system. We also conducted the experimental evaluation based on the server's NUMA architecture, as demonstrated in Section IV-D.

Prototype Implementation: We implemented the prototype of LodgeTree in C++11 based on the FAST+FAIR [13]. By default, the node size of LodgeTree is 512 bytes, which also includes the 27 key-value slots in a PM leaf node and 29 key-value slots in each internal node. We set the threshold t_{node} to be 2, t_{split} to be 26 for leaf-count based flush. All these parameters are configurable, and we conduct the corresponding sensitivity evaluations. Additionally, based on the fact many tree-based indexes are optimized for fixed-sized KV [13], [16], [22], we also evaluate the performance of LodgeTree with the same size (8 bytes) of the key, compared with other schemes: FAST+FAIR [13], DPTree [32], B^{ϵ}-Tree [5], and ROART [22], illustrated as follows.

- **FAST+FAIR**: The FAST+FAIR is a state-of-the-art persistent B+-Tree to optimize the insertion and search algorithms for PM.
- **DPTree**: DPTree is designed to batch multiple writes in a concentrated memory persistently and later merge them into a PM component to amortize persistence overhead.
- **ROART**: ROART is a Range-query Optimized Adaptive Radix Tree to reduce PM persistence overhead.
- B^{ϵ}**-Tree**: Designed for write-optimizing for the on-disk storage, B^{ϵ}-Tree allocates the internal node space for a buffer to store messages, which absorbs updates that will eventually be flushed to items in leaves under this node.

Since the B^{ϵ}-Tree is designed for disk-based storage, we rewrite it for DRAM/PM hybrid storage based on its open-source implementation [3]. Moreover, we use open-source implementations of the other schemes (e.g., FAST+FAIR, DPTree, and ROART) for comparison.

B. Micro-benchmark

We evaluate the basic insert and search efficiency, exploiting the micro-benchmark [26] in which we can also configure its parameters. In the micro-benchmark [26], different values of the *zipfianconstant* parameter represent different skewness of the data. We can change the values of these parameters to adjust the distribution of Zipf and generate other data with various read/write ratios or skewness. The experiment generated 50M of data in each of the five workload configurations: (1) insert only, 100% inserts. (2) search only, 100% search. (3) insert with strong locality, 100% inserts (i.e., 80% of the insertions are for 20% data). (4) insert : search = 1:1, 50% insert and 50% search, (5) insert : search = 1:2, 34% insert and 66% search approximately. Based on these configurations' data, we assessed the performance variation of different schemes as the number of threads increased. Moreover, we perform an insert operation employing 50M key-value pairs generated randomly to warm up the system before executing the following experiments.

Insert only: Figure 11(1) shows the *insert only* throughput curves of all schemes. LodgeTree performs best and outperforms other schemes by up to 2.5× with an average of 1.8×. First, compared with FAST+FAIR and ROART, LodgeTree can better utilize the PM access granularity to reduce random access. Second, LodgeTree's fine-grained flush mechanism can avoid DPTree's high flush overhead and better utilize PM's bandwidth advantage in multi-threaded scenarios. Moreover, compared with the B^{ϵ}-Tree, LodgeTree can also avoid the high cascade flush, leading to a much higher throughput.

Search only: The Figure 11(2) shows the search throughput curves of all schemes. LodgeTree and DPTree outperform the other schemes by up to 4.7× with an average of 1.7×. Compared with ROART and FAST+FAIR, both without the buffer design, LodgeTree can directly hit the key-value pairs

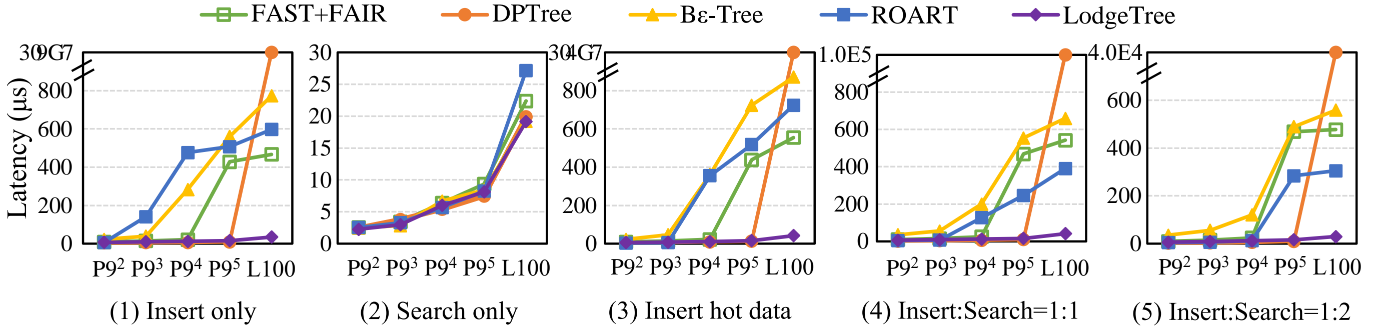


Fig. 12. Tail latency of different schemes with 1 thread. The $P9^2$, $P9^3$, $P9^4$, $P9^5$ and L100 represent P99, P99.9, P99.99, P99.999 and Last 100 respectively.

buffered in last-level internal nodes in DRAM, thus avoiding costly PM random accesses. However, the B^ϵ-Tree scheme uses a multiple-level buffer design, in which the searching process will go through multiple levels to increase the search overhead greatly. Consequently, the search performance of LodgeTree and DPTree is much better than that of B^ϵ-Tree.

Insert with strong locality: Figure 11(3) shows the insert throughput curves of all schemes under workload with strong locality. LodgeTree outperforms the other methods by up to 2.9× with an average of 2.0×. The critical reason includes three aspects. First, since ROART and FAST+FAIR update data in PM, the hot data will inevitably cause repeated PM flushes. By contrast, the buffer design in LodgeTree can absorb these repeated updates in the buffer and avoid repeated PM flushes. Second, the multiple-level buffer design in B^ϵ-Tree first absorbs data in the upper-level nodes. This can cause the buffer space in upper-level nodes to fill quickly, triggering cascade flushes that degrade the performance. Third, as the number of threads increases, the impact of flush-induced write stalls in DPTree would incur performance degradation.

Mixed workloads: Figures 11(4) and (5) show the throughput curves of all schemes driven by mixed workloads which consist of different ratios between insert and search requests (1:1 and 1:2). We can observe that, as the number of threads increases, the throughput of LodgeTree consistently increases and notably outperforms other schemes by up to 3.0× with an average of 1.6×. Meanwhile, the result also implies that LodgeTree can provide better scalability than other schemes, especially for mixed workloads.

C. Tail Latency

The buffer's superiority over a Tree-based index during the writing process is evident in two key aspects: throughput, as illustrated in Section IV-B, and tail latency, a crucial performance metric in system evaluation. Figure 12 presents the tail latency of various schemes, including P99, P99.9, P99.99, P99.999, and the average latency of the highest 100 requests (*L100*). LodgeTree consistently achieves the lowest tail latency across all workloads. For instance, under the *search-only* workload (Figure 12(2)), where flush operations are not intensive, all schemes exhibit tail latencies below 30μs. However, under workloads involving insertions (Figure 12(1),(3-5)), only LodgeTree maintains a tail latency below 50μs. In contrast, other schemes experience significantly higher P99

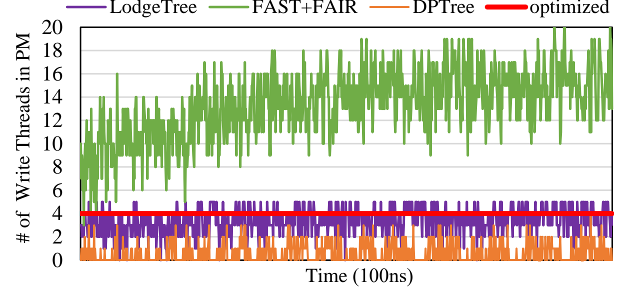


Fig. 13. Variation in the number of threads concurrently writing to PM over time.

latencies. For instance, under the insert-only workload, the *P99.999* latencies for FAST+FAIR, ROART, and LodgeTree are 486.6μs, 507μs, and 16.06μs, respectively.

The disparity in *L100* latency between LodgeTree and other schemes is significant. LodgeTree's distributed buffer design conducts fine-grained flushes, notably reducing flush overhead compared to DPTree. For ROART and FAST+FAIR, long tail latency primarily results from structural modification operations (SMO) like node splits. However, by adopting the hotness-aware multiple split approach, LodgeTree ensures that each split considers available space in upper-level nodes and node data volume, thus avoiding long-latency SMOs in upper-level nodes and reducing write waits at hot nodes. In B^ϵ-Tree, long tail latency stems from cascade flushes and concurrent leaf-node splits. Moreover, LodgeTree's buffer layout and leaf-count-based flush strategy help avoid cascade flushes and reduce the number of leaf node splits during a single flush, resulting in significantly lower tail latency.

D. Other Evaluations

Multi-thread validation: Research findings [28] and our analysis reveal a non-linear relationship between thread count and throughput during write operations on PM devices. This is due to limitations in the number and size of write buffers within PM's integrated memory controller (IMC). Exceeding the optimal thread count (e.g., 4 threads) or reaching buffer capacity triggers blocking, degrading system performance. Figure 13 depicts thread dynamics during data writing with 32 threads, comparing DPTree, FAST+FAIR, and our approach. The red line represents the optimal scenario of four threads for maximum bandwidth, as established by prior research [28]. In FAST+FAIR, where data resides entirely on PM, most thread accesses are PM-based, resulting in the highest PM thread

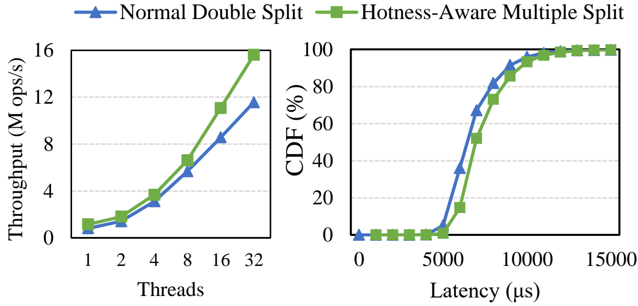


Fig. 14. Effectiveness of hotness-aware multiple split.

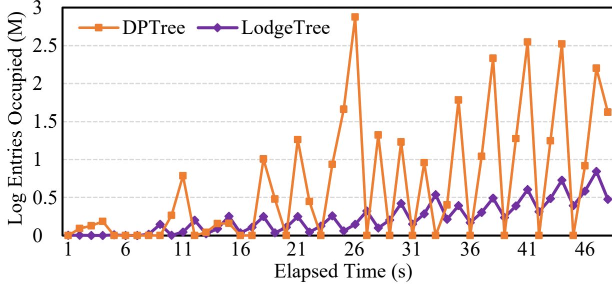


Fig. 15. The number of log entries during execution for DPTree and LodgeTree.

count. However, its throughput is lower due to heavy contention with more than four threads. DPTree's design causes PM idling and reduced throughput, while LodgeTree sustains around four threads writing to PM for extended periods. This minimizes competition, enhances multi-thread throughput, and maximizes PM bandwidth utilization, leading to superior write throughput compared to the others.

Hotness-Aware Multiple Split: We compared insertion throughput and latency between Normal Double Split and Hotness-Aware Multiple Split in workloads with strong data locality. Hotness-Aware Multiple Split ranges from 1.1 $\times$ to 1.4 $\times$ in multi-threaded scenarios, with the advantage becoming more pronounced with additional threads. For instance, it's 12% faster with one thread but 45% faster with 32 threads compared to a Normal Double Split. This improvement is due to simultaneous access by multiple threads to hot nodes, allowing more efficient node splits to create extra buffer space and reduce contention. The tail latency curve of hotness-aware multiple split closely mirrors that of the Normal Double Split, thanks to upper-level node space allocation during splitting, minimizing complex modifications and reducing tail latency.

Partitioned version log: The design goal of the partitioned version log is to simplify log management. As the data are inserted, expired logs must be reclaimed in time without taking up too much space. Figure 15 shows the number of log entries (stored key-value pairs in logs) occupied by valid key-value pairs per second during the data insertion of DPTree and LodgeTree under four threads under workloads with no locality. As data is inserted and buffer space increases, the number of log entries used by LodgeTree with PVL slowly grows, and invalid logs are continuously reclaimed. Compared with DPTree, LodgeTree with PVL uses much fewer log entries and thus consumes less PM space, demonstrating the efficiency of PVL in managing WAL.

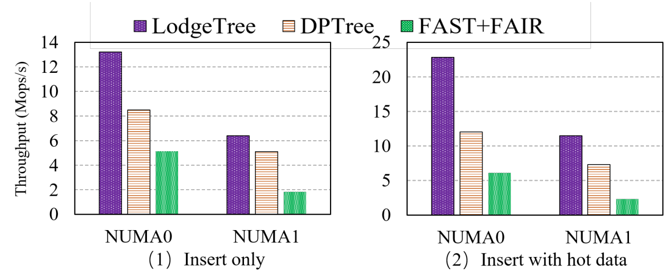


Fig. 16. Evaluation with 32 threads based on NUMA architecture. Note that *NUMA0* presents running in the PMs in CPU0. In contrast, the *NUMA1* illustrates running in the PMs bounded in the far-end node (CPU1).

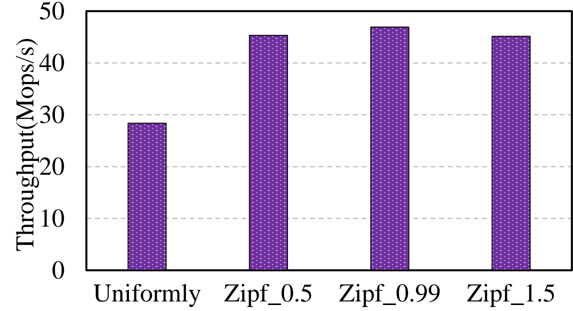


Fig. 17. The system's throughput under different data skew. The uniformly denotes the uniform distribution. The Zipf_0.5 denotes the parameter of Zipf is 0.5, and so on.

NUMA Architecture: We conducted experiments with two distinct workloads under MUNA architecture: *insert only* means all insert operations obey uniform distribution, and *insert with hot data* means 80% of the inserts are for 20% data. Our server features two CPU sockets, both utilizing PM. We designate tests bound to CPU0 as local (NUMA0) and those linked to CPU1 as remote (NUMA1). Figure 16 illustrates the evaluation within the NUMA architecture. The results demonstrate that system throughput is nearly doubled when accessing data locally on Node0 compared to Node1, which aligns with the anticipated latency disparity between remote and local node access in a typical NUMA setup. This emphasizes the significant performance advantage of LodgeTree's buffer mechanism, which is particularly evident in NUMA environments.

Zipf distribution: The performance of a cache system can be affected by the degree of data skewing. Using different Zipf distributions, our study delved into LodgeTree's performance under varying degrees of data skewness. Illustrated in Figure 17, the system's performance across these distributions demonstrates that higher Zipf distribution parameters boost system throughput, reflecting increased data skewness. Under a stable Zipf distribution, the system achieves 1.65$\times$ higher throughput than a uniform distribution. This enhancement results from improved data caching, enabling more requests to be serviced from the cache, thereby elevating system throughput. Despite reaching a threshold where most hot data resides in the cache, the system maintains a consistent throughput level through cache replacement and refreshing processes.

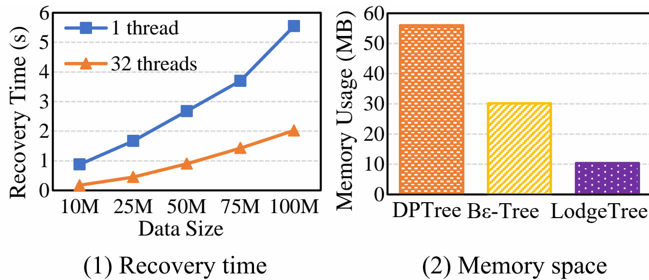


Fig. 18. Overhead results of recovery and memory space.

E. Overhead and Limitation

Overhead of Recovery: Figure 18(1) illustrates the recovery time of LodgeTree under different data sizes with 1 and 32 threads. When utilizing a single thread, LodgeTree's recovery time escalates from 0.87s to 5.55s as the data size grows. Conversely, with 32 threads, the recovery time rises from 0.17s to 2.10s with increasing data size. The results indicate LodgeTree's superior performance in a multi-threaded environment. This enhancement can be attributed to the efficient parallel reinsertion of key-value pairs in the log to last-level internal nodes facilitated by LodgeTree's partitioned version log design. By executing reinsert operations concurrently, LodgeTree achieves a substantial boost in recovery performance when multiple threads are engaged.

Overhead of Memory Consumption: Figure 18(2) shows the actual DRAM memory space usages of DPTree, B^{ϵ}-Tree, and LodgeTree under 50M data single-threaded insertion, excluding the key-value data stored in the PM. Compared with DPTree and B^{ϵ}-Tree, LodgeTree's surrogate buffer design dramatically reduces the memory space overhead by 81.4% and 65.5%. The main reason is that LodgeTree doesn't need extra memory as the particular buffer. This indicates that the surrogate buffer design can exploit both the workload and the B+-tree characteristics to reduce memory space overhead.

Limitation: Based on the above experimental analysis, we discuss the following two limitations. 1) The buffer size in LodgeTree is limited, illustrated in Section II-C, indicating that the number of key-value pairs that can be stored is limited. However, this is also the key to LodgeTree's ability to reduce the high flush (cascade flush) overhead without extra memory space. 2) LodgeTree can not improve the performance of scan search. Figure 19 shows the throughput under different YCSB workloads. We observe that the throughput of LodgeTree lowers FAST+FAIT and DPTree under benchmark (95% range-scan and 5% insert). The reason is that the center of gravity of optimization of LodgeTree significantly reduces the latency overhead from flush operations in the writes path.

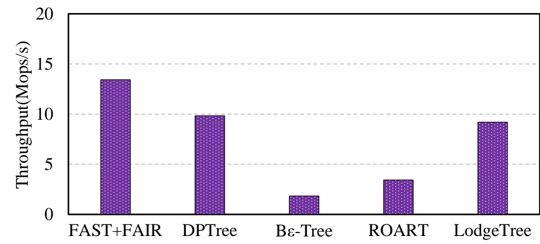


Fig. 19. The *scan* performance of 50M KV pairs for different schemes along with the elapsed time.

V. Related Work

The emergence of NVMs has underscored the significance of tree-based indexing structures in the realm of storage system development. FPTree [23] innovatively employs a fingerprint technique to streamline leaf probes to just one, albeit necessitating logging during node splits to maintain consistency. In contrast, LB+-Tree [21] optimizes node splits without logging by leveraging Intel TSX instructions for concurrency control. However, it is worth noting that the TSX directives are deprecated and disabled by default in Linux environments. On a different front, NBTree [30] achieves lock-free operations by harnessing eADR in Optane PM, ensuring scalability and minimal PM overhead. Beyond these approaches, various studies delve into algorithm optimization and device property exploitation. For instance, FAST+FAIR [13] eliminates logging requirements while upholding the order of data entries. Similarly, ROART [22] tackles PM persistence overhead through strategies like selective metadata persistence, entry compression, and streamlined ordered split operations. Additionally, WORT [18] guarantees consistency by executing a single 8-byte failure-atomic write per update, thereby preventing the necessity for extra logging or copy-on-write mechanisms.

Research on Intel Optane PM has revealed distinctive characteristics, such as lower write latency than read latency at the system layer and prolonged latency due to repeated flushes to the same cacheline. While existing strategies primarily focus on minimizing persistent memory writes in B+-Tree-based systems, LodgeTree takes a different approach. It targets reducing persistent memory read overhead from random insertions and eliminating repeated flush operations. By utilizing the free space of internal nodes and combining multiple random insertions into a batch write to PM leaf nodes, LodgeTree simultaneously reduces PM random read operations and eliminates repeat flushes.

VI. Conclusion

This paper introduces LodgeTree, aiming to maximize the performance characteristics of PM and the structural features of B+-Tree. LodgeTree utilizes the unused space in the last-level internal nodes to create a distributed and dynamic surrogate buffer for B+-Tree-based schemes. We have implemented a prototype of LodgeTree and conducted extensive experiments. The results demonstrate that LodgeTree significantly enhances the system performance of state-of-the-art B+-Tree-based schemes. Moreover, even in NUMA architecture, our scheme outperforms other alternatives.

Acknowledgment

This work was supported by the National Key R&D Program of China No. 2023YFB4502703, the National Natural Science Foundation of China under Grant No. U22A2027 and No. 61972325, and Open Project Program of Wuhan National Laboratory for Optoelectronics No. 2021WNLOKF011.

REFERENCES

[1] CrossBar High-Density Memory. https://www.crossbar-inc.com/products/high-density-memory/, 2019.

[2] Intel® Optane™ DC PM. https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html, 2019.

[3] A simple, reference implementation of a Bϵ-Tree. https://github.com/oscarlab/Be-Tree, 2020.

[4] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. BzTree: A High-Performance Latch-Free Range Index for Non-Volatile Memory. *Proceedings of the VLDB Endowment*, 11(5):553–565, 2018.

[5] Michael A Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C Kuszmaul, Donald E Porter, Jun Yuan, and Yang Zhan. An Introduction to B-trees and Write-Optimization. *login; magazine*, 40(5), 2015.

[6] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*, 2020.

[7] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage. In *Proceedings of 19th USENIX Conference on File and Storage Technologies (FAST'21)*, pages 17–32, 2021.

[8] Shimin Chen and Qin Jin. Persistent B+-Trees in Non-Volatile Main Memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, 2015.

[9] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. uTree: a Persistent B+-Tree with Low Tail Latency. *Proceedings of the VLDB Endowment*, 13(12):2634–2648, 2020.

[10] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, 2020.

[11] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC'20)*, pages 49–63, 2020.

[12] Kecheng Huang, Zhaoyan Shen, Zhiping Jia, Zili Shao, and Feng Chen. Removing Double-Logging with Passive Data Persistence in LSM-tree based Relational Databases. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST'22)*, Santa Clara, CA, 2022.

[13] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*, 2018.

[14] S Ikeda, K Miura, H Yamamoto, K Mizunuma, HD Gan, M Endo, Sl Kanai, J Hayakawa, F Matsukura, and H Ohno. A perpendicular-anisotropy CoFeB–MgO Magnetic Tunnel Junction. *Nature Materials*, 9(9):721–724, Jul. 2010.

[15] Yizheng Jiao, Simon Bertron, Sagar Patel, Luke Zeller, Rory Bennett, Nirjhar Mukherjee, Michael Bender, Michael Condict, Alex Conway, Martin Farach-Colton, Xiongzi Ge, William Jannen, Rob Johnson, Donald Porter, and Jun Yuan. BetrFS: A Compleat File System for Commodity SSDs. In *Proceedings of the European Conference on Computer Systems (EuroSys'22)*, April 2022.

[16] Wook-Hee Kim, R Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*, 2021.

[17] Wook-Hee Kim, Jihye Seo, Jinwoong Kim, and Beomseok Nam. CLFB-tree: Cacheline Friendly Persistent B-Tree for NVRAM. *ACM Transactions on Storage*, 14(1):1–17, 2018.

[18] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*, 2017.

[19] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. Evaluating persistent memory range indexes. *Proceedings of the VLDB Endowment*, 13(4):574–587, 2019.

[20] Yongkun Li, Zhen Liu, Patrick PC Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. Differentiated Key-Value Storage Management for Balanced I/O Performance. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC'21)*, pages 673–687, 2021.

[21] Jihang Liu, Shimin Chen, and Lujun Wang. Lb+Trees: Optimizing Persistent Index Performance on 3DXpoint Memory. *Proceedings of the VLDB Endowment*, 13(7):1078–1090, 2020.

[22] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. ROART: Range-query Optimized Persistent ART. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST'21)*, 2021.

[23] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16)*, 2016.

[24] Chundong Wang, Sudipta Chattopadhyay, and Gunavaran Brihadiswarn. Crash Recoverable ARMv8-oriented B+-Tree for Byte-Addressable Persistent Memory. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'19)*, 2019.

[25] Jing Wang, Youyou Lu, Qing Wang, Minhui Xie, Keji Huang, and Jiwu Shu. Pacman: An Efficient Compaction Approach for Log-Structured Key-Value Store on Persistent Memory. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'22)*, July 2022.

[26] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. Building a Bw-Tree Takes More Than Just Buzz Words. Proceedings of the 2018 International Conference on Management of Data (SIGMOD'18), 2018.

[27] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. Characterizing the Performance of Intel Optane Persistent Memory–A Close Look at its On-DIMM Buffering. In *Proceedings of the European Conference on Computer Systems (EuroSys'22)*, April 2022.

[28] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*, 2020.

[29] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC'20)*, pages 17–31, 2020.

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

[31] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. ChameleonDB: a Key-value Store for Optane Persistent Memory. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys'21)*, pages 194–209, 2021.

[32] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. DPTree: Differential Indexing for Persistent Memory. *Proceedings of the VLDB Endowment*, 13(4):421–434, 2019.