Received March 5, 2022, accepted April 18, 2022, date of publication April 28, 2022, date of current version May 5, 2022.

*Digital Object Identifier 10.1109/ACCESS.2022.3170916*

# Pivotal B+tree for Byte-Addressable Persistent Memory

**JONGHYEON YOO[1,2], HOKEUN CHA[3], WONBAE KIM[4], WOOK-HEE KIM[5], SUNG-SOON PARK[6,7], AND BEOMSEOK NAM[1], (Member, IEEE)**
[1]Department of Software, Sungkyunkwan University, Suwon 16419, South Korea
[2]Kakao Corporation, Jeju-do 63309, South Korea
[3]Department of Computer Sciences, University of Wisconsin –Madison, Madison, WI 53706, USA
[4]Ulsan National Institute of Science and Technology, Ulsan 44919, South Korea
[5]Department of Software, Konkuk University, Seoul 05029, South Korea
[6]Gluesys, Anyang 14028, South Korea
[7]Department of Computer Engineering, Anyang University, Anyang 14028, South Korea

Corresponding author: Beomseok Nam (bnam@skku.edu)

This work was supported in part by the Institute for Information & communication Technology Planning & evaluation (IITP) under Grant 2018-0-00549, Grant 2021-0-01817, and Grant 2021-0-00862; and in part by the Electronics and Telecommunications Research Institute (ETRI) through the Korean Government under Grant 20ZS1310.

**ABSTRACT** Over the past few years, various indexes have been redesigned for byte-addressable persistent memory. In this work, we design and implement *PB+tree* (Pivotal B+tree) that resolves the limitations of state-of-the-art fully persistent B+trees. First, PB+tree reduces the number of expensive shift operations by up to half by managing two sub-arrays separated by a *pivot* key. Second, PB+tree reads cachelines in ascending order, which makes PB+tree benefit from hardware prefetchers and run faster than state-of-the-art persistent B+trees that access cachelines in non-contiguous or descending order. Third, PB+tree employs an optimistic lock-free search algorithm to avoid repeatedly visiting the same tree node. Although the optimistic lock-free search algorithm involves a risk of visiting incorrect child nodes, PB+tree guarantees correct search results using the *lazy correction* algorithm using doubly linked sibling pointers. Our performance study shows that PB+tree outperforms the state-of-the-art fully persistent indexes by a large margin. A search algorithm without optimistic locking risks visiting the wrong child node, but PB+tree uses a *lazy correction* algorithm with doubly linked sibling pointers to ensure correct search results. Our performance studies show that PB+trees outperform state-of-the-art fully persistent indexes.

**INDEX TERMS** Tree data structures, fault tolerance, database concurrency operations.

## I. INTRODUCTION

Since Intel's Optane DC Persistent Memory (DCPM) [1] has broken the boundaries between memory and storage devices, numerous efforts have been made to re-design various system software [2]–[15]. Efficient indexing techniques are essential in nearly all software domains, and byte-addressable PM poses new challenges for them because byte-addressability, failure-atomicity, memory coherence, and durability have not been considered altogether in legacy data structure designs.

Data structures can be classified into two groups - ordered indexes and unordered indexes. Unordered indexes such as hash tables support only point queries, but their point query performance is known to be superior to ordered indexes. However, ordered indexes are still preferred in many application domains including database systems because B+trees can process more complex queries such as range queries.

In the past few years, various ordered indexes for persistent memory have been designed, such as FAST and FAIR [16], FPTree [17], NV-Tree [18], wB+-tree [19], WORT [20], [21], DP-tree [22], and $\mu$-Tree [23], just to name a few. However, these previous studies have overlooked the performance impact of how cachelines are accessed and prefetched by hardware. For example, WORT [20], a radix tree extension, stores data in multiple levels, which implies high read and write amplification for small accesses because the 3D XPoint access granularity is 256 Bytes [24]. wB+tree [19] and FP-tree [17] are also sub-optimal since they limit the tree node size. With a small tree node size, the performance gains that can be obtained from the hardware prefetcher and

The associate editor coordinating the review of this manuscript and approving it for publication was Gianmaria Silvello.

memory level parallelism are limited [25]. FAST and FAIR B+tree [16] does not limit the tree node size, thus it can get a performance benefit from the hardware prefetcher and memory level parallelism. However, FAST and FAIR B+tree has to access a tree node in descending order if the node was updated by a delete operation, which prevents it from getting performance gain from hardware prefetchers. Besides, as it performs a large number of shift operations, its performance degrades sharply as we increase the node size.

In this work, we present *Pivotal B+tree (PB+Tree)* that simplifies memory access patterns for a variety of queries. The contributions of this work are as follows.

- PB+Tree employs a novel tree node format that uses the *pivot key*, allowing it to double the node size without increasing the number of shift operations. The sorted array format is known to perform better than append-only methods since it avoids a level of indirection [16]. However, sorted arrays require a large number of expensive shift operations especially for insertion workloads with descending keys. A pivot key allows PB+tree to split a sorted array into two sub-arrays. With the two separate sub-arrays, PB+tree reduces the number of shift operations by up to half. As a result, PB+tree better utilizes CPU caches and significantly improves the range query performance, which is one of the most important query types that make ordered indexes more popular than hash-based indexes.
- PB+tree reads the cachelines of the tree node in ascending order to take advantage of hardware prefetchers. Append-only methods that require a level of indirection access cachelines in an irregular manner, which makes hardware prefetchers fail to predict access patterns. FAST and FAIR B+tree, the only persistent B+tree that stores key-values in sorted arrays often reads cachelines in descending order to enable lock-free search, which we show perform slower than accessing cachelines in ascending order. In contrast, PB+tree scans cachelines in a tree node always in ascending order, which makes memory access faster.
- PB+tree simplifies lock-free search algorithm with a novel optimistic lock-free tree traversal algorithm with the lazy correction method. The lazy correction method corrects any transient inconsistent tree traversals, which FAST and FAIR B+tree suffers from. Our experiments show that the state-of-the-art lock-free search implementation of open source FAST and FAIR B+tree is far from correct and fails to run complex TPC-C database benchmarks. We show that our lazy correction method for lock-free search does not have correctness issues and achieves up to 10.5x higher throughput than FAST and FAIR B+tree, which we fixed with the crabbing protocol for correct query results.

The rest of this paper is organized as follows. In Section II, we present the background and the challenges of the state-of-the-art byte-addressable persistent B+trees. In Section III, we present the design of our PB+tree. In Section IV, we evaluate the performance of PB+tree against FAST and FAIR B+tree and wB+tree. In Section V, we conclude the paper.

## II. BACKGROUND

### A. CHALLENGES IN PERSISTENT INDEX

One of the key challenges in designing a persistent data structure is to ensure the order of memory writes because dirty cachelines in CPU caches can be flushed to persistent memory prematurely. Flushing partially updated dirty cachelines and reordering memory writes can make data structures inconsistent and such transient inconsistency becomes durable and exposed to other transactions if a system crashes.

To resolve this problem, various works [17]–[19], [26] proposed *append-only update* methods such that the consistent part of data structures remains unmodified. However, the append-only update methods give up storing keys in sorted order and require a level of indirection that hurts memory locality, making it difficult to utilize hardware prefetchers due to irregular memory access patterns.

Another challenge for persistent B-trees is the failure-atomicity of tree rebalancing operations, i.e., tree rebalancing operations need failure-atomic updates to multiple cachelines. Since it is impossible to atomically update multiple cachelines in current processor designs, previous works such as NV-tree [18], FP-tree [19], DP-tree [22], and $\mu$-Tree [23] proposed *selective persistence* methods. I.e., internal tree nodes are stored in volatile DRAM instead of persistent memory. If internal nodes are stored in DRAM, we do not need to make rebalancing operations failure-atomic because they will be lost anyway if a system crashes.

Selective persistence not only helps avoid explicit logging but also improves the write performance. However, although the reconstruction of internal tree nodes is feasible for certain applications, it is not always possible for database systems to keep all the internal tree nodes in small DRAM if they manage a large number of tables. If the working set of queries exceeds the memory capacity, the database systems need to discard volatile internal tree nodes, which should be reconstructed later. Even if the selective persistence schemes have a set of sorted records in their persistent leaf nodes, it is not free to reconstruct the whole tree structure for each different incoming query. Therefore, in this work, we only consider fully persistent indexes that store all indexing components in persistent memory.

### B. FAST AND FAIR B+tree

FAST and FAIR B+tree [16] is a persistent B-link-tree that leverages the hardware prefetchers. Hardware prefetching is limited to sequential and stride access patterns in commercial platforms [25]. Therefore, FAST (Failure-Atomic ShifT) algorithm shifts sorted keys in a tree node in a data-dependent way to leverage hardware prefetchers and also to reduce the number of calls to memory barrier operations.

**FAST:** In a tree structure, pointers always have unique memory addresses. Because of this property, read transactions can detect and ignore a transient inconsistent state partially updated by a write transaction. When we insert a new key-value record into a sorted array, we shift records to make space for the new record. As a result, one pointer in the array will be duplicated in the adjacent positions. Since duplicate pointers are not normal in B+tree, subsequent transactions can detect duplicate pointers and ignore the key in between them. Based on this observation, FAST algorithm shifts keys and pointers in tandem without using expensive logging.

**FAIR:** As a fully persistent index, FAST and FAIR B+tree stores all internal nodes in persistent memory, but it avoids expensive logging for tree rebalancing operations by connecting an overflow node and its new sibling node using a sibling pointer as in B-link tree. That is, using the sibling pointer, two sibling nodes are logically combined and exposed as a single virtual node to other concurrent read transactions. During rebalancing operations, redundant entries can be found in the overflow node and its new sibling node. But, the FAIR (failure-atomic in-place rebalancing) algorithm of FAST and FAIR B+tree allows read transactions to detect and ignore redundant entries, which ensures the correctness and invariants of the index.

**Lock-Free Search:** If every store instruction in FAST and FAIR algorithms guarantees that no read transaction will ever access inconsistent tree nodes, it is guaranteed that read transactions will always return the correct results. As such, FAST and FAIR algorithms enable non-blocking lock-free search.

### C. LIMITATIONS OF FAST AND FAIR B+tree

Although FAST and FAIR B+tree is shown to outperform other state-of-the-art persistent indexes [16], we find there are still rooms for performance improvement. The major limitations of FAST and FAIR B+tree that we find are three folds.

First, the insertion performance of FAST and FAIR B+tree degrades as we increase the tree node size. Although FAST algorithm benefits from hardware prefetchers and memory level parallelism of modern architectures, the number of shift operations linearly increases as the size of the tree node grows.

Second, the FAST algorithm uses the *scan direction flag* to enable lock-free scans and guides the direction in which subsequent read transactions should scan the tree node. In particular, if a sorted array is updated by a delete transaction, the FAST algorithm shifts array elements in descending order and subsequent read transactions have to read the array from right to left. The scan direction flag not only increases the code complexity of search functions but also memory access latency [25].

Data Cache Unit (DCU) prefetcher (also known as the streaming prefetcher) in Intel Xeon CPUs prefetches data from persistent memory to L1 cache only if the data is sequentially accessed in ascending order. I.e., if we read tree nodes in descending order, it fails to leverage the DCU prefetcher.

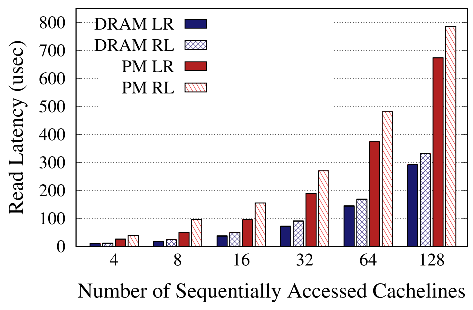


**FIGURE 1.** Performance Effect of L2 Hardware Prefetcher.

Intel Xeon CPUs also use an L2 hardware prefetcher that changes the prefetching direction, but it does so only after it suffers from lots of cache misses. I.e., changing the scan direction for each small tree node is not friendly to the current CPU designs.

In our experiments shown in Figure 1, we observe hardware prefetchers in Intel Xeon CPUs typically work best with ascending access order. In the experiments, we measure the performance of linear scanning with a microbenchmark on a testbed machine that we will describe in Section IV. The performance of linear scanning in ascending order, denoted as `LR` is up to 14% faster than that of linear scanning in descending order, denoted as `RL`.

Third, the lock-free search algorithm of FAST and FAIR B+tree does not limit the number of accesses to the same tree node. In the lock-free search algorithm, a read transaction determines in which direction it scans the node, and it double checks whether the flag remains unchanged after the scan. If the flag is found to be different, the read query has to scan the node once again so that it can read the updated array. If a read transaction scans the array slowly, and concurrent write transactions keep inserting and deleting the array, the read transaction must repeat to scan the array an unbounded number of times.

A more serious problem with the lock-free search algorithm of FAST and FAIR B+tree is that it is vulnerable to incorrect search results due to various causes. For example, Hwang *et al.* [16] mentioned that `gcc` compiler with `-O3` option changes the order of machine instructions and leads to incorrect search results. In our experiments, even with lower optimization levels, we found FAST and FAIR B+trees return incorrect search results occasionally.

## III. PIVOTAL B+tree

Pivotal B+tree is a fully persistent B+tree that employs the failure-atomic shift and in-place rebalancing algorithms, but it resolves the limitations of FAST and FAIR B+tree.

Figure 2 shows the internal node structure of Pivotal B+tree. The structure of the leaf node is almost the same with the internal node except that values are stored instead of child pointers. The header of PB+tree node contains metadata about each tree node, i.e., the node level, the *split key (the smallest key of the leftmost sub-tree)*, the leftmost child

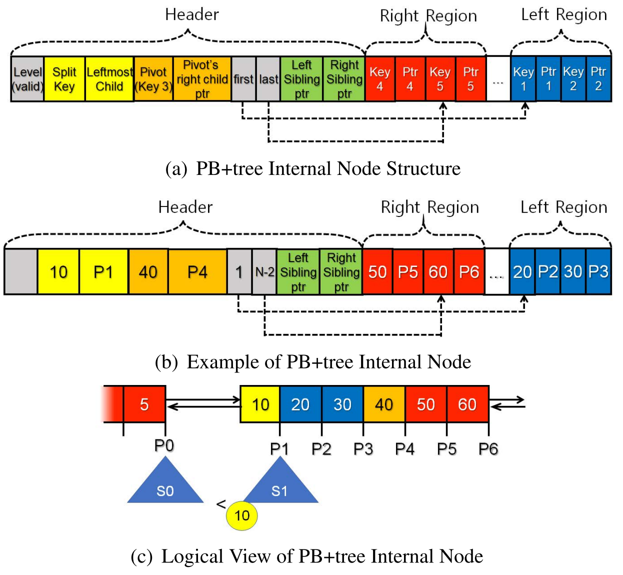


**FIGURE 2.** **Node Structure of Pivotal B+tree.**

**Algorithm 1** *insert*(*node*, *key*, *ptr*)

```
1:  if node.isNotFull() then
2:      if $key < node.pivot$ then
3:          – add new record into left region via RL shift
4:          $front \leftarrow node.hdr.front$;
5:          $node.record[\text{front-2}].ptr \leftarrow node.record[\text{front-1}].ptr$;
6:          for $i \leftarrow front$; $i < max\_offset$; $i++$ do
7:              if $key > node.record[i].key$ then
8:                  – shift records in reverse order
9:                  $node.record[i-1].key \leftarrow node.record[i].key$;
10:                 $node.record[i-1].ptr \leftarrow node.record[i].ptr$;
11:                 if $(\&node.record[i] - cacheline\_size)$
12:                   is at cacheline boundary then
13:                     $clwb(\&node.record[i] - cacheline\_size)$;
14:             else
15:                 $node.record[i-1].key \leftarrow key$;
16:                 $node.record[i-1].ptr \leftarrow ptr$;
17:                 $clwb(\&node.record[i-1])$;
18:                 return
19:         $node.record[max\_offset-1].key \leftarrow key$;
20:         $node.record[max\_offset-1].ptr \leftarrow ptr$;
21:         $clwb(\&node.record[max\_offset-1])$;
22:     else
23:         – add new record into right region via LR shift
24:         – omitted due to symmetry and lack of space
25: else
26:     $split(node, key, ptr)$;
```

pointer (right child of a split key), the *pivot key* that divides records into left and right regions, the pivot's right child pointer, the left sibling pointer, the right sibling pointer, and the *front* and *rear* offsets.

**split key**: In B+tree, the median key of an overfull node is pushed to its parent node when an overfull internal node splits. However, in PB+tree, the median key is not only pushed to its parent node but also stored as the *split key* in the new right sibling node. The split key is required for the failure-atomic in-place rebalancing (FAIR) algorithm. We note that the open source FAST and FAIR B+tree implementation does not store the median key in the right sibling node, leading to correctness issues with concurrent insertions. For example, if a query accesses an overfull node before we push the median key into the parent node, the query may visit an incorrect child node as it misses the median key.[1] To resolve this problem, PB+tree stores the median key as the split key in the right split node, and the split key is never updated until the node is deleted. The split key has only the right child pointer that points to the leftmost sub-tree of the node. This is because the left child of the split key is the rightmost child of its left sibling node.

**pivot key**: Another static element of the PB+tree node is the pivot key. When a right sibling node is created, PB+tree selects the median key of the right sibling node and stores it as the pivot key. With the pivot key, migrated records are separated into two sub-arrays and stored in different regions. I.e., larger keys than the pivot key are stored next to the header, i.e., in the region denoted as *right region* in Figure 2(a), and the keys smaller than the pivot key are stored at the end of the tree node, in the region denoted as *left region*. The rationale behind this design is to reduce the number of shift operations by half, i.e., we separate a large sorted array into two small sub-arrays.

[1] https://github.com/DICL/FAST_FAIR/pull/4

## A. PIVOTAL FAILURE-ATOMIC ShifT

The insertion and deletion algorithms of PB+tree employ the failure-atomic shift (FAST) algorithm of FAST and FAIR B+tree, i.e., keys and pointers are shifted in tandem. Algorithm 1 shows the insertion algorithm of PB+tree. If the insertion key is greater than the pivot key, we update the right region and shift elements greater than the insertion key from left to right. If the insertion key is smaller than the pivot key, we update the left region and shift elements smaller than the insertion key from right to left.

To delete an element from the sorted array, the direction of shift operations need to be reversed. Due to the symmetry and lack of space, we omit the detailed discussion of the deletion algorithm. The details of the FAST algorithm and how it tolerates transient inconsistency are referred to [16].

When inserting a new record, the right region grows from left to right but the left region, located at the end of the array, grows towards the beginning of the node. In between the left and right regions, there is a free space. Since the sizes of both regions change dynamically, the header stores an offset for the beginning of the left region (*front*) and another offset for the end of the right region (*rear*).

We note that there is a trade-off in storing offsets in a tree node. I.e., updating an offset and shifting existing records cannot be performed atomically. Therefore, FAST and FAIR B+tree uses sentinel pointers instead of offsets. However, if we use offsets, the implementation of FB+tree becomes greatly simplified as we can quickly access both ends of the sorted array. Besides, read transactions can access not only the right region but also the left region in ascending order, which helps improve the read performance as we describe in Section II-C.

Since updating an offset and shifting a sorted array cannot be performed atomically, as a compromise between consistency and efficiency, PB+tree uses the front and rear offsets as approximate hints. I.e., instead of accessing the entire array, PB+tree reads the array from the front or rear offset to find the actual boundary of the left or right region. As such, PB+tree does not call `clwb` for the front and rear offsets.

### B. LINEAR SCAN WITH PIVOT

When the size of an array is smaller than a few KBytes, linear scanning performs faster than binary search because the binary search often stalls due to poor cache locality and branch prediction failures while linear scanning benefits from hardware prefetchers and memory level parallelism [16]. Therefore, Pivotal B+tree also performs linear scanning as in FAST and FAIR B+tree. But, the search algorithm of PB+tree is slightly different from that of FAST and FAIR B+tree in two aspects. The first difference is that PB+tree compares a search key against the pivot key and accesses either the left or right region, but not both. The other difference is that PB+tree reads arrays always in ascending order because the performance of ascending memory access is faster than descending memory access as we discussed in Section II-C.

However, reading an array in ascending order is vulnerable to transient inconsistency issues if a concurrent write transaction is shifting the array in descending order. For example, suppose a read transaction is suspended after reading the first element but before reading the second element of the array that has 10, 20, and 30. While the read transaction is suspended, a delete transaction may shift 30 to the second position and 20 to the first position. When the read transaction wakes up, it will miss 20. FAST and FAIR B+tree resolves this inconsistency problem by employing a scan direction flag making read transactions read the node in a specified direction such that no key is missed, even if the read transaction reads the same key multiple times. PB+tree does not employ such a flag-based lock-free method because the algorithm that shifts a sorted array while it is being accessed by concurrent read transactions is so sophisticated that it often returns incorrect results and fails in our experience.

Instead, PB+tree proposes an optimistic search algorithm that corrects search paths in a lazy manner. I.e., regardless of the shift direction of concurrent write transactions, PB+tree reads tree nodes always in ascending order and improves the node access performance. However, since read transactions can miss some keys being shifted, PB+tree has to detect and correct incorrect search paths in a lazy manner. We will discuss the lazy correction method in Section III-D.

### C. NODE SPLIT, MERGE, AND REBALANCING

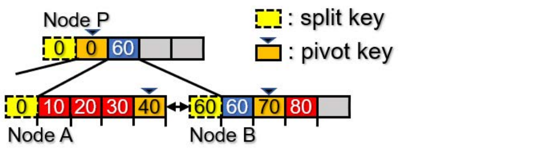
The rebalancing algorithm of PB+tree is shown in Algorithm 2. If the right region has more records than the left

**Algorithm 2** *split*$(node, key, ptr)$

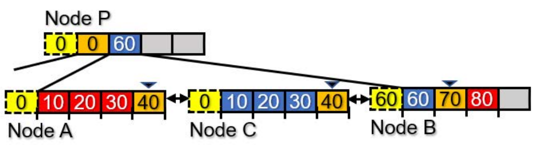
```
node.lock.acquire();
sibling ← nv_malloc(sizeof(node));
if node.front < max_offsets/2 then
    – more records are in the left region
    sibling.pivot ← getThirdQuarter(node);
    – create a copy node with a new pivot as a sibling
    sibling ← fullCopyWithNewPivot(node, sibling.pivot);
    sibling.left_sibling_ptr ← node;
    prev_rs ← node.right_sibling;
    sibling.right_sibling ← prev_rs;
    clwb(&sibling, sizeof(node));
    sibling.lock.acquire();
    – change the node's sibling pointer
    – this step will invalidate the overfull node
    node.right_sibling ← sibling;
    clwb(&node.right_sibling);
    prev_rs.left_sibling ← sibling;
    clwb(&prev_rs.left_sibling_ptr);
    – right sibling is a back-up node, so we can update the node.
    median ← getMedian(sibling);
    node.pivot ← getQuarter(sibling);
    node ← copyLowerHalfWithNewPivot(sibling, node.pivot);
    clwb(&node, sizeof(node));
    – validate the overfull node
    sibling.split_key ← sibling.record[median].key;
    clwb(&sibling.split_key);
    – put a sentinel to invalidate migrated keys in sibling
    sibling.record[median].ptr ← −1;
    clwb(&sibling.record[median]);
    sibling.lock.release();
    node.lock.release();
else
    – omitted due to symmetry
```

region, we create a right sibling node and migrate a half of the records of the overfull node from the right region. If the left region has more records, we migrate a half of the records from the left region. A key challenge of the node split algorithm in PB+tree is that we need to update the pivot key and rebalance the left and right regions accordingly. Otherwise, tree nodes will have skewed pivots and the size of regions will become unbalanced. Since selecting a new pivot and shifting records from one region to the other can not be done atomically, PB+tree creates a copy of the overfull node as its sibling node, which behaves as a back-up log node. But we note that the sibling node does not need to have the exactly same structure with the overfull node as long as its logical view is the same. I.e., as described in Algorithm 2, we create a new sibling node and redistribute all the records using a new pivot in the sibling node such that the sibling node eventually has the balanced left and right regions. Once we create a sibling node as a back-up node, we invalidate the overfull node and update it accordingly. Once the overfull node is invalidated, concurrent read transactions will ignore the overfull node and access the back-up sibling node instead. Hence, we can make changes to the overfull node in a non-failure-atomic manner.

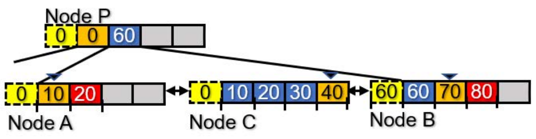
Figure 3 illustrates each step of the node split algorithm of PB+tree. Suppose a query tries to insert a key 50 into the tree shown in Figure 3(a). Since Node A is full, it has to split. In the first step, we create a right sibling back-up node. As in the FAIR algorithm [16], we consider the right sibling and the overfull node as a virtual single node. We note that the new



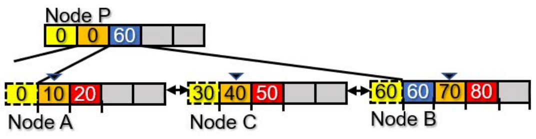
(a) Step 1. Node A overflows if new record is inserted



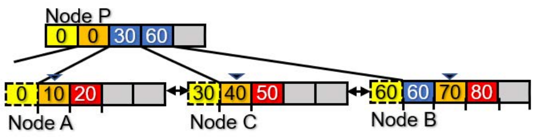
(b) Step 2. Create a back-up copy as a new right sibling node.



(c) Step 3. Delete migrated keys and change pivot key in Node A.



(d) Step 4. Update split key and put sentinel to create free space in Node C.



(e) Step 5. Insert Node C to parent Node P.

**FIGURE 3.** **Failure-Atomic In-place Rebalancing in PB+tree.**

sibling node is a copy of the overfull node but it has a different pivot key. The pivot key of the right sibling node should be the third quarter key of the overfull node since the smaller half of the sibling node will be deleted when the split completes. In the example shown in Figure 3(b), we select 40 as the pivot of the right sibling node. Although the pivot is different, its logical view is no different from the overfull Node A.

In the next step, as shown in Figure 3(c), we invalidate the overfull node and delete a larger half of the records. This step shifts records and changes the pivot key of the overfull node, which cannot be done in a failure-atomic way. However, such inconsistency is acceptable since we have invalidated the overfull node and concurrent read transactions visit its back-up sibling node. However, we note that the invalidation of the overfull node is not a flawless solution for concurrent read transactions because there exist various subtle corner cases in concurrent workloads. For example, a read transaction finds the overfull node is valid, but it goes sleep while the node is being split and the pivot is changed, then it wakes up and visits the incorrect child node. To address the consistency issues, the optimistic tree traversal algorithm, which we describe in Section III-D, becomes necessary.

In the next step shown in Figure 3(d), we update the split key of the right sibling node, which will behave as a commit mark of split operations. I.e., once we update the split key, the update of the right sibling node is complete and any query that searches for keys small than the split key will not scan the right sibling node. Then, we put a sentinel pointer and update the front or rear offset in the overfull node to delete migrated records.

In the final step shown in Figure 3(e), we add the split key of the right sibling node into its parent node or create a new root node using the split key. Then, the split is complete.

### *D. LOCK-FREE SEARCH WITH LAZY CORRECTION*

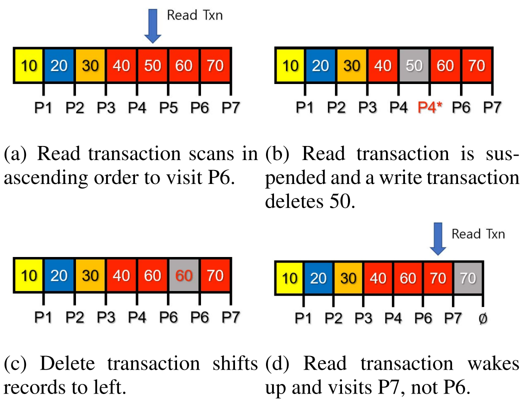
The failure-atomicity of data structures for byte-addressable persistent memory is difficult to achieve because of the instruction reordering and unexpected cacheline flushes. If lock-freedom is considered together, the challenge becomes even harder. Lee *et al.* [21] proposed to transform well-stabilized DRAM-based lock-free indexes into PM-based indexes using a systematic approach called *RECIPE*. However, RECIPE neglected that lock-free data structures for DRAM, such as Bw-tree [27], cannot be simply converted into persistent index unless CPUs support *memory persistency* [28], [29]. Without memory persistency, we need to call `clwb` and `mfence` for every memory access. So, the authors of RECIPE revised the paper in [30] and proposed to call `clwb` and `mfence` for every store and load instruction, as was proposed in *durable linearizability* [31]. Without a doubt, calling `clwb` and `mfence` for every store and load instruction causes unacceptably high overhead.

Alternatively, FAST and FAIR B+tree makes read transactions be aware of the order of write operations such that they can tolerate transient inconsistency. However, FAST and FAIR B+tree is very sensitive to the reordering of memory accesses, and we observe that its open source implementation often fails to return correct results. Another lock-free index, BzTree [32] employs Microsoft's *PMwCAS* library that atomically changes multiple 8-byte words in a lock-free manner. However, it has been shown that the overhead of *PMwCAS* is not negligible, i.e., *PMwCAS* library adds up to 12% higher overhead than hardware transactional memory [33].

The insertion and search algorithms for PB+tree make read transactions vulnerable to transient inconsistency caused by a concurrent write transaction. Consider an example shown in Figure 4. A read transaction looking for 65 is scanning a node in ascending order. After it reads key 50 but before reading key 60, it is suspended. In the next step, a write transaction deletes <50, P5> by shifting the array elements to left. When the read transaction wakes up, it reads the second to last element and finds the key is 70. Thus, it visits P7 instead of P6. To prevent this error, we may employ a node version counter and make all transactions double-check whether the version remains unchanged while queries scan the node.

**Algorithm 3** *LockFreeSearch*($node$, $key$)

```
1: while true do
2:     if $pivot.ptr == NULL$ then return $node.leftmost\_ptr$;
3:     if $key >= pivot.key$ then
4:         $i \leftarrow pivot\_idx + 1$
5:         while $record[i].ptr \neq NULL$ do
6:             if $key < record[i].key$ then
7:                 if $record[i].ptr \neq (t \leftarrow record[i-1].ptr)$ then
8:                     if $t == NULL$ then
9:                         break;
10:                    else return $t$;
11:            $i++$
12:        if $(t \leftarrow node.right\_sibling) \neq NULL$ then
13:            if $key >= sibling.split\_key$ then return $t$;
14:        $t \leftarrow record[i-1].ptr$;
15:        if $t == NULL$ then
16:            continue;
17:        else return $t$;
18:    else
19:        if $key < record[node.front].key$ then
20:            if $(t \leftarrow node.left\_sibling) \neq NULL$ then
21:                if $key < node.split\_key$ then return $t$;
22:            $t \leftarrow node.leftmost\_ptr$;
23:            if $t == NULL$ then
24:                continue;
25:            else return $t$;
26:        for $i \leftarrow node.front + 1$; $i < pivot\_idx$; $i++$ do
27:            – omitted due to symmetry and lack of space
```



(a) Read transaction scans in ascending order to visit P6.

(b) Read transaction is suspended and a write transaction deletes 50.

(c) Delete transaction shifts records to left.

(d) Read transaction wakes up and visits P7, not P6.

**FIGURE 4.** Scan in ascending order leads to inconsistency.

We note that this is a common strategy used for various lock-free data structures [34]–[37]. Although such version-based lock-free tree traversal algorithms are theoretically sound and complete, it requires to scan the same tree node repeatedly.

The complexity of lock-free algorithms leads to various subtle implementation bugs. Besides, we note that bugs can be introduced by compile optimizations. Memory access reordering can be made not only by CPUs during run-time but also by a compiler at compile time. Although the cardinal rule of memory reordering followed by compiler developers is to keep the behavior of a single-threaded program, there have been several memory reordering bugs in `gcc`.[2] As such, FAST and FAIR B+tree disabled compiler optimization options when they evaluate the performance of multi-threaded indexing.

[2]gcc 4.9.2 has a bug (GCC Bugzilla – Bug 59448) in `memory_order_consume`.

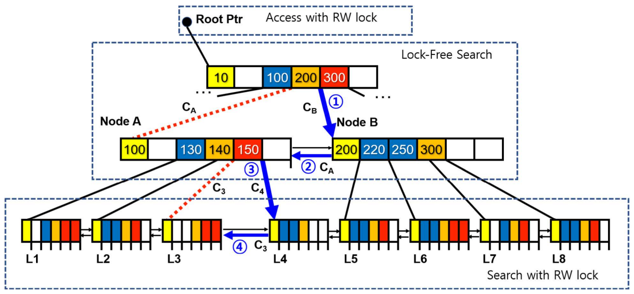


**FIGURE 5.** Optimistic Lock-Free Search with Lazy Correction.

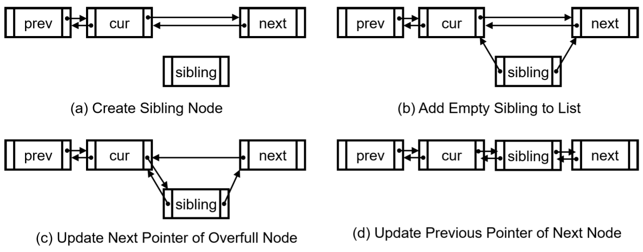

prev
cur
next
sibling
(a) Create Sibling Node
(b) Add Empty Sibling to List
(c) Update Next Pointer of Overfull Node
(d) Update Previous Pointer of Next Node

**FIGURE 6.** Failure-Atomic Updates to Doubly Linked List.

### 1) OPTIMISTIC LOCK-FREE SEARCH AND LAZY CORRECTION

To mitigate the memory access reordering problem and remove burdens from programmers, we propose an optimistic tree traversal method that corrects incorrect node visits using doubly linked lists.

In PB+tree, we use read/write locks for the root node pointer. The read/write lock for the root node pointer is required since the root node may split and the global root pointer can be changed. Once we read the root node, the read lock is immediately released so that subsequent transactions can access the index.

We also use read/write locks for leaf nodes to prevent the well-known *phantom reads* and *dirty reads* such that even if a write transaction inserts multiple data into multiple leaf nodes, concurrent read transactions can find all the inserted data or none of them. I.e., we serialize concurrent transactions in leaf level, which is sufficient because write transactions commit only in leaf nodes.

For internal tree nodes, we enable optimistic lock-free search, which is vulnerable to transient inconsistency problems as we described earlier. However, an incorrect internal search path does not mean incorrect query results as long as the multiple paths guide to the correct leaf node.

Consider an example shown in Figure 5. Suppose a read transaction is looking for a key 145. It has to visit Node A, but the query may visit Node B because of the conflict of concurrent threads, implementation bugs, or compiler bugs. This will result in incorrect search results in traditional B+tree. However, PB+tree will detect the search key is smaller than the split key of Node B, and the read transaction will be steered to visit its left sibling node for the correct search path. If the search key is greater than the largest key of the node, the read transaction will compare the search key against the split key of its right sibling node to correct the search path. Similarly, even if the query visits an incorrect leaf node again, L4 in the example, it will follow the left sibling pointer and access the correct leaf node.

If a lot of concurrent transactions conflict, our optimistic lock-free search algorithm may visit incorrect child nodes many times. But, even if a query ends up arriving at a leaf node very far from the correct leaf node, it is guaranteed that the query will eventually visit the correct leaf node. The optimistic lock-free search algorithm is shown in Algorithm 3.

### E. FAILURE-ATOMIC DOUBLY LINKED LIST

To enable the optimistic lock-free search, tree nodes in the same level need to be managed as a doubly linked list. Adding or removing a node in a doubly linked list in a failure-atomic manner is trivial. As shown in Figure 6 (a) and (b), we make a new node point to its left and right sibling. Since the new node is not accessible from the list, the new node will be ignored if a system crashes. Therefore, it is failure-atomic. In the next steps, shown in Figure 6 (c) and (d), we change the next pointer of the left sibling and the previous pointer of the right sibling one by one. Updating these two pointers is not atomic. However, even if a system crashes after we update only one pointer, a recovery process can easily detect and correct it. We note that we may use the `PMwCAS` library to update both pointers atomically. With the `PMwCAS` library, the recovery process will not be even necessary.

## IV. EXPERIMENTS

We evaluate the performance of PB+tree against the state-of-the-art fully persistent B+trees - wB+-tree with slot+bitmap nodes [19], FAST and FAIR B+tree [16], and BzTree [32]. All four implementations use Intel's PMDK library and we compiled our implementations using g++ 7.4 with -O3 option unless stated otherwise. For failure-atomicity, we carefully enforce the ordering of memory accesses by calling PMDK APIs for memory barrier and cacheline flush instructions.

We run experiments on a workstation that has two Intel Xeon Gold 5215 processors (2.5 GHz, 20 vCPUs with hyper-threading enabled, and 13.75 MB L3 cache) that guarantee total store ordering (TSO). The workstation has 64 GB of DDR4 DRAM and four 128 GB Optane DCPMM modules evenly distributed across NUMA nodes. The two DCPMM modules per NUMA node are configured in app-direct interleaved mode and managed as a single region (i.e., `pmem0` on NUMA node 0 and `pmem1` on NUMA node 1). In our experiments, we bind threads to NUMA nodes such that all threads access the local PM region and avoid NUMA effect.

For each index, we allocate a directory-based poolset in a single PM region. The directory-based poolset allows its size to grow or shrink dynamically, which is suitable for dynamic index. Although we do not present the performance of a single static pool due to the lack of space, we find the performance difference between a single pool and directory-based poolset is negligible, i.e., less than 2% overhead is incurred by directory-based poolset in our experiments.

Since all tree nodes are stored within a directory-based poolset, PB+tree can make use of atomic 8-byte store instructions. That is, even if we call `pmemobj_alloc()` for each tree node, which returns a fixed 8-byte poolset ID and an 8-byte offset, only the 8-byte offset is used as a node pointer in our implementation.

### A. INSERTION PERFORMANCE

In the first set of experiments, we measure the insertion and deletion performance of persistent indexes. For the workloads, we make use of three synthetically generated distributions of 8-byte keys and 8-byte values. While B+tree is known to be generally insensitive to key distributions, the FAST algorithm is sensitive to sequential insertions and deletions due to the number of shift operations and the direction of memory accesses.

If we insert monotonically increasing keys, the FAST algorithm does not shift any record but appends the largest key to the rightmost leaf node. Unless a node splits, FAST and FAIR B+tree and PB+tree flush only one cacheline, which is optimal, while wB+tree flushes four cachelines. For this workload, PB+tree does not use the left region but only the right region. As such, PB+tree behaves similarly to FAST and FAIR B+tree and shows similar insertion performance. Due to the minimal number of cacheline flushes, the insertion throughputs of all persistent B+trees are insensitive to the index size as shown in Figure 7(a). For workloads with increasing keys, a larger tree node size, i.e., 1024-byte node size (denoted as `PB(1024)` and `FF(1024)`), reduces the number of splits and results in higher throughputs than 512-byte node size.

Figure 7(b) shows the throughputs when we insert records in descending order. This workload is the worst case for FAST and FAIR B+tree because FAST and FAIR B+tree has to shift all existing array elements from left to right so that it can make a free slot at the beginning of the array. As a result, FAST and FAIR B+tree is even outperformed by wB+tree, which always appends a new record to the end of an unsorted array regardless of whether keys are increasing or decreasing. Unlike FAST and FAIR B+tree, PB+tree does not perform any shift operation for descending key workload because descending keys are appended to the front of the left region. For example, if we insert key 15 to the PB+tree node shown in Figure 2(b), it will be stored in the third to last position without shifting any existing keys. As such, the insertion throughput of descending key workload is similar to that of ascending key workload in PB+tree. The small performance difference between ascending key and descending key workloads for PB+tree is because ascending key workload benefits from hardware prefetchers that prefetches the right

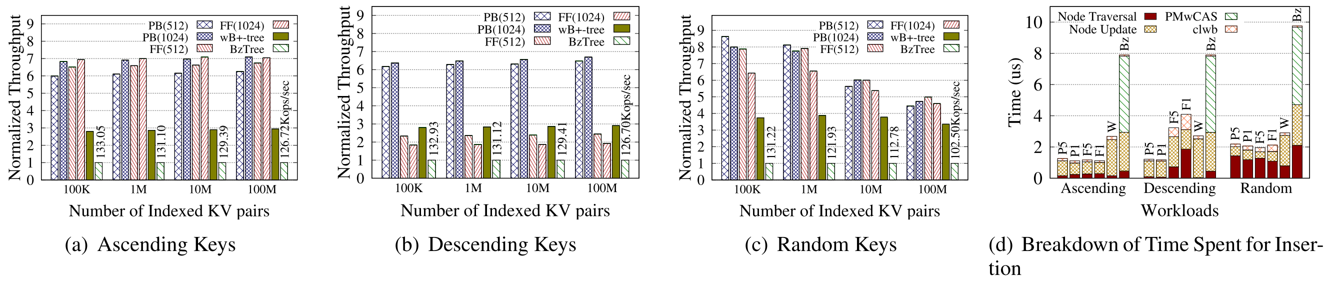

(a) Ascending Keys (b) Descending Keys (c) Random Keys (d) Breakdown of Time Spent for Insertion

**FIGURE 7.** **Insertion Performance:** ***P5: PB(512), P1: PB(1024), F5: FF(512), F1: FF(1024), H5: HB(512), H1: HB(1024), W: wB+tree.***

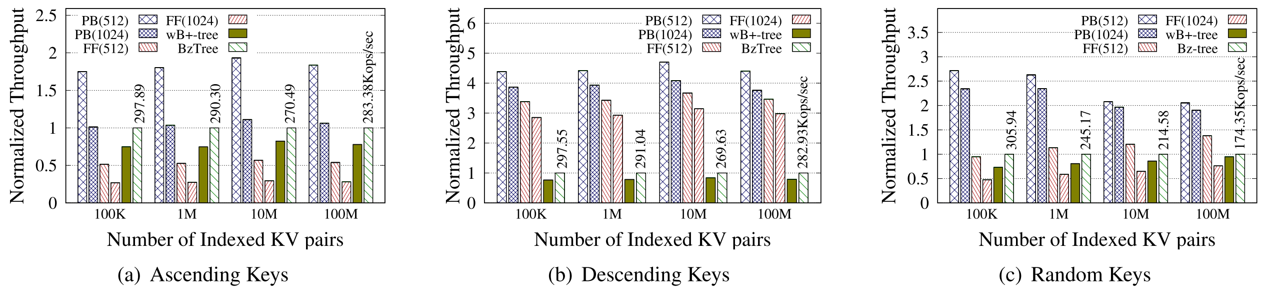

(a) Ascending Keys (b) Descending Keys (c) Random Keys

**FIGURE 8.** **Deletion Performance.**

region while reading the header, but descending key workload does not.

Figure 7(c) shows the throughput for random keys. We note that PB+tree insertion performs worst when we keep inserting keys close to pivot keys. Therefore, PB+tree performs worse on the workloads with random key insertions than the workloads with increasing or decreasing key insertions. In random key workloads, the insertion throughput degrades as we increase the tree node size. This is because larger tree node sizes require a larger number of records to be shifted. When we insert 10 million records, the insertion throughput of FAST and FAIR B+tree with 1 KByte nodes is 10.5% lower than when the node size is 512 Bytes. PB+tree also suffers from larger tree node sizes when the number of indexed records is smaller than 1 million. However, PB+tree suffers less than FAST and FAIR B+tree because of the regions separated by the pivot key. In particular, when we insert more than 10 million records, `PB(1024)` outperforms `PB(512)` because of less frequent node splits.

In the experiments shown in Figure 7(d), we insert 100 million records in batches and breakdown the insertion time into (1) Node Update time, (2) Node Traversal time, and (3) cacheline flush time.

BzTree shows the worst performance due to the high overhead of `PMwCAS`, which is a library that allows applications to update multiple 8-byte words in a failure-atomic and lock-free manner [33]. Besides this, BzTree suffers from expensive CoW operations; BzTree replaces an existing internal node via CoW when a new child node is inserted.

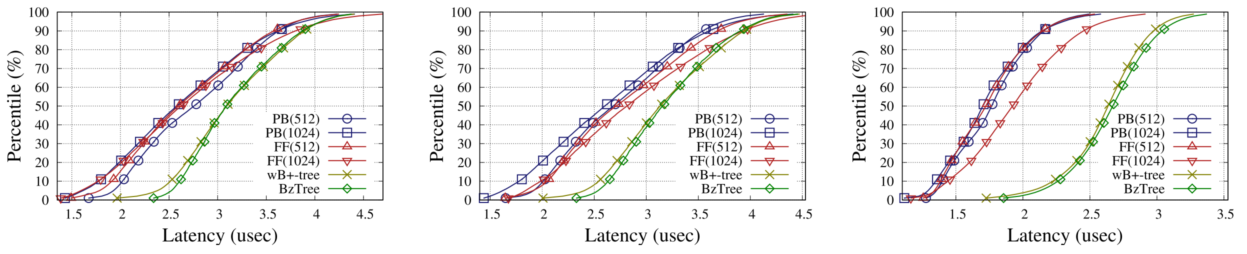
wB+tree suffers from expensive CoW splits, which results in longer Node Update times. Besides this, wB+tree suffers from a level of indirection, which makes search operations jump around cachelines. The cacheline flush time of wB+tree is also about twice higher than the other persistent indexes because of the metadata (slot array and bitmap) updates.

For the workload with descending keys, FAST and FAIR B+tree suffers from a large number of shift operations and a large number of `clwb` calls. Note that the `clwb` overhead increases as the node size increases. Interestingly, the node traversal time of FAST and FAIR B+tree is much higher than the other two indexes. This is because `clwb` instruction in our Intel Xeon Gold 5215 processor is just an alias of `clflushopt`, which evicts dirty cachelines from CPU caches. Therefore, the number of cache misses that occur while traversing the tree structure is much higher than the other indexes that call `clwb` less frequently.

With the workload with random keys, node splitting occurs frequently and the node utilization decreases down to 66% on average. With such a low node utilization and a large number of cacheline flushes that evict cachelines from CPU caches, the node traversal time and the insertion time become higher than the other two workloads.

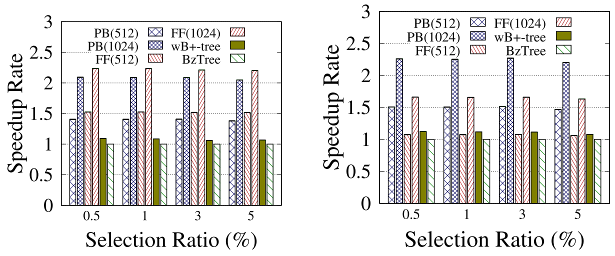
## B. DELETION PERFORMANCE

Figure 8 shows the deletion performance. Again, FAST and FAIR B+tree is sensitive to the order in which keys are deleted. If keys are deleted in descending order, FAST and FAIR B+tree does not shift any record, but if keys are deleted in ascending order, it has to shift all records. Unlike FAST and FAIR B+tree, PB+tree does not shift any record regardless of whether keys are deleted in ascending or descending



(a) 10,000 reads after 100 million insertions (b) 10,000 reads after 100 million insertions and 10 million deletions (c) 100 million reads after 100 million insertions

**FIGURE 9.** Latency CDF for Point Queries.



(a) Range Query without Deletions (b) Range Query with Deletions

**FIGURE 10.** Range Query Throughput Normalized to wB+tree.

order. As a result, the deletion performance of PB+tree is 3x higher than FAST and FAIR B+tree if records are deleted in ascending order. Even for random key deletions, PB+tree shows superior throughput to FAST and FAIR B+tree. This is because PB+tree allows an underfull node to borrow records from its right sibling node whereas FAST and FAIR B+tree does not. In FAST and FAIR B+tree, if an underfull node borrows a record from its right sibling node, the sibling node has to shift all the records to left. However, in PB+tree, the redistribution between sibling nodes is rather easy to implement because a borrowed record can be removed from the sibling node and appended to underfull node without any shift operation.

We note that the deletion throughput of BzTree is much higher than that of wB+tree, which is different from the results presented in Figure 7. This is because BzTree calls `PMwCAS` twice for insertions, but once for deletions because the invalidation of a single metadata is sufficient for a delete operation.

## C. SEARCH PERFORMANCE

Figure 9 shows the search performance of persistent indexes. For the experiments shown in Figure 9(a), we insert 100 million records, clear CPU caches, and submit 10,000 point queries. Since this workload does not delete any record, the scan direction of FAST and FAIR B+tree nodes is not reversed, which is the optimal case for FAST and FAIR B+tree. As a result, PB+tree and FAST and FAIR B+tree show similar search performance. It is noteworthy that the use of 1 KB tree node size is beneficial to the point queries that have low latency. This is because of the CPU cache effect. I.e., the queries that read tree nodes from CPU caches have low latency and a larger tree node size benefits from more cached data. However, a larger tree node size suffers from a higher cache miss overhead. Hence, FAST and FAIR B+tree with 1 KB node size (denoted as FF(1024)) have higher tail latency. Since PB+tree reduces the number of cacheline accesses using the pivot key, the tail latency of PB+tree with 1 KB node size is shorter than FF(1024).

In the experiments shown in Figure 9(b), we insert 100 million records, delete 10% of them, i.e., 10 million records, clear CPU caches, and submit 10,000 point queries. Due to the deletions, about 10% of leaf nodes have their scan directions reversed in FAST and FAIR B+trees. The reversed scan direction negatively affects the search performance of FAST and FAIR B+trees. With a larger number of deletions, the performance gap between two indexes widens.

In the experiments shown in Figure 9(c), we insert 100 million records, clear CPU caches, and submit 100 million queries. Unlike the previous two workloads, most of the queries in this workload benefit from CPU cache hits due to a large number of read-only batch queries. Therefore, the tail latencies are lower than 2.8 usec while those of previous workloads are as high as 4.8 usec. Although this workload does not reverse the scan direction of FAST and FAIR B+tree, PB+tree outperforms FAST and FAIR B+tree because PB+tree reduces the number of comparisons by half using the pivot key. wB+tree and BzTree exhibit longer latencies than other indexes because they employ a level of indirection and store key-values in append-only manner. As a result, they fail to leverage hardware prefetchers and memory level parallelism.

Figure 10 shows the range query performance while we vary the selection ratio, which is the percentage of records selected out of 100 million indexed records. As we increase the node size, the range query performance improves, i.e., 1 KByte node size shows about 1.5x higher throughput compared to 512 Byte node size. Without deletions, FAST and FAIR B+tree slightly outperforms PB+tree because it accesses cachelines in ascending order. But if 10% of records

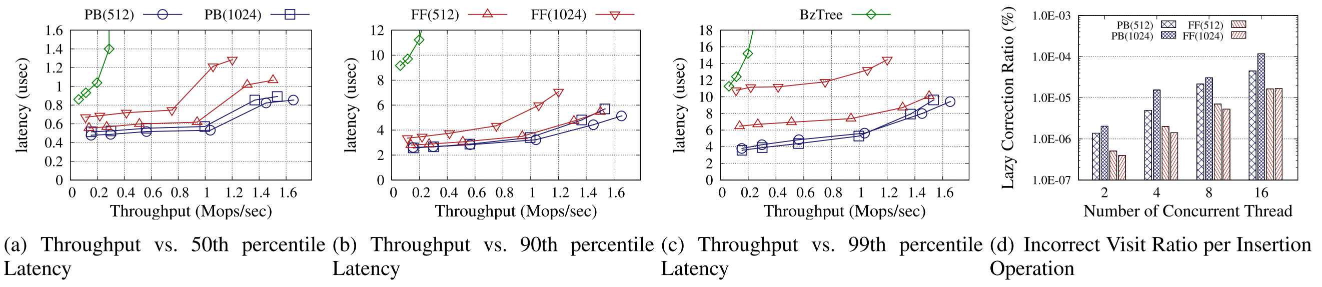

(a) Throughput vs. 50th percentile Latency (b) Throughput vs. 90th percentile Latency (c) Throughput vs. 99th percentile Latency (d) Incorrect Visit Ratio per Insertion Operation

**FIGURE 11.** **Throughput of Concurrent Mixed Workloads.**

have been deleted, read transactions scan some of FAST and FAIR B+tree nodes in the reverse direction. As such, FAST and FAIR B+tree is outperformed by PB+tree.

### D. MULTI-THREADED PERFORMANCE

In the experiments shown in Figure 11, we evaluate the throughput and latency of multi-threaded FAST and FAIR B+tree, BzTree, and PB+tree. We vary the number of threads from 1 to 20 as we run experiments on a single NUMA socket to avoid the NUMA effect. With two NUMA nodes, we observed that all indexing structures suffer from the NUMA effect and their overall throughputs degrade. We plan to investigate the NUMA effect of persistent indexing structures as our future work.

For the experiments, we populated each persistent index with 100 million records. Then, we run 100 million concurrent transactions mixed with writes and reads, i.e., 19% inserts, 5% deletes, and 76% point queries. We compiled both FAST and FAIR B+tree and PB+tree implementations with the compiler optimization disabled because FAST and FAIR B+tree returns incorrect results with concurrent workloads if we turn on optimization flags. We note that, even if we use the compile optimizations, PB+tree returns correct results while being much faster. We use `numactl` to bind all threads to a single socket since the poolset is allocated in a single DCPMM device.

The results show that PB+tree with 512 Bytes shows the highest throughput and lowest latency. This is because we use read/write locks for leaf nodes and the small leaf nodes benefit from the fine-grained locks. Overall, PB+tree shows a higher concurrency level than FAST and FAIR B+tree since the optimistic lock-free search algorithm of PB+tree ignores transient inconsistency and does not rescan tree nodes. Figure 11(d) shows the probability of such incorrect node visits. As we increase the number of concurrent threads, the probability increases, but from a very small probability, i.e., only 0.00012% with 16 threads. FAST and FAIR B+tree also allows read transactions to visit incorrect child nodes due to its FAIR algorithm, which steers queries to right sibling nodes. However, the probability of FAST and FAIR B+tree visiting incorrect nodes is up to 11x less than that of PB+tree at the cost of rescanning the same tree nodes.

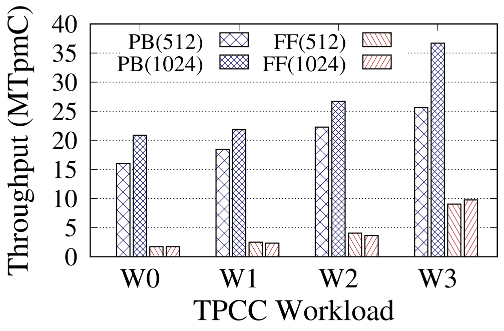


**FIGURE 12.** **Performance with TPCC Benchmarks:** ***W0(StockLevel: 4%, Delivery: 4%, OrderStatus: 4%, Payment: 43%, NewOrder: 45%), W1(4%, 4%, 14%, 43%, 35%), W2(4%, 4%, 24%, 43%, 25%), W3(4%, 4%, 34%, 43%, 15%).***

### E. TPC-C RESULTS

In the final set of experiments shown in Figure 12, we evaluate the performance using TPC-C benchmark. For the experiments, we run 16 threads that submit 10 million transactions. TPC-C benchmark consists of 5 types of transactions; *StockLevel*, *Delivery*, *OrderStatus*, *Payment*, and *NewOrder*. We generated four workloads with varying the percentage of these queries so that the proportion of search queries increases from workload W0 to W3. Unfortunately, even if we disable the compile optimization flags, FAST and FAIR B+tree fails to run concurrent TPC-C benchmarks. One of the reasons for the incorrect results is that FAST and FAIR B+tree does not store the split key in sibling nodes. To fix the concurrency issues, we had to implement the crabbing protocol [38] in FAST and FAIR B+tree for TPC-C benchmark. Due to the lock-contention coming from the crabbing protocol, FAST and FAIR B+tree shows 3.7-11.5x lower query processing throughput (TpmC) than PB+tree.

## V. CONCLUSION

In this work, we designed and implemented *Pivotal* B+tree (PB+tree) for byte-addressable persistent memory. PB+tree employs the pivot key, which helps reduce the number of expensive failure-atomic shift operations and makes PB+tree insensitive to key access patterns. PB+tree also reduces the complexity of lock-free tree traversal algorithm by proposing the optimistic traversal with the lazy correction method. Our performance study shows PB+tree outperforms FAST and FAIR B+tree on various workloads.

## REFERENCES

[1] Intel. *Intel and Micron Produce Breakthrough Memory Technology*. [Online]. Available: https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology

[2] D. Bittman, P. Alvaro, and E. L. Miller, "A persistent problem: Managing pointers in NVM," in *Proc. 10th Workshop Program. Lang. Operating Syst. (PLOS)*, 2019, pp. 30–37.

[3] R. Fang, H.-I. Hsiao, B. He, C. Mohan, and Y. Wang, "High performance database logging using storage class memory," in *Proc. IEEE 27th Int. Conf. Data Eng.*, Apr. 2011, pp. 1221–1231.

[4] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via JUSTDO logging," in *Proc. 21st Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Mar. 2016, pp. 427–442.

[5] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "High-performance transactions for persistent memories," in *Proc. 21st Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Mar. 2016, pp. 399–411.

[6] E. Lee, H. Bahn, and S. H. Noh, "Unioning of the buffer cache and journaling layers with non-volatile memory," in *Proc. 11th USENIX Conf. File Storage Technol. (FAST)*, 2013, pp. 73–80.

[7] G. Oh, S. Kim, S.-W. Lee, and B. Moon, "SQLite optimization with phase change memory for mobile applications," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1454–1465, Aug. 2015.

[8] J. Ou, J. Shu, and Y. Lu, "A high performance file system for non-volatile main memory," in *Proc. 11th Eur. Conf. Comput. Syst.*, Apr. 2016, pp. 1–6.

[9] A. Rudoff, "Programming models for emerging non-volatile memory technologies," *Login*, vol. 38, no. 3, pp. 40–45, Jun. 2013.

[10] P. Sehgal, S. Basu, K. Srinivasan, and K. Voruganti, "An empirical study of file systems on NVM," in *Proc. 31st Symp. Mass Storage Syst. Technol. (MSST)*, May 2015, pp. 1–14.

[11] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger, "Metadata efficiency in versioning file systems," in *Proc. 2nd USENIX Conf. File Storage Technol. (FAST)*, 2003, pp. 43–58.

[12] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *Proc. 9th USENIX Conf. File Storage Technol. (FAST)*, 2011, pp. 1–15.

[13] J. Xu and S. Swanson, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," in *Proc. 14th USENIX Conf. File Storage Technol. (FAST)*, 2016, pp. 323–338.

[14] Y. Zhang and S. Swanson, "A study of application performance with non-volatile main memory," in *Proc. 31st Symp. Mass Storage Syst. Technol. (MSST)*, May 2015, pp. 1–10.

[15] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2013, pp. 421–432.

[16] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in byte-addressable persistent $B^+$-tree," in *Proc. 11th USENIX Conf. File Storage (FAST)*, 2018, pp. 187–200.

[17] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 2016, pp. 371–386.

[18] J. Yang, Q. Wei, C. Chen, C. Wang, and K. L. Yong, "NV-Tree: Reducing consistency cost for NVM-based single level systems," in *Proc. 13th USENIX Conf. File Storage Technol. (FAST)*, 2015, pp. 167–181.

[19] S. Chen and Q. Jin, "Persistent $B^+$-trees in non-volatile main memory," *Proc. VLDB Endowment (PVLDB)*, vol. 8, no. 7, pp. 786–797, 2015.

[20] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, "WORT: Write optimal radix tree for persistent memory storage systems," in *Proc. 15th USENIX Conf. File Storage Technol. (FAST)*, 2017, pp. 257–270.

[21] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, "RECIPE: Converting concurrent DRAM indexes to persistent-memory indexes," in *Proc. 27th ACM Symp. Operating Syst. Princ. (SOSP)*, 2019, pp. 462–477.

[22] X. Zhou, L. Shou, K. Chen, W. Hu, and G. Chen, "DPTree: Differential indexing for persistent memory," *Proc. VLDB Endowment*, vol. 13, no. 4, pp. 421–434, Dec. 2019.

[23] Y. Chen, Y. Lu, K. Fang, Q. Wang, and J. Shu, "$\mu$Tree: A persistent $B^+$-tree with low tail latency," *Proc. VLDB Endowment*, vol. 13, no. 12, pp. 2634–2648, Jul. 2020.

[24] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *Proc. 18th USENIX Conf. File Storage Technol. (FAST)*, 2020, pp. 169–182.

[25] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *Proc. IEEE 13th Int. Symp. High Perform. Comput. Archit.*, Feb. 2007, pp. 63–74.

[26] J. Seo, W.-H. Kim, W. Baek, B. Nam, and S. H. Noh, "Failure-atomic slotted paging for persistent memory," in *Proc. 22nd Int. Conf. Architectural Support for Program. Lang. Operating Syst.*, Apr. 2017, pp. 91–104.

[27] J. J. Levandoski, D. B. Lomet, and S. Sengupta, "The Bw-tree: A B-tree for new hardware platforms," in *Proc. IEEE 29th Int. Conf. Data Eng. (ICDE)*, Apr. 2013, pp. 302–313.

[28] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Princ. (SOSP)*, 2009, pp. 133–146.

[29] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *Proc. ACM/IEEE 41st Int. Symp. Comput. Archit. (ISCA)*, Jun. 2014, pp. 265–276.

[30] S. Kwon Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, "RECIPE : Converting concurrent DRAM indexes to persistent-memory indexes," 2019, *arXiv:1909.13670*.

[31] J. Izraelevitz, H. Mendes, and M. L. Scott, "Linearizability of persistent memory objects under a full-system-crash failure model," in *Proc. 30th Int. Symp. Distrib. Comput. (DISC)*, 2016, pp. 313–327.

[32] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson, "Bztree: A high-performance latch-free range index for non-volatile memory," *Proc. VLDB Endowment*, vol. 11, no. 5, pp. 553–565, 2018.

[33] T. Wang, J. Levandoski, and P.-A. Larson, "Easy lock-free indexing in non-volatile memory," in *Proc. IEEE 34th Int. Conf. Data Eng. (ICDE)*, Apr. 2018, pp. 461–472.

[34] A. Braginsky and E. Petrank, "A lock-free $B^+$tree," in *Proc. 24th ACM Symp. Parallelism Algorithms Archit. (SPAA)*, 2012, pp. 58–67.

[35] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel, "Non-blocking binary search trees," in *Proc. 29th ACM SIGACT-SIGOPS Symp. Princ. Distrib. Comput. (PODC)*, 2010, pp. 131–140.

[36] M. Fomitchev and E. Ruppert, "Lock-free linked lists and skip lists," in *Proc. 23rd Annu. ACM Symp. Princ. Distrib. Comput. (PODC)*, 2004, pp. 50–59.

[37] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proc. 14th Annu. ACM Symp. Parallel algorithms Archit. (SPAA)*, 2002, pp. 73–82.

[38] A. Silberschatz, H. Korth, and S. Sudarshan, *Database Systems Concepts*. New York, NY, USA: McGraw-Hill, 2005.

**JONGHYEON YOO** received the B.S. and M.S. degrees from SungKyunKwan University, in 2020 and 2021, respectively. He is currently a Software Engineer with Kakao Corporation. His research interests include persistent memory, system software, data analytics, and machine learning.

**HOKEUN CHA** received the B.S. and M.S. degrees from SungKyunKwan University, in 2018 and 2020, respectively. He is currently pursuing the Ph.D. degree in computer science with the University of Wisconsin–Madison. His research interests include database systems and storage systems.

**WONBAE KIM** received the B.S. degree in computer science and engineering from the Ulsan National Institute of Science and Technology (UNIST), South Korea, in 2015, where he is currently pursuing the Ph.D. degree with the School of Electrical and Computer Engineering. His research interests include big data processing systems, machine learning platforms, key-value stores, and persistent memory.

**WOOK-HEE KIM** received the B.S. and Ph.D. degrees from the Ulsan National Institute of Science and Technology, in 2013 and 2019, respectively. He was a Postdoctoral Associate at Virginia Tech and SungKyunKwan University. He is an Assistant Professor with the Department of Computer Science and Engineering, Konkuk University. His research interests include database systems, storage systems, and systems software for persistent memory.

**SUNG-SOON PARK** received the B.S. degree from Hongik University, in 1984, the M.S. degree from Seoul National University, in 1987, and the Ph.D. degree from Korea University, in 1994. He was a Postdoctoral Researcher at Northwestern University, from 1996 to 1998, and a Visiting Researcher at the IBM T.J Watson Research Center, NY, USA, in 1999. He is a Professor with the Department of Computer Science and Engineering, Anyang University, and the Founder, the President, and the CEO of Gluesys Company Ltd.—which is a leading NAS solution company in South Korea.

**BEOMSEOK NAM** (Member, IEEE) received the B.S. and M.S. degrees from Seoul National University, South Korea, and the Ph.D. degree in computer science from the University of Maryland, College Park, Maryland, in 2007. He was an Assistant/Associate Professor at the Ulsan National Institute of Science and Technology (UNIST), South Korea. He is an Associate Professor with SungKyunKwan University, South Korea. His research interests include data-intensive computing, database systems, and embedded system software.