NBTree: a Lock-free PM-friendly Persistent B$^+$-Tree for eADR-enabled PM Systems

Bowen Zhang
Shanghai Jiao Tong University
bowenzhang@sjtu.edu.cn

Shengan Zheng
MoE Key Lab of Artificial Intelligence, AI Institute,
Shanghai Jiao Tong University
venero1209@sjtu.edu.cn

Zhenlin Qi
Shanghai Jiao Tong University
qizhenlin@sjtu.edu.cn

Linpeng Huang
Shanghai Jiao Tong University
lphuang@sjtu.edu.cn

ABSTRACT

Persistent memory (PM) promises near-DRAM performance as well as data persistency. Recently, a new feature called eADR is available on the 2^{nd} generation Intel Optane PM with the 3^{rd} generation Intel Xeon Scalable Processors. eADR ensures that data stored within the CPU caches will be flushed to PM upon the power failure. Thus, in eADR-enabled PM systems, the globally visible data is considered persistent, and explicit data flushes are no longer necessary. The emergence of eADR presents unique opportunities to build lock-free data structures and unleash the full potential of PM.

In this paper, we propose NBTree, a lock-free PM-friendly B$^+$-Tree, to deliver high scalability and low PM overhead. To our knowledge, NBTree is the first persistent index designed for eADR-enabled PM systems. To achieve lock-free, NBTree uses atomic primitives to serialize leaf node operations. Moreover, NBTree proposes four novel techniques to enable lock-free access to the leaf during structural modification operations (SMO), including *three-phase SMO*, *sync-on-write*, *sync-on-read*, and *cooperative SMO*. For inner node operations, we develop a *shift-aware search* algorithm to resolve read-write conflicts. To reduce PM overhead, NBTree decouples the leaf nodes into a metadata layer and a key-value layer. The metadata layer is stored in DRAM, along with the inner nodes, to reduce PM accesses. NBTree also adopts *log-structured insert* and *in-place update/delete* to improve cache utilization. Our evaluation shows that NBTree achieves up to 11× higher throughput and 43× lower 99% tail latency than state-of-the-art persistent B$^+$-Trees under YCSB workloads.

PVLDB Reference Format:
Bowen Zhang, Shengan Zheng, Zhenlin Qi, Linpeng Huang. NBTree: a Lock-free PM-friendly Persistent B$^+$-Tree for eADR-enabled PM Systems. PVLDB, 15(6): 1187-1200, 2022.
doi:10.14778/3514061.3514066

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/SJTU-DDST/NBTree.

Proceedings of the VLDB Endowment, Vol. 15, No. 6 ISSN 2150-8097.
doi:10.14778/3514061.3514066
* Linpeng Huang and Shengan Zheng are corresponding authors.

1 INTRODUCTION

Byte-addressable persistent memory (PM), such as Intel Optane DC persistent memory module (DCPMM) [17] is now commercially available. PM offers DRAM-comparable performance as well as disk-like durability. In general, PM-equipped platforms support the asynchronous DRAM refresh (ADR) feature [21], which ensures that the content of the PM DIMMs, as well as the writes that have reached the memory controller's write pending queues (WPQ), survives power failures. However, writes within CPU caches remain volatile. Thus, explicit cache line flush instructions and memory barriers are required to guarantee the persistence of PM writes.

Recently, a new feature called extended ADR (eADR) is available with the arrival of the 3^{rd} generation Intel Xeon Scalable Processors and the 2^{nd} generation Intel Optane DCPMM [24]. Compared with ADR, eADR further guarantees that data within CPU caches will be flushed back to PM after a crash through the reserved energy. It ensures the persistence of the globally visible data in CPU caches and eliminates the need to issue costly synchronous flushes. The emergence of eADR not only facilitates the design of lock-free data structures but reduces PM write overhead.

Building efficient index structures in PM is promising to offer both high performance and data durability for in-memory databases. Most existing persistent indexes [2, 4–6, 28, 32, 37, 40, 43, 47, 59, 61] are solely designed for ADR-based PM systems. On eADR-enabled platforms, an intuitive transformation approach is to simply remove all cache line flush instructions [45]. However, this naïve approach cannot fully exploit the potential of eADR. Those indexes still suffer from two major drawbacks even with the eADR support.

First, existing persistent indexes suffer from inefficient concurrency control. Locks are widely used in persistent indexes because none of the existing primitives can atomically modify and persist data on ADR-based platforms. Atomic CPU hardware primitives, such as Compare-And-Swap (`CAS`), can atomically modify the data but do not guarantee its persistence because CPU caches are volatile. Therefore, without locking, it's possible that a `store` hasn't been persisted before a dependent read from another thread, leading to *dirty read* anomaly. Fortunately, eADR closes the gap between the visibility and persistency of the data in CPU caches, making sure that threads always read persistent data. Thus, eADR provides us an opportunity to develop efficient lock-free data structures.

Second, existing persistent indexes still impose high overheads on PM accesses. Prior researches strive to lower PM overhead by

reducing the number of flush instructions, because data flushing is the primary bottleneck of ADR-based PM systems [37]. With eADR, explicit data flushes to PM are no longer necessary. However, the performance of persistent indexes is still restricted by excessive PM accesses since PM has higher read latency and lower bandwidth than DRAM [16, 52, 55]. Especially for the write operations, although data flushing to PM is off the critical path, dirty cache lines will eventually be written back to PM due to the limited CPU cache capacity. Therefore, it's necessary to redesign the PM-friendly persistent indexes for eADR-enabled PM systems.

In this paper, we present NBTree, a lock-free PM-friendly B$^+$-Tree to deliver high scalability and low PM overhead. To our knowledge, NBTree is the first PM index based on eADR-enabled PM systems.

To achieve high scalability, NBTree proposes a fully lock-free concurrency control protocol. For leaf node operations, NBTree adopts *log-structured insert* and *in-place update/delete*, combining with `CAS` primitives, to support lock-free accesses. When the inserted leaf is full, NBTree replaces the old leaf with new leaves to maintain the balance of nodes via structural modification operations (SMO). NBTree proposes three novel techniques (*three-phase SMO*, *sync-on-write*, and *sync-on-read*) to deal with the potential anomalies during the lock-free accesses to the leaf in SMO: (1) *Lost update* caused by concurrent updates and deletions to the leaf. NBTree addresses this anomaly by utilizing *three-phase SMO* and *sync-on-write*. When an update or deletion operates on the leaf during SMO, it first in-place modifies the old leaf. Then, the modification is either passively migrated to the new leaf by *three-phase SMO* or actively synchronized to the new leaf using *sync-on-write*. (2) *Dirty or stale read* caused by concurrent search operations. The lock-free search on the SMO leaf might read uncommitted dirty data or stale data. NBTree uses the *sync-on-read* technique to detect and resolve those anomalies. To further reduce tail latency, we propose *cooperative SMO* to make concurrent insertions to the same SMO leaf work cooperatively. For inner node operations, NBTree applies hardware transactional memory (HTM) [26] to achieve atomic writes. Meanwhile, NBTree designs a *shift-aware search* algorithm to ensure the lock-free inner node search reaches the correct leaf.

To reduce PM overhead, NBTree minimizes PM line accesses and improves cache utilization. For leaf nodes in NBTree, the metadata and key-value pairs are decoupled into two layers. The metadata layer is stored in DRAM along with inner nodes. PM only contains the key-value layer so that the number of PM line reads and writes is minimized. The volatile part of NBTree can be rebuilt with the persistent key-value layer after a crash. Moreover, our proposed *log-structured insert* and *in-place update* improve the possibility of *write combining* and *write hits*, optimizing the cache utilization in eADR-enabled PM systems.

In summary, the contributions of this paper include:

- We provide an in-depth analysis of the benefits of the eADR feature. Then, we propose NBTree, the first persistent index based on eADR-enabled PM systems as far as we know.
- We propose lock-free concurrency control for NBTree to achieve high scalability. Our proposed techniques, such as *three-phase SMO*, *sync-on-write*, *sync-on-read*, *cooperative SMO*, and *shift-aware search*, ensure strong consistency for lock-free operations.
- We propose a two-layer leaf node structure for NBTree, which reduces the number of PM line reads and writes in each operation and improves cache utilization.
- We implement NBTree and our evaluation results show that NBTree achieves up to 11× higher throughput and 43× lower 99% tail latency than state-of-the-art counterparts under YCSB workloads.

2 BACKGROUND AND MOTIVATION

In this section, we introduce the background of persistent memory and eADR (Section 2.1), the PM overhead analysis in eADR-enabled PM systems (Section 2.2), and the challenges of designing lock-free persistent data structures (Section 2.3).

2.1 Persistent Memory and eADR

Persistent memory (PM), which is now commercially available, provides many attractive features, such as byte-addressability and data persistency. However, PM still has higher latency and lower bandwidth than DRAM. To reduce the write latency, existing PM-based systems utilize the ADR mechanism [21] to drain the writes sitting on the write pending queues (WPQ) to PM by the reserved energy during a power outage. Therefore, data that reaches the WPQ in ADR-based PM systems is considered persistent, whereas data in the CPU caches remains volatile. As a result, an additional pair of the flush instruction (e.g. `clwb`, `clflush`, `clflushopt`) and memory barrier (e.g. `mfence`, `sfence`) is necessary for programmers to guarantee data persistency [45].

Fortunately, eADR is supported on the 2^{nd} generation Intel Optane DCPMM with the 3^{rd} generation Intel Xeon Scalable Processors. eADR-enabled PM systems reserve more energy that enables them to flush data in CPU caches to PM after a power failure, thereby expanding the persistence domain to include CPU caches [46]. eADR offers the following advantages over ADR.

The first one is reducing PM write overhead. With eADR, the synchronous flush instructions are no longer necessary, which reduces PM write overhead in two aspects. (1) Reducing the latency in the critical path. Previously, the flush instructions and memory barriers result in high latency in the critical path [37]. (2) Saving PM write bandwidth. Delaying writes to PM increases *write hits* and *write combining* in CPU caches, which reduces PM writes.

The second one is facilitating the lock-free design. Data structures can atomically modify and persist data with eADR, which facilitates the lock-free design in PM. Most lock-free data structures [1, 18, 36, 41] rely on atomic CPU hardware primitives, such as `CAS`. However, in ADR-based PM systems, those primitives can atomically modify data but cannot ensure their persistence because CPU caches are volatile. Threads are likely to read unpersisted data in CPU caches, resulting in the *dirty read* anomaly. With eADR, the globally visible data in CPU caches is ensured to be persisted. Thus, it is possible to modify and persist data atomically.

2.2 PM Overhead Analysis

The performance gap between PM and DRAM encourages people to design PM-friendly storage systems to reduce I/O overhead. Previous works [7, 8, 12, 31, 42, 49, 53, 54, 58] designed for ADR-based PM systems mostly focused on reducing the costly flush

Table 1: The PM overhead of tree operations. ($a/b/c$ indicates the PM overhead of an individual insert/delete/update. n indicates the number of key-value pairs in the leaf node.)

	Flush	PM line write	PM line read
NVTree [56]	2/2/2	2/2/2	O(n)
WB$^+$Tree [3]	4/3/3	3/2/2	O(log(n))
FPTree [43]	3/1/3	3/1/3	3
RNTree [39]	2/1/2	3/1/3	O(log(n))
BzTree [2]	15/7/10	11/6/7	O(log(n))
FAST&FAIR [20]	O(n)/O(n)/1	O(n)/O(n)/1	O(n)
uTree [4]	2/1/1	2/2/1	2
NBTree	**1/1/1**	**1/1/1**	**1**

instructions. With eADR, flushing is no longer required. However, although the persistence latency of the writes is hidden by CPU caches, dirty cache lines will eventually be evicted to PM according to the cache replacement policy. Excessive PM writes still result in high latency due to the poor PM write bandwidth. Besides, PM also has higher read latency than DRAM. Thus, the unique features of eADR require a rethinking of how to reduce PM overhead.

We conclude the following three design goals to reduce PM overhead. First, reducing the number of *PM line writes* per operation. *PM line writes* indicate the 64-byte aligned PM lines modified in CPU caches. Reducing *PM line writes* per operation can produce less dirty cache lines, saving PM write bandwidth. Second, increasing the possibility of *write combining* and *write hits* in CPU caches. In this way, multiple write operations can write to the same cache line, reducing PM writes. ADR-based PM systems do not benefit much from it because write operations often need to be synchronously flushed. Third, reducing the number of *PM line reads* per operation. The relatively higher read latency of PM is overlooked in the previous works [3, 39, 56]. However, it is non-negligible, especially in read-intensive data structures, such as B$^+$-Tree.

Table 1 lists PM costs of state-of-the-art persistent B$^+$-Trees and NBTree. We notice that the strategies of reducing the number of flushes sometimes result in fewer *PM line writes*. However, they are not equivalent. RNTree [39], for example, applies selective metadata persistence to reduce the number of flushes but cannot avoid the *PM line writes*. We also find that trees that keep the order of leaf nodes or slot arrays produce non-constant *PM line reads* per operation, which incurs non-negligible overhead.

2.3 The Design Challenges of Lock-free Persistent Data Structures

It's non-trivial to design lock-free data structures (LFD) in PM because they not only need to handle subtle race conditions like volatile indexes, but need to make sure that writes are persisted before any dependent read [14, 51, 62]. There are the following two hardware restrictions to keep us away from designing efficient LFDs in PM. (1) The granularity of atomicity in memory `load` and `store` is one word, that is 8 bytes. Atomic CPU hardware primitives used in many LFDs, such as compare-and-swap (`CAS`), can only atomically modify a single word. However, a single operation in non-trivial data structures needs to read and write multiple words. Therefore, LFDs are likely to expose intermediate states to concurrent threads. Moreover, when a thread is performing an operation, data structures might be changed by other threads. Those problems may result in anomalies such as *lost update*, *stale read*, and *inconsistent read*. (2) ADR-based PM systems do not support atomic primitives to modify and persist the data. Updates are first sent to the CPU caches and then persisted using flush instructions and memory barriers. As the globally visible data in CPU caches are volatile, other threads can easily read unpersisted data, resulting in *dirty read* anomaly.

In the following, we use the persistent B$^+$-Tree as an example to specify how anomalies mentioned above happen.

Lost Update/Stale Read. Updates may be lost permanently, and reads might access stale data due to non-atomic state changes. For example, structural modification operations, such as split, are the most complex state change in B$^+$-Trees. During the split, B$^+$-Tree transfers the content of the old node to newly allocated nodes and then replaces the old node with new nodes. As the split cannot be completed atomically, a concurrent update may occur in the old node but be missed in the new nodes. In this situation, new nodes are facing the risk of the *stale read* anomaly since they are stale. Even worse, if the update is not synchronized to new nodes in a proper way, it will be lost permanently, incurring the *lost update* anomaly. Thus, existing B$^+$-Trees often lock the leaf during the modification. BzTree [2] uses the `PMwCAS` [51] to guarantee the atomicity of writes. However, BzTree performs even worse than lock-based B$^+$-Trees due to the high software overhead of `PMwCAS` [35].

Inconsistent Read. Threads might read the inconsistent state of data structures due to non-atomic state change. For example, the shift operation to keep nodes in B$^+$-Tree sorted cannot be completed atomically. During an insertion or deletion, B$^+$-Tree needs to shift array elements by calling a sequence of `load` and `store` instructions. During shifting, the same entry may appear twice in different slots, which is an inconsistent state that can result in the *inconsistent read*. FAST&FAIR [20] proposes a lock-free search algorithm, which tolerates such inconsistency. During searching, the key is ignored if its left and right child pointers have the same address. However, *inconsistent read* may occur when the state of the node changes between two load operations [57].

Dirty Read. Reads might access the uncommitted dirty data due to non-durable writes. Because of the lack of atomic instructions with the functionality of persistence, there is a temporal gap between when an update becomes globally visible and when it becomes durable. During the update, we firstly store the data in CPU caches, then persist it using a flush instruction and a memory barrier. Other concurrent threads may view the new update before it persists. If a power outage occurs between these two steps, the read operation will get the unpersisted dirty data. Previous works propose several approaches to deal with this issue. ROART [40] and P-ART [32] use the `non-temporal store` to prevent the unpersisted data from being globally visible, but this method does not benefit from CPU caching. `Link-and-persist` [10] and `PMwCAS` [51] use the help mechanism, which allows read threads to flush unpersisted data proactively. However, it adds additional software overhead and design complexity. With eADR, the *dirty read* anomaly is less likely to happen as the globally visible data is always persistent.

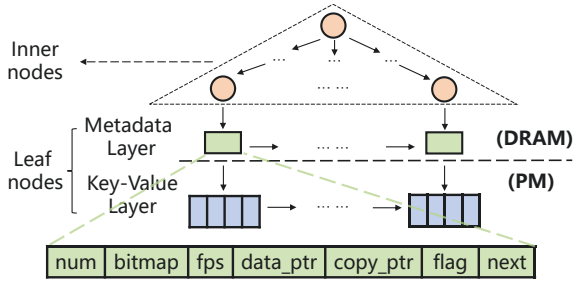


Figure 1: The overall architecture of NBTree.

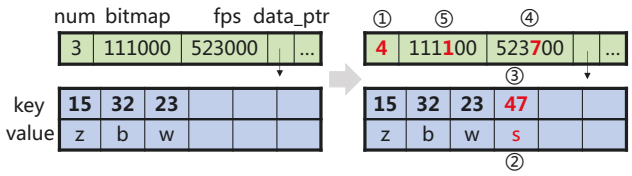


Figure 2: Procedure of an insertion on an NBTree's leaf. (The fingerprints (fps) of the keys are set to $key\%10$ for brevity.)

3 PM-FRIENDLY B$^+$-TREE

NBTree achieves low latency and high scalability by lowering PM overhead in the following two aspects: (1) Reduce the number of *PM line reads/writes* per operation. (2) Leverage the eADR benefits to increase the *write combining* and *write hits* in CPU caches.

In this section, we describe the PM-friendly design of NBTree. We first present the overall architecture (Section 3.1), and then describe the base operations of NBTree (Section 3.2).

3.1 NBTree Structure

The overall architecture of NBTree is shown in Figure 1. In NBTree, the metadata and key-value pairs of the leaf nodes are separated into two layers. The metadata layer, as well as the inner nodes of NBTree, is maintained in DRAM. They can be rebuilt from the persistent key-value layer of leaf nodes in PM during recovery. The two-layer leaf node design enables NBTree to absorb metadata operations in DRAM, reducing PM line accesses drastically.

Specifically, for leaf nodes, both the metadata layer and the key-value layer are linked into a singly linked-list. In the key-value layer, each key-value block is an unsorted array of key-value entries. Each key-value entry stores a 64-bit key and a 64-bit payload. The highest 2 bits (`copy_bit` and `sync_bit`) of the payload are reserved for concurrency control. For variable-sized key-value entries, NBTree stores pointers that indicate the actual keys or values. Each leaf's metadata consists of the following fields: (1) `fps` to store the one-byte fingerprint (hash value) for each key in the leaf, which speeds up key-search on the unsorted array. (2) `num` to store the number of entries occupied by both the committed and in-flight insertions, which handles concurrent insertions. (3) `bitmap` to track the position of the committed insertions in the leaf. (4) `data_ptr` to indicate the address of its key-value block. (5) `copy_ptr` to store the address of newly allocated leaves when the leaf performs SMO. (6) `flag` to track the status of SMO. (7) `next` to indicate the address of the sibling leaf. For inner nodes, NBTree adopts the structure of FAST&FAIR [20], which maintains the sorted array.

3.2 Base operations

NBTree reduces the overhead of base operations (insert, update, delete, and search) by minimizing PM line accesses and maximizing the cache utilization. For each base operation, NBTree first locates the corresponding leaf by searching the inner nodes in DRAM. Then, it uses *log-structured insert*, *in-place update/delete*, and *efficient search* to reduce the average number of *PM line read/writes* on the persistent leaf node to 1 and increase the *write hits* and *write combining*. During recovery, NBTree retrieves all the key-value entries with non-zero keys from the persistent leaf nodes.

Log-structured Insert. Insertions perform in a log-structured manner in NBTree. Figure 2 illustrates the steps of inserting a new key-value pair in NBTree's leaf. First, NBTree increases the `number` (①) to occupy the next free slot. Then, NBTree writes the value (②) and the key (③) to the occupied slot. After writing the key, the new insertion can survive a power failure. Finally, NBTree updates the `fps` (④) and `bitmap` (⑤) to make insertion visible. We observe that the only PM overhead in an insertion is storing a key-value pair. Moreover, with eADR, the *log-structured insert* manner also allows the consecutive insertions on the same leaf to combine in CPU caches, reducing PM writes.

In-place Update/Delete. Conventional log-structured B$^+$-Trees, such as NVTree [56] and RNTree [39], update or delete key-value entries by appending new entries. NBTree, on the other hand, performs *in-place update/delete*. To update a key-value entry, NBTree modifies its value in-place. To delete an entry, NBTree invalidates it by resetting its key to 0. Update and delete can survive system crashes by modifying and persisting the 8-byte key or value with eADR support. In-place update manner is not favored in ADR-based PM systems, as repeatable flushes to the same cache line cause extra latency especially running on skewed workload [5]. With eADR, *in-place* update manner fully utilizes CPU caches and minimizes *PM line writes*.

Efficient Search. The search range is confined to the valid entries indicated by the `bitmap`. This ensures that the entry found by the search operation is persistent and committed. NBTree further narrows the average number of candidate entries to one by checking the `fingerprints`. Finally, NBTree scans the candidate entries to filter the unmatched keys and the deleted keys. In most cases, the search operation produces only one *PM line read*, since the candidate entry is often unique.

Crash Consistency. NBTree can restore its metadata layer and inner nodes using the key-value layer after a crash. During recovery, NBTree scans the list of key-value blocks and labels the slots with non-zero keys as the valid key-value entries. Then, NBTree rebuilds the metadata layer and the inner nodes based on those valid entries. We find that rebuilding NBTree from persistent leaf nodes with 16 million key-value entries takes only 0.32s with a single thread, which is 32× quicker than recovering from the base data.

NBTree maintains consistency even if a crash occurs in the middle of a write operation (insertion/update/deletion). As shown in

Figure 2, during an insertion, writing the key (③) happens after writing the value (②). If the crash happens after writing the key, the intact key-value pair will survive. Otherwise, the in-flight insertion will not leave NBTree in an inconsistent state because the key of the occupied slot is still 0, which is discarded after the recovery. For updates and deletions, they can be completed atomically.

4 LOCK-FREE DESIGN

In this section, we introduce the lock-free concurrency control of NBTree, which is based on a precondition guaranteed by the eADR-enabled platform: **globally visible data is persistent**. We propose different concurrency-control protocols for operations on normal leaf nodes (Section 4.1), leaf nodes during SMO (Section 4.2), and inner nodes (Section 4.3).

4.1 Leaf Node Operations

We divide the base operations in NBTree into two categories. The first category is `insert`, which appends new data to the free slot. The second category is UDS operations, including update, deletion, and search. The UDS operations always work on the committed insertions. In the following, we discuss how NBTree resolves the `insert-insert`, UDS-UDS, and `insert-UDS` conflicts.

Insert-Insert Conflicts. We use atomic primitives to serialize concurrent insertions. As shown in Figure 2, to begin an insertion, NBTree uses the `fetch_and_add` to atomically increase the `num`, occupying the next free slot. This ensures that concurrent insertions are placed in separate slots. At the end of an insertion, NBTree uses CAS to atomically update the `bitmap`, which commits the insertion. In this way, NBTree achieves lock-free insert on the leaf nodes.

Insert-UDS Conflicts. Those conflicts are naturally solved in NBTree. Firstly, an insertion always writes the data into the unused space, which does not affect the UDS operations. Secondly, NBTree commits an insertion by atomically updating the `bitmap`, which makes the new insertion visible to UDS operations. Therefore, UDS operations always operate on the completed insertions.

UDS-UDS Conflicts. UDS-UDS conflicts in NBTree are resolved in eADR-enabled PM systems without additional overhead. As mentioned above, updates and deletions are completed atomically without exposing the intermediate state. With eADR, those modifications are atomically persisted. Thus, the order of commit and visibility for concurrent updates and deletions are always maintained, and the search operation always reads the latest committed data. Moreover, UDS operations are never aborted by other threads, which dramatically improves NBTree's scalability under the workloads with high contentions.

4.2 Structural Modification Operations

Structural modification operations (SMOs) are initiated when a key-value entry is inserted into a full leaf. The conventional procedure of SMO is to copy the entries from the full old leaf to the newly allocated leaves, and then replace the old leaf with the new leaves. However, since the copy phase cannot be completed atomically, lock-free concurrent modifications to the old leaf may not be synchronized to the new leaves, resulting in the *lost update* anomaly.

Table 2: The approaches employed during different phases of SMO to facilitate lock-free leaf node operations.

SMO Phase	Copy	Sync	Link
Update/ Delete	*three-phase SMO* (sync phase)	*sync-on-write*	
Search	unnecessary	*sync-on-read*	unnecessary
Insert	*cooperative SMO*		

Moreover, the lock-free search might read *dirty* or *stale* data due to the inconsistency between the old leaf and new leaves.

Table 2 shows the approaches used by NBTree to resolve the potential anomalies and facilitate lock-free accesses. In NBTree, SMO is divided into three phases (copy phase, sync phase, and link phase). During each phase, different approaches are used to handle concurrent operations on the SMO leaf. For UD (update/delete) operations, NBTree resolves the *lost update* anomaly with the sync phase of the SMO and the *sync-on-write* technique. For search operations, NBTree uses *sync-on-read* to prevent the *dirty read* and *stale read* anomaly. We also propose *cooperative SMO*, which enables concurrent insertions to complete SMO cooperatively.

Three-phase SMO. Different from the SMO of traditional B^+-Trees that only includes the copy phase and link phase, NBTree adds a sync phase to avoid the *lost update* anomaly caused by UD operations during the copy phase. In the following, we will describe the procedure of each phase.

In the copy phase, SMO copies the valid entries with non-zero keys in the full leaf to new leaves. As shown in Figure 3, NBTree allocates two new leaves if the number of valid entries exceeds a certain threshold (half of the leaf capacity by default). Otherwise, only one new leaf is allocated. Then, NBTree distributes key-value pairs to the new leaves and constructs their metadata layer. Finally, NBTree sets the `copy_ptr` in the old leaf to indicate the address of the first new leaf.

In the sync phase, NBTree synchronizes the lost UD operations to new leaves. During the copy phase, concurrent UD operations still write to the old leaf. As the copy phase cannot be completed within an atomic instruction, those UD operations, such as `Update(2,s)` in Figure 3, might not have been migrated to new leaves yet. Therefore, in the sync phase, NBTree employs CAS to synchronize the missed UD operations to new leaves.

The link phase replaces the old leaf in NBTree with new leaves. NBTree firstly links new leaves into the singly linked-list of the key-value layer and the metadata layer by changing the `next` pointer of the previous leaf. Then, NBTree installs new leaves to the parent node (described in Section 4.3).

Sync-on-write. In the post-copy phases of SMO, UD operations resolve the *lost update* anomaly by adopting a *sync-on-write* approach, which actively synchronizes the modification from the old leaf to the new leaf. Specifically, for an update, after modifying a key-value in the old leaf, it re-searches the target key in the new leaf. If the corresponding value in the new leaf is not up-to-date, NBTree synchronizes the latest update to the new leaf using CAS.

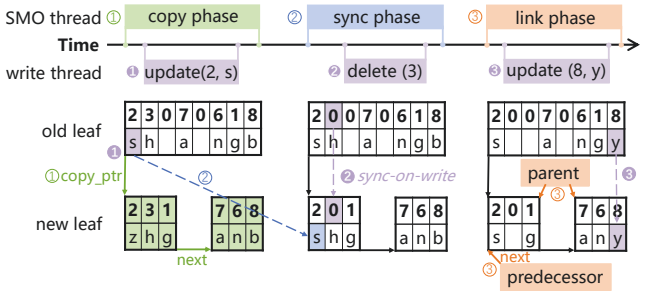


Figure 3: The procedure of *three-phase SMO* and *sync-on-write* when updates and deletions operate on an SMO leaf.

With the support of eADR, CAS can atomically modify and persist the synchronization. Similar to the update, the deletion also re-executes in the new leaf if it contains the target key. NBTree imposes low-overhead on *sync-on-write* because it only incurs one additional search on the new leaf and one CAS primitive.

During the post-copy phases of SMO, *sync-on-write* prevents lock-free UD operations from suffering the *lost update* anomaly. As illustrated in Figure 3, during the copy phase, any UD operation that happens on the old leaf (e.g. `update(2,s)`) will be synchronized to the new leaf in SMO's sync phase. However, UD operations happen after the copy phase (e.g. `update(8,y)`, `delete(3)`) may still be lost. To avoid the *lost update* anomaly, UD operations need to actively synchronize the modification by calling *sync-on-write*.

Through *sync-on-write* and *three-phase SMO*, we ensure that NBTree always maintains a consistent state that includes all committed operations after a crash. As we previously mentioned, SMO's *durability point* is when the new leaves replace the old leaf in PM by linking themselves into the key-value layer during the link phase. If a crash occurs before the *durability point*, the old leaf will remain in the key-value layer. During SMO, any modification must first operate on the old leaf. Therefore, as illustrated in Table 3, the old leaf always holds both the *latest* committed and uncommitted operations during SMO. The uncommitted operations in the old leaf won't cause inconsistency because the in-flight *sync-on-write* is unnecessary after discarding new leaves. If a crash occurs after the *durability point*, the new leaf will be linked into the key-value layer. At that time, all UD operations committed in the copy phase have been synchronized to the new leaf in the sync phase. For UD operations that happen after the copy phase, they are only committed when they write to a new leaf. As a result, new leaves hold the *latest* committed operations after the *durability point*. Besides, with the support of eADR, the new leaf is consistent as the *sync-on-write* is atomic. To summarize, the consistent leaf with the *latest* committed operations will survive whenever the crash happens.

SMO threads (sync phase) and UD threads (*sync-on-write*) may synchronize the same value to the new leaf concurrently. NBTree can serialize those synchronizations using the highest two bits of each entry's value. We will discuss this scenario in Section 5.

Sync-on-read. To deal with the potential inconsistency between the old leaf and the new leaves, the search operations employ the *sync-on-read* approach to synchronize the corresponding key-value

Table 3: The latest and clean leaves in different phases of SMO. (The *latest* leaf contains all committed writes. The *Clean* leaf does not contain any uncommitted dirty write.)

SMO Phase	Latest		Clean	
	old leaf	new leaf	old leaf	new leaf
Copy	✓		✓	
Sync	✓			✓
Link	✓	✓		✓
After SMO		✓		✓

entries from the old leaf to the new leaf. Specifically, NBTree searches the target key in both old and new leaves. If the returned results differ, the search operation updates or deletes the key-value in the new leaf to match the one in the old leaf. The overhead of the *sync-on-read* is as low as the *sync-on-write*.

Sync-on-read guarantees that a lock-free concurrent search returns the latest and committed version of a key-value entry. During SMO's sync phase, reading from either old or new leaves without performing *sync-on-read* may lead to the *stale read* or *dirty read* anomaly. As illustrated in Table 3, in the sync phase, the old leaf is possibly *dirty* because the UD operations might have not committed due to an on-going *sync-on-write*. Meanwhile, the new leaf is likely *stale* as the SMO thread may not have finished synchronizing the *latest* modification that happened during the copy phase. To address this problem, the search operation uses *sync-on-read* to synchronize the *latest* key-value from the old leaf to the new leaf. It makes sure that the target key-value pair in the new leaf is both the *latest* and *clean* before returning the search result.

Search operations on the leaf that is not in the sync phase can directly read the correct value without calling *sync-on-read*. Table 3 shows the destination of reads, which is the leaf that holds both the *latest* and *clean* key-value pairs. During the copy phase, incoming reads go to the old leaf, which is both the *latest* and *clean* since concurrent UD operations directly commit in the copy phase. After the sync phase, reads go to the new leaf. This is because previous UD operations that happened in the copy phase have already been synchronized to the new leaf, while later UD operations are committed once they are visible in the new leaf with the eADR support.

Cooperative SMO. In NBTree, concurrent insertions to the leaf during SMO employ *cooperative SMO*. The insertion thread that encounters a leaf with an in-flight SMO will help complete its SMO before continuing. NBTree uses atomic primitives, such as CAS, to coordinate multiple SMO threads, making sure only the fastest modification can be visible. In this way, instead of waiting for the completion of SMO, NBTree guarantees that SMO moves forward at the fastest speed, even when a certain SMO thread is suspended.

Specifically, in the copy phase, multiple SMO threads prepare new nodes respectively and use CAS primitive to atomically install the `copy_ptr`. In the sync phase, the synchronization of each key-value entry can also be completed cooperatively by using CAS primitive. In the link phase, NBTree uses CAS to link the new leaf into the metadata layer and the key-value layer. Then, NBTree uses HTM (described in Section 4.3) to atomically update the parent node and set `flag.link`.

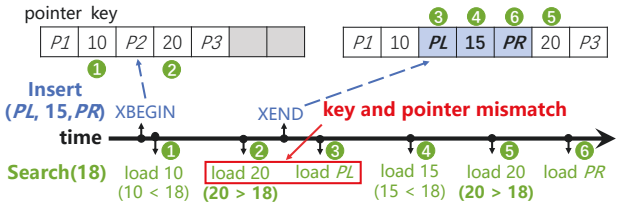


Figure 4: *Shift-aware search* on the inner node under the read-write conflict. (XBEGIN means the start of the transaction, XEND means the end of the transaction.)

Algorithm 1: Insert(K key, V val)

```
1 leaf = findLeaf(key);
2 pos = fetch_and_add(&leaf→num, 1);
3 if pos >= LEAF_NODE_SIZE then
4 |   leaf→setSMOBit();    // Set the highest bit of bitmap
5 |   SMO(leaf);
6 |   goto Line 1;
7 leaf→insert(key, val, pos);
8 if ! leaf→atomicSetBitmap(pos) then
9 |   goto Line 1;
```

4.3 Inner Node Operations

We propose a *shift-aware search* algorithm and use the *HTM-based update* to coordinate concurrent inner node operations.

HTM-based Update. NBTree uses HTM to atomically update inner nodes following FPTree [43]. HTM is an optimistic concurrency control tool that uses hardware transactions to make multiple writes atomically visible. It is non-blocking since it only aborts when conflicts are detected. Wrapping updates in HTM is efficient because the conflict of inner node modifications rarely happens. Moreover, updates do not expose intermediate states to other threads, which avoids the read thread viewing the inconsistent state.

Shift-aware Search. Although the modifications to the inner nodes are atomic, lock-free inner node search might find the wrong child pointer due to the read-write conflict. As illustrated in Figure 4, the search operation performs a linear search (`Search(18)`) to retrieve the key of 20 (②). However, before it fetches the corresponding pointer *P2*, an insertion (`Insert(PL,15,PR)`) modifies the node. Therefore, the search operation loads the wrong pointer *PL* (③) instead of *P2* or *PR*. As a result, an existing key might be missed in the search operations.

NBTree proposes a *shift-aware search* algorithm to ensure the correctness of lock-free inner node search despite concurrent write transactions (insertion or deletion) on the same node. As shown in Figure 4, before proceeding to the next level of the tree through *PL*, NBTree checks if the fetched key (20 in ②) has been shifted by concurrent write transactions. If so, NBTree re-searches the node from the current position (④-⑥), making sure that the target key lies within the sub-tree indicated by the returned pointer.

The *shift-aware search* algorithm always finds the correct pointer for the following two reasons. First, NBTree keeps searching and data shifting in the same direction. As shown in Figure 4, during the search, the insertion shifts the data from left to right. If the search proceeds in the same direction, then it never misses the newly inserted key. Inspired by FAST&FAIR, we maintain a `switch_counter` in each node, which is increased when the insertion and deletion on the inner node take turns. Second, we adopt the concurrency protocol of the B-link tree [33] to handle SMO. NBTree maintains a `high_key` (the largest key in the node) and `sibling_ptr` in each inner node. NBTree re-searches in the sibling node if the target key is less than the `high_key`, which indicates an SMO has happened on the node.

Shift-aware search is efficient for two reasons: (1) It achieves lock-free search without using HTM, avoiding transaction abortions and hardware overheads. (2) Linear search on small arrays is more efficient than binary search for its better cache locality, which is illustrated in FAST&FAIR [20].

5 IMPLEMENTATION

In this section, we first describe the implementation of NBTree: insert (Section 5.1), update/delete (Section 5.2), and search (Section 5.3). Then we present the limitation of our work (Section 5.4).

5.1 Insert

Algorithm 1 describes the insert operation. After locating the target leaf, NBTree occupies the next free slot using the atomic primitive (Line 2). If the leaf is full, NBTree initiates SMO by setting the `smo bit`, the highest bit of the `bitmap` (Line 3-5). NBTree commits the insertion by updating the `bitmap` using `CAS` (Line 8).

The procedure of SMO is listed in Algorithm 2. In the copy phase (Line 2-9), NBTree distributes valid entries with non-zero keys to newly allocated leaf nodes and constructs the metadata layers for the new leaves (Line 4-8). For each copied entry, NBTree sets the highest bit (`copy_bit`) of the value (Line 5). The `copy_bit` indicates that the value is copied from the old leaf in the copy phase.

In the sync phase (Line 11-24), NBTree sequentially obtains the valid entries in the old leaf (`entry`, Line 11) and new leaves (`syncEntry`, Line 13). If their keys match but values mismatch (Line 14-16), NBTree uses `CAS` to synchronize the lost update, clear the `copy_bit` and set the `sync_bit` (the second highest bit) of the target value in the new leaf. The `sync_bit` indicates the value is synchronized from the old leaf. `CAS` will abort if the `copy_bit` of the target value has been cleared. This ensures that any key-value pair is synchronized at most once during the sync phase. If the key of `entry` and `syncEntry` mismatch, NBTree synchronizes the lost deletion because it usually indicates the key of `syncEntry` has been deleted in the old leaf (Line 19-22). However, there are two exceptions that the key of `entry` has been deleted in the new leaf. (1) SMO has been completed by another SMO thread. As a result, the latest operations directly delete the entry in the new leaf without having to go through the old leaf. In this case, NBTree directly aborts SMO (Line 17). (2) A concurrent operation deletes the key of `entry` on the old leaf after the `entry` has been read. Meanwhile, the deletion has been synchronized by other threads before `syncEntry` is read. In this case, synchronization is not required (Line 18).

In the link phase (Line 26-36), NBTree uses `CAS` to link the leaf to both the key-value layer and the metadata layer (Line 27-28). If the previous leaf is in SMO, NBTree will join the *cooperative SMO* to

Algorithm 2: SMO(Leaf leaf)

```
1  if ! leaf→copy_ptr then // Copy phase
2      splitKey = leaf→findSplitKey();
3      leftLeaf, rightLeaf = allocNewNode();
4      while (entry = leaf→next()).key != 0 do
5          entry.val |= copy_bit;              // Set the copy_bit
6          copy(entry, leftLeaf, rightLeaf, splitKey)
7      end
8      setMetadata(leftLeaf, rightLeaf);
9      CAS(&(leaf→copy_ptr), NULL, leftLeaf);
10 if ! leaf→flag.sync then // Sync phase
11     while (entry = (key, val) = leaf→next()).key!=0 do
12         syncLeaf = key < splitKey ? leftLeaf : rightLeaf;
13         (k, v) = syncEntry = syncLeaf→next();
14         if k == key then
15             if (v & ~(copy_bit | sync_bit)) != (val &= ~(copy_bit |
                sync_bit)) then
16                 CAS(&syncEntry.val, v|copy_bit, val|sync_bit);
17         else if leaf→flag.link then return ;
18         else if ! entry.key then syncLeaf→prev() ;
19         else
20             syncEntry.key = 0;
21             mfence();
22             goto Line 13;
23     end
24     leaf→flag.sync = 1;
25 if ! leaf→flag.link then // Link phase
26     pred = findPredLeaf(key);
27     CAS(&(pred→data_ptr→next), leaf→data_ptr,
        leftLeaf→data_ptr);
28     CAS(&(pred→next), leaf, leftLeaf);
29     if pred→isSMO() then
30         SMO(pred);
31         goto Line 25;
32     xbegin();                          // Start an HTM transaction
33     if ! leaf→flag.link then
34         update_parent(leaf);
35         leaf→flag.link = 1;
36     xend();
```

Algorithm 3: Update(K key, V val)

```
1  leaf = findLeaf(key);
2  entry = leaf→update(key, val);
3  if leaf→isSMO() and leaf→copy_ptr then
4      (nKey, nVal) = nEntry = leaf→copy_ptr→find(key);
5      if nKey and ((v=nVal) & ~(copy_bit | sync_bit)) !=
        (val=entry.val) and v & (copy_bit | sync_bit) then
6          if ! CAS(&nVal, v, val|sync_bit) then
7              goto Line 5;
8  return true;
```

Algorithm 4: Delete(K key)

```
1  leaf = findLeaf(key);
2  leaf→delete(key);
3  if leaf→isSMO() and leaf→copy_ptr then
4      nEntry = leaf→copy_ptr→find(key);
5      if nEntry then
6          nEntry.key = 0;
7          mfence();
8  return true;
```

Algorithm 5: Search(K key)

```
1  leaf = findLeaf(key);
2  entry = leaf→find(key);
3  val = entry.key ? (entry.val & ~(copy_bit | sync_bit)) : 0;
4  if leaf→isSMO() and leaf→copy_ptr then
5      if leaf→flag.sync then return leaf→copy_ptr→search(key) ;
6      (nKey, nVal) = nEntry = leaf→copy_ptr→find(key);
7      if ! nKey then return 0 ;
8      if entry.key == 0 then
9          nEntry.key = 0;
10         mfence();
11     else if (nVal & ~(copy_bit | sync_bit)) != val then
12         CAS(&(nEntry.val), nVal|copy_bit, val|sync_bit);
13     val = nEntry.key ? (nEntry.val & ~(copy_bit | sync_bit)) : 0;
14 return val;
```

avoid the lost update of `next` pointer (Line 29-31). Finally, NBTree employs HTM to update the parent node and set the `flag.link` atomically (Line 32-36).

NBTree adopts the epoch-based garbage collection [13] to reclaim the old leaves. Epoch-based garbage collection recycles the old leaves two epochs after the end of SMO, which ensures that those leaves have no concurrent references.

5.2 Update/Delete

Algorithm 3 shows the process of the update operation. NBTree firstly performs an *in-place update* in the target leaf (Line 1-2). If the leaf is in the post-copy phases of its SMO and the target key exists in the new leaf but its value is not the latest, NBTree will perform *sync-on-write* via `CAS` (Line 5-6). *Sync-on-write* clears the `copy_bit`, which prevents the synchronization invoked by SMO threads from overwriting the current update. It also sets the `sync_bit` to distinguish itself from the update directly operated on the new leaf. When an update performs *sync-on-write*, the SMO might have been completed by other threads. At that time, the latest updates directly operate on the new leaf without setting `sync_bit` or `copy_bit`. *Sync-on-write* does not overwrite those new updates to keep the linearizability. Besides, if the value in the old leaf has been changed by new updates, NBTree will synchronize the latest one.

Algorithm 4 depicts the delete operation. Similar to the update, it will synchronize the deletion if necessary.

5.3 Search

The search operation is shown in Algorithm 5. For the inner node search (Line 1), we directly reuse the code of FAST&FAIR and add the key-checking procedure before returning the child pointer to detect if any update happens. For the leaf node search (Line 2-14), NBTree directly returns the search result on the target leaf if SMO

is not taking place or it is in the copy phase. Otherwise, NBTree searches the target key in the new leaf (Line 4-13). NBTree performs *sync-on-read* if the SMO is in the sync phase and the search results in two leaves mismatch (Line 8-13).

5.4 Limitation

In the current design, NBTree does not address the NUMA-related performance issues in PM. Similar to the prior work on persistent indexes [4, 20, 42], NBTree cannot scale well across multiple NUMA nodes for the following two reasons. First, due to the multi-socket cache coherence traffic, accessing PM on a remote NUMA node has much lower bandwidth than accessing local PM, according to recent studies [25, 27, 50]. Second, atomic primitives used in NBTree perform poorly across NUMA nodes. We plan to address the scalability issues of NBTree in the environment of multiple NUMA nodes in our future work.

6 EVALUATION

In this section, we evaluate the performance of NBTree against other state-of-the-art persistent B$^+$-Trees. We first describe our experiment setup (Section 6.1). Then, we perform single-threaded evaluation (Section 6.2), multi-threaded evaluation (Section 6.3), and YCSB evaluation (Section 6.4). After that, we compare the performance of indexes in two persistence modes (Section 6.5). Finally, we evaluate the performance in real-world systems (Section 6.6).

6.1 Experiment Setup

Testbed. Our testbed machine is a dual-socket Dell R750 server with two Intel Xeon Gold 6348 CPUs, the third generation Xeon Scalable processors that support eADR and TSX. Each CPU has 28 cores and a shared 42MB L3 cache, while each CPU core has a 48KB L1D cache, 32KB L1I cache, 1280KB L2 cache. The system is equipped with 512GB DRAM and 4TB PM (eight 256GB Barlow Pass DIMM per socket). In our evaluation, threads are pinned to NUMA node 0, and are only allowed to access the local DRAM and PM to avoid NUMA effects. We install a PM-aware file system (Ext4-DAX) in `fsdax` mode to manage PM devices. Then, we map large files into the virtual address using PMDK [23] to serve tree nodes allocation. We evaluate the performance of two persistence modes, eADR and ADR. To persist a `store`, we use `clwb` and `mfence` in ADR mode and solely use `mfence` in eADR mode.

Compared Systems. We compare NBTree against seven state-of-the-art persistent B$^+$-Trees, including NVTree, WB$^+$Tree, FPTree, RNTree, BzTree, FAST&FAIR, and uTree. We directly use the open-sourced code of uTree [15], FAST&FAIR [29], BzTree [30], and RNTree [38]. We borrow Liu's [38] implementations of WB$^+$Tree, FPTree, and NVTree. We skip the evaluation of the multi-threaded performance of NVTree and WB$^+$Tree as their implementations do not support concurrency control. For variable-sized keys, we only compare NBTree with BzTree as the implementations of other trees do not support this function.

Default Configuration. We warm up each tree with 16 million key-value pairs and then run enough time for different workloads. By default, we use 8-byte keys and values. For variable-sized keys and values, we store them in the external memory region, and only keep the pointers (48-bit) in indexes to indicate their addresses. The node size of each tree is configured to 1KB. We run all trees in eADR mode except in Section 6.5.

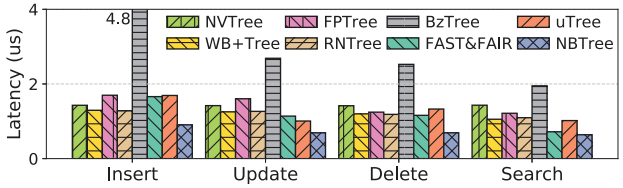


Figure 5: The average latency of base operations. (Single thread, uniform access.)

6.2 Single Thread Evaluation

In this section, we evaluate the single-thread performance of base operations (search, insert, update, and delete) in eADR mode. We run individual operations under random key-access distribution and then calculate the average latency.

As shown in Figure 5, NBTree achieves the lowest latency in every base operation. As the persistence overhead of a write is hidden by CPU caches in eADR mode, we attribute the good performance of NBTree to low *PM line reads*. In most cases, NBTree only causes one *PM line read* in each operation because it places the metadata of the leaf nodes in DRAM and uses fingerprints to filter the unmatched keys. The only PM overhead of NBTree comes from accessing the matched key-value pairs.

In contrast, as illustrated in Table 1, other persistent B$^+$-Trees produce more *PM line reads*, resulting in higher latency. We conclude the sources of *PM line reads* in the following: (1) Most of the B$^+$-Trees need to access the metadata of the leaf node. The metadata is often stored in different PM lines from the actual key-value pair, resulting in additional *PM line reads*. (2) Searching in the leaf node causes multiple *PM line reads*. FAST&FAIR and NVTree use linear search to locate the key-value pair, which needs to traverse half of the leaf on average. WB$^+$Tree, RNTree, and BzTree perform the binary search, which has a similar PM overhead to the linear search when the array size is small. (3) FAST&FAIR and BzTree produce extra *PM line reads* when they perform inner node search because they store inner nodes in PM. (4) uTree invokes additional *PM line reads* to access the sibling node in the linked list, which shows poor locality with the current node. (5) BzTree applies `PMwCAS` [51] to atomically persist the modification. Each `PMwCAS` produces multiple *PM line reads* to access the descriptors.

6.3 Multi-threaded Evaluation

We evaluate the multi-threaded performance of base operations under random key access distribution. As shown in Figure 6, NBTree achieves the highest throughput in each operation. Compared with other trees, the throughput of NBTree in 56 threads is 1.6-7.5× higher on insert, 1.5-5.0× higher on update, 2.0-4.9× higher on delete, 1.6-5.1× higher on search. This is primarily because NBTree minimizes both PM line reads and writes. Reducing *PM line writes* in eADR-enabled PM systems is important. The reason is that the modified PM lines in CPU caches are eventually evicted to PM with

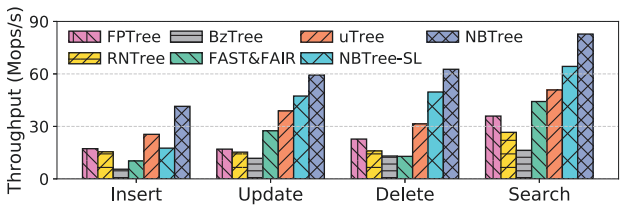


Figure 6: The throughput of base operations. (56 threads, uniform access. NBTree-SL applies the single-layer leaf nodes.)

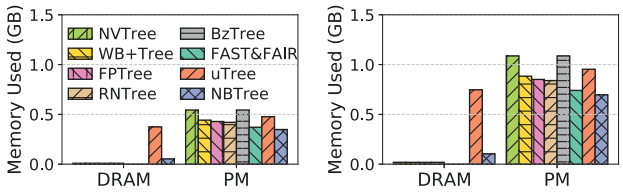


Figure 7: The space consumption of DRAM and PM after initializing the trees with 16M key-value entries (Left) and inserting 16M more key-value entries to the trees (Right).

low bandwidth. Multi-threaded writes can saturate CPU caches and WPQ, resulting in high latency. Besides, excessive *PM line reads* also degrade the multi-threaded performance due to the high latency.

NBTree scales well in the multi-threaded evaluation as it limits the *PM line read/writes* per operation to 1 in most cases. Figure 6 also shows that the two-layered leaf node design speeds up the insert operations by 2.4× because it absorbs metadata modifications in DRAM. Furthermore, although the UDS operations do not modify the metadata due to the optimization of NBTree, the two-layered leaf node design still improves their throughputs by up to 29% due to fewer *PM line reads*. Meanwhile, the additional DRAM consumption from the metadata layer is tolerable. As shown in Figure 7, the ratio of DRAM and PM consumption in NBTree is around 1:7, which is close to the ratio of our testbed configuration (1:8). In practice, this ratio will be much smaller if the value size is large because the values only reside in PM.

As shown in Table 1, other persistent indexes have lower scalability for their high *PM line read/writes*. We have analyzed the cost of PM reads in detail in Section 6.2. Compared with NBTree, other trees produce extra *PM line writes* in the following aspects: (1) They modify the persistent metadata of the leaf nodes for various usages, such as correct recovery, traversal acceleration, and concurrency control. (2) BzTree produces the most *PM line writes* because it needs to record a descriptor in each PMwCAS, resulting in the lowest scalability. (3) FAST&FAIR causes additional *PM line writes* to maintain the order of leaf nodes. As a result, its throughputs on insertions and deletions are low.

6.4 YCSB Evaluation

In this section, we evaluate the performance of persistent B$^+$-Trees with real-world YCSB [9] workloads. We generate the skewed (zipfian key access distribution) and read-write mixture workloads

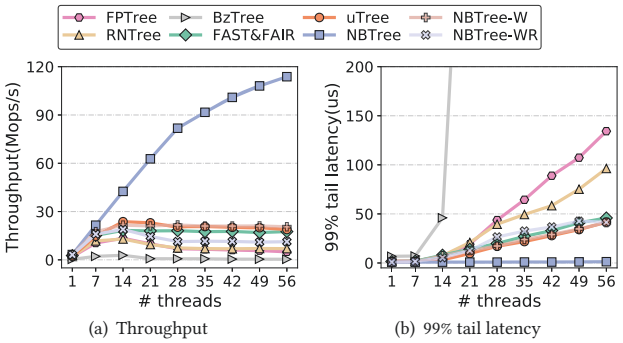


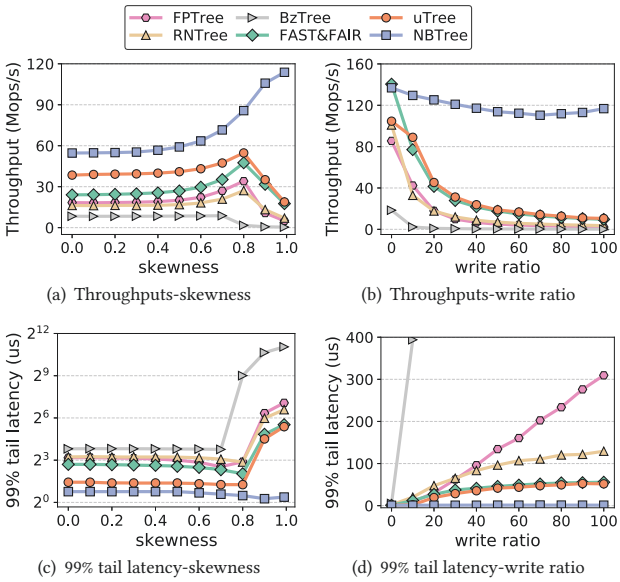
Figure 8: The throughput and 99% tail latency under YCSB workload. (Read:write=50:50, skewness=0.99. NBTree-W disables lock-free write schemes. NBTree-WR disables both lock-free write and read schemes.)

based on YCSB. By default, the write operations in the workload are upsert. Upsert will insert a new key if the target key does not exist. Otherwise, it performs an update.

Overall Evaluation. Figure 8 reports the evaluation results under YCSB workload (read:write=50:50) in a zipfian key access distribution with the default 0.99 skewness. We observe that NBTree has almost linear scalability on throughput and near-constant 99% tail latency with the increase of threads, while other trees only scale up to 14 threads. In 56 threads, NBTree achieves 6.0× higher throughput and 32× lower 99% tail latency than other trees.

We attribute the high performance of NBTree to our efficient lock-free design. The skewed workload often introduces a lot of leaf-level conflicts. The lock-free leaf node operations in NBTree can scale well under high contentions. When operating on the leaf that is not performing SMO, NBTree only employs a small number of atomic primitives to support lock-free access. When operating on the leaves in SMO, UDS operations fix the potential anomaly by our proposed techniques, such as *three-phase SMO*, *sync-on-write*, and *sync-on-read*, which only introduces at most one additional leaf node search and one CAS primitive. Concurrent insertions also apply *cooperative SMO* to achieve lock-free accesses. Besides, as the writes happen infrequently in inner nodes, our proposed *shift-aware search* and *HTM-based* updates can also scale well.

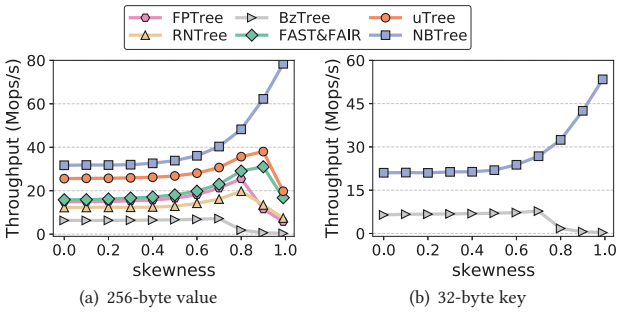
To better demonstrate the impact of our lock-free design and illustrate the performance gap between NBTree and other indexes, we implement two additional versions of NBTree. NBTree-W disables our lock-free write schemes and applies node-grained write-locks instead. As shown in Figure 8, NBTree achieves 5.5× higher throughput than NBTree-W due to our lock-free write design. Meanwhile, FAST&FAIR and uTree achieve similar performance with NBTree-W as they use similar schemes for concurrency control. NBTree-WR further disables our lock-free read approaches and replaces them with HTM-based read, which is used in FPTree and RNTree. We find that NBTree-W achieves 1.8× higher throughput than NBTree-WR. Moreover, NBTree-WR is still 1.9× faster than RNTree and 2.6× faster than FPTree due to our PM-friendly design. As for BzTree, it achieves lock-free by utilizing PMwCAS, which is an optimistic approach implemented by a series of CAS and RDCSS [19]



(a) Throughputs-skewness (b) Throughputs-write ratio

(c) 99% tail latency-skewness (d) 99% tail latency-write ratio

Figure 9: The performance varying the write ratio and skewness under YCSB workload.



(a) 256-byte value (b) 32-byte key

Figure 10: The YCSB performance of the indexes with large key/value. (Read:write=50:50, skewness=0.99.)

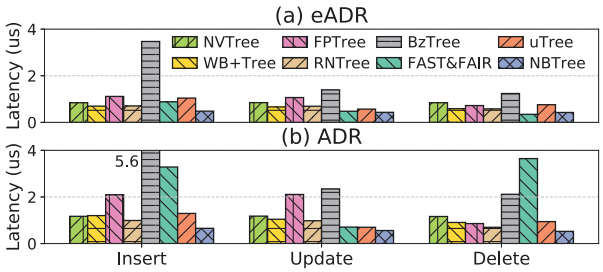


Figure 11: The PM overhead of the leaf nodes in eADR/ADR mode. (Single thread, uniform access.)

operations. However, PMwCAS is vulnerable to high contentions and brings high software overhead [35]. Therefore, BzTree has the worst performance despite its lock-free design.

Effect of Skewness. Figure 9(a) and Figure 9(c) report the evaluation results when we vary the skewness (zipfian coefficient) in YCSB workload (read:write=50:50). We notice that NBTree has better performance with the increase of skewness. The reason is two-fold: (1) Our efficient lock-free designs prevent the concurrency control from becoming the performance bottleneck. (2) Our cache-friendly designs, including *in-place update* and *log-structured insert*, have larger effects with the increase of the skewness. Those designs increase the possibility of *write combining* and *write hits* in CPU caches, which saves the PM write bandwidth.

With the increase of skewness, other trees have a slight performance improvement when the skewness is less than 0.8, which benefits from better cache utilization. When the skewness is larger than 0.8, they have a dramatic performance drop because they cannot scale well under frequent leaf-level contentions.

Effect of Write Ratio. Figure 9(b) and Figure 9(d) show the evaluation results when we vary the write ratio of the YCSB workload (skewness=0.99). We observe that the performance gap becomes larger between NBTree and other B$^+$-Trees with the increase of write ratio. NBTree achieves 11× higher throughput and 43× lower 99% tail latency under the write-only workload. This is because NBTree applies efficient lock-free algorithms for both reads and writes. In contrast, previous works focus on the optimization of concurrent reads but do not support efficient concurrent writes.

Effect of Large Key/Value. Figure 10 reports the evaluation of indexes when the key-value size is larger than 8 bytes. We observe that NBTree still achieves significantly higher throughput than other indexes, especially when the skewness is high. However, the performance gap between NBTree and other indexes becomes smaller. The reason is two-fold: (1) The PM write bandwidth is dominated by persisting large values, dwarfing the performance benefits from our optimization on NBTree. (2) NBTree employs the 8-byte pointers to indicate variable-sized keys, which incurs a lot of pointer dereferences in inner node search. In contrast, Bztree continuously stores the variable-sized keys in a single node, which avoids expensive pointer dereferences.

6.5 Comparison of Persistence Modes

In this section, we first compare the performance of persistent indexes between two persistence modes: ADR and eADR. Then, we evaluate the performance of NBTree in the pure DRAM setting.

Figure 11 reports the PM overhead of the leaf nodes in two persistence modes. Firstly, we find that the PM overhead of all trees is reduced in eADR mode, compared with ADR mode. The primary reason is that the latency on the critical path caused by flush instructions is removed. For example, FAST&FAIR has a significant performance improvement because eADR minimizes the large overhead of data shifts to keep arrays sorted. The performance of BzTree also improves a lot because a large number of flush instructions needed by PMwCAS are removed. However, PMwCAS is still costly due to excessive PM accesses, resulting in the poor performance of BzTree. Secondly, we observe that NBTree has the lowest PM overhead in ADR mode. This is attributed to our PM-friendly designs, which cost only one flush instruction in each write operation.

Figure 12 shows the performance of two persistence modes under YCSB workload (56 threads, read:write=50:50, skewness=0.99). We

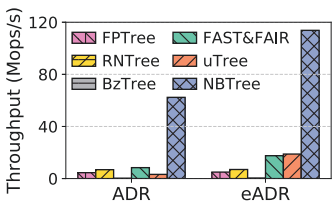


Figure 12: The performance in ADR/eADR mode under YCSB workloads.

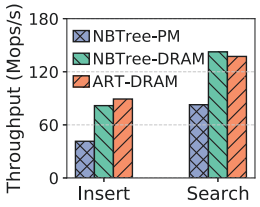


Figure 13: The performance of base operations on the DRAM setting.

have the following four observations. First, NBTree achieves the most performance improvement with the eADR support. This is because the cache-friendly designs (e.g. *in-place update*, and *log-structured insert*) of NBTree take effect in eADR mode. Second, NBTree also performs the best among all trees in ADR mode due to the lock-free design. However, the *dirty read* anomaly is likely to happen in ADR mode because the CPU caches are volatile. Third, uTree and FAST&FAIR also speed up significantly because of the better cache utilization in eADR mode. Fourth, RNTree, FPTree, and BzTree do not benefit from eADR because their concurrency control is vulnerable to high contentions.

Figure 13 shows the performance of NBTree in a pure DRAM setting (uniform access). We observe that NBTree achieves 1.7× higher throughput on search operations and 1.8× higher throughput on insert operations when running on DRAM. We attribute this to the limited bandwidth and high latency of PM compared with DRAM. Meanwhile, NBTree has comparable performance with state-of-the-art indexes (e.g. ART [34]) designed for DRAM because NBTree avoids using redundant operations to guarantee durability.

6.6 End-to-End Evaluation

Redis [44] is a popular in-memory key-value store using a hash table as its index. We use the multi-threaded version of the Redis [48] and replace its internal index with our evaluated trees. We run 28 threads on the Redis server in our evaluation. NBTree achieves the throughput of 1719.4 Kops/s, which is 1.13-1.53× higher than other state-of-the-art persistent B$^+$-Trees under the YCSB-A workload [9]. The evaluation results confirm our previous experiments. Note that the performance gap among indexes becomes smaller due to the high software overhead of Redis.

7 RELATED WORK

Indexes Optimized for PM. In ADR-based PM systems, the slow write is the performance bottleneck of persistent indexes because flush instructions introduce high latency on the critical path. Therefore, previous works have proposed various ways to optimize the write performance of persistent indexes. Most persistent B$^+$-Trees, such as WB$^+$Tree [3], NVTree [56], FPTree [43], RNTree [39], and LB$^+$Tree [37], implement the unsorted leaf nodes. The reason is that inserting or deleting an element of the sorted leaf node produces lots of PM writes to shift array elements. FPTree firstly proposes the selective persistence technique to place inner nodes in DRAM, which speeds up the inner node operations. The volatile inner nodes can be reconstructed from persistent leaf nodes after a crash. RNTree and ROART [40] further remove the flush instructions when modifying reconstructable metadata in the leaf node to reduce the critical path latency. uTree places the sorted leaf nodes in DRAM and adds a persistent shadow list-based layer to ensure crash consistency. In this way, uTree offloads the expensive structural refinement operations (SRO) to DRAM. Persistent hash indexes, such as level hashing [61], path hashing [60], and CCEH [42], also make lots of efforts to write-efficient designs.

Concurrency Control for Persistent Indexes. Previous works propose various concurrency control strategies for persistent indexes to leverage the benefits of multi-core processors. For persistent B$^+$-Trees, FPTree proposes the selective concurrency technique, which handles the concurrency of inner nodes by HTM and serializes the accesses of leaf nodes by the node-grained locks. It improves the scalability in the situation with infrequent contentions but performs poorly in skewed workloads. Based on FPTree, RNTree excludes some slow persistent instructions out of the critical section to achieve more concurrency in the leaf nodes. FAST&FAIR designs a lock-free search algorithm inspired by B-link tree [33], which tolerates the transient inconsistent states caused by write transactions. It improves search performance but tends to cause consistency problem [32]. uTree supports lock-free concurrency control for the list layer but still uses the coarse-grained locks in the leaf nodes. BzTree [2] develops the first lock-free persistent B$^+$-Tree with `PMwCAS` [51], which guarantees both the atomicity and persistence of multi-word writes. However, `PMwCAS` causes high software overhead, and it is also vulnerable to high contentions [35]. As for hash-based persistent indexes, most of them are lock-based, such as level hashing [61], CCEH [42], and CMAP [22]. P-CLHT [32] is a persistent version of CLHT [11], which supports lock-free search. Clevel hashing [6] is the concurrent version of level hashing, which uses atomic primitives to implement lock-free algorithms. However, it doesn't address the *dirty read* anomaly.

8 CONCLUSION

Existing persistent indexes suffer from low scalability and high PM overhead. Fortunately, the new platform feature for persistent memory (PM) called eADR offers opportunities to build lock-free persistent indexes and unleash the potential of PM. In this paper, we propose a lock-free PM-friendly B$^+$-Tree, named NBTree, which leverages the benefits of eADR. To achieve high scalability, NBTree develops lock-free concurrency control strategies. To reduce PM overhead, NBTree proposes a two-layer leaf node structure, which reduces PM line accesses and improves cache utilization. The real-world YCSB evaluation shows that NBTree achieves up to 11× higher throughput and 43× lower 99% tail latency than state-of-the-art persistent B$^+$-Trees.

ACKNOWLEDGMENTS

This work was supported by Natural Science Foundation of Shanghai (No. 21ZR1433600, 22ZR1435400), and Shanghai Municipal Science and Technology Major Project (No. 2021SHZDZX0102). We also thank Liangxu Nie and Yanyan Shen for their assistance and valuable feedback.

REFERENCES

[1] Andrei Alexandrescu. 2004. Generic< Programming>: Lock-Free Data Structures. In *C++ Users Journal*. Citeseer.

[2] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. BzTree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment* 11, 5 (2018), 553–565.

[3] Shimin Chen and Qin Jin. 2015. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment* 8, 7 (2015), 786–797.

[4] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. 2020. uTree: a persistent B+-tree with low tail latency. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2634–2648.

[5] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1077–1091.

[6] Zhangyu Chen, Yu Huang, Bo Ding, and Pengfei Zuo. 2020. Lock-free Concurrent Level Hashing for Persistent Memory. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*. 799–812.

[7] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 105–118.

[8] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 133–146.

[9] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[10] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-free concurrent data structures. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 373–386.

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

[12] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*. 1–15.

[13] Keir Fraser. 2004. *Practical lock-freedom*. Technical Report. University of Cambridge, Computer Laboratory.

[14] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A persistent lock-free queue for non-volatile memory. *ACM SIGPLAN Notices* 53, 1 (2018), 28–40.

[15] Storage Research Group. 2020. *uTree*. Tsinghua University. Retrieved August 26, 2021 from https://github.com/thustorage/nvm-datastructure

[16] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. 2020. Understanding the idiosyncrasies of real persistent memory. *Proceedings of the VLDB Endowment* 14, 4 (2020), 626–639.

[17] Jim Handy. 2015. Understanding the intel/micron 3d xpoint memory. *Proc. SDC* (2015).

[18] Timothy L Harris. 2001. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing*. Springer, 300–314.

[19] Timothy L Harris, Keir Fraser, and Ian A Pratt. 2002. A practical multi-word compare-and-swap operation. In *International Symposium on Distributed Computing*. Springer, 265–279.

[20] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *16th {USENIX} Conference on File and Storage Technologies ({FAST} 18)*. 187–200.

[21] Intel. 2016. *Deprecating the PCOMMIT Instruction*. Retrieved August 26, 2021 from https://software.intel.com/content/www/us/en/develop/blogs/deprecate-pcommit-instruction.html

[22] Intel. 2019. *Key/Value Datastore for Persistent Memory*. Retrieved August 26, 2021 from https://pmem.io/pmemkv/index.html

[23] Intel. 2020. *Persistent Memory Development Kit*. Retrieved August 26, 2021 from http://pmem.io/pmdk

[24] Intel. 2021. *eADR: New Opportunities for Persistent Memory Applications*. Retrieved August 26, 2021 from https://software.intel.com/content/www/us/en/develop/articles/eadr-new-opportunities-for-persistent-memory-applications.html

[25] Intel. 2021. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Retrieved August 26, 2021 from https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf

[26] Tomas Karnagel, Roman Dementiev, Ravi Rajwar, Konrad Lai, Thomas Legler, Benjamin Schlegel, and Wolfgang Lehner. 2014. Improving in-memory database index performance with Intel® Transactional Synchronization Extensions. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 476–487.

[27] Wook-Hee Kim, R Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*. 424–439.

[28] R Madhava Krishnan, Wook-Hee Kim, Xinwei Fu, Sumit Kumar Monga, Hee Won Lee, Minsung Jang, Ajit Mathew, and Changwoo Min. 2021. Tips: Making volatile index structures persistent with DRAM-NVMM tiering. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*.

[29] Data Intensive Computing Lab. 2020. *FAST&FAIR*. SKKU/UNIST. Retrieved August 26, 2021 from https://github.com/DICL/FAST_FAIR

[30] Data-Intensive Systems Lab. 2018. *BzTree*. Simon Fraser University. Retrieved August 26, 2021 from https://github.com/sfu-dis/bztree

[31] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh. 2017. {WORT}: Write optimal radix tree for persistent memory storage systems. In *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*. 257–270.

[32] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 462–477.

[33] Philip L Lehman and S Bing Yao. 1981. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems (TODS)* 6, 4 (1981), 650–670.

[34] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 38–49.

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

[36] 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)*. IEEE, 302–313.

[37] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+ Trees: Optimizing persistent index performance on 3DXPoint memory. *Proceedings of the VLDB Endowment* 13, 7 (2020), 1078–1090.

[38] Mengxing Liu. 2020. *RNTree*. Tsinghua University. Retrieved August 26, 2021 from https://github.com/liumx10/ICPP-RNTree

[39] Mengxing Liu, Jiankai Xing, Kang Chen, and Yongwei Wu. 2019. Building Scalable NVM-based B+ tree with HTM. In *Proceedings of the 48th International Conference on Parallel Processing*. 1–10.

[40] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. 2021. ROART: Range-query Optimized Persistent ART. In *19th {USENIX} Conference on File and Storage Technologies ({FAST} 21)*. 1–16.

[41] Maged M Michael. 2002. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. 73–82.

[42] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. 2019. Write-optimized dynamic hashing for persistent memory. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*. 31–44.

[43] 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*. 371–386.

[44] Redis. 2009. *Redis*. Retrieved August 26, 2021 from https://redis.io

[45] Andy Rudoff. 2017. Persistent memory programming. *Login: The Usenix Magazine* 42, 2 (2017), 34–40.

[46] A Rudoff. 2020. Persistent memory programming without all that cache flushing. *SDC* (2020).

[47] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, Roy H Campbell, et al. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory.. In *FAST*, Vol. 11. 61–75.

[48] Vipshop. 2017. *Redis*. Retrieved August 26, 2021 from https://github.com/vipshop/vire

[49] Haris Volos, Andres Jaan Tack, and Michael M Swift. 2011. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 91–104.

[50] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. 2021. Nap: A black-box approach to numa-aware persistent memory indexes. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Virtual*.

[51] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. 2018. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 461–472.

[52] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2020. Characterizing and modeling non-volatile memory systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 496–508.

[53] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In *USENIX ATC 17*. 349–362.

[54] Jian Xu and Steven Swanson. 2016. {NOVA}: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*. 323–338.

[55] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*. 169–182.

[56] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*. 167–181.

[57] Bowen Zhang. 2021. *The consistency issue in the inner node search of FAST&FAIR*. Retrieved December 13, 2021 from https://github.com/DICL/FAST_FAIR/issues/16

[58] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. 2019. Ziggurat: a tiered file system for non-volatile main memories and disks. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*. 207–219.

[59] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. 2019. DPTree: differential indexing for persistent memory. *Proceedings of the VLDB Endowment* 13, 4 (2019), 421–434.

[60] Pengfei Zuo and Yu Hua. 2017. A write-friendly and cache-optimized hashing scheme for non-volatile memory systems. *IEEE Transactions on Parallel and Distributed Systems* 29, 5 (2017), 985–998.

[61] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 461–476.

[62] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient lock-free durable sets. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–26.