B^3-Tree: Byte-Addressable Binary B-Tree for Persistent Memory

HOKEUN CHA, Sungkyunkwan University
MOOHYEON NAM and KIBEOM JIN, Ulsan National Institute of Science and Technology (UNIST)
JIWON SEO, Hanyang University
BEOMSEOK NAM, Sungkyunkwan University

17

In this work, we propose *B^3-tree*, a hybrid index for persistent memory that leverages the byte-addressability of the in-memory index and the page locality of B-trees. As in the byte-addressable in-memory index, B^3-tree is updated by 8-byte store instructions. Also, as in disk-based index, B^3-tree is failure-atomic since it makes every 8-byte store instruction transform a consistent index into another consistent index without the help of expensive logging. Since expensive logging becomes unnecessary, the number of cacheline flush instructions required for B^3-tree is significantly reduced. Our performance study shows that B^3-tree outperforms other state-of-the-art persistent indexes in terms of insert and delete performance. While B^3-tree shows slightly worse performance for point query performance, the range query performance of B^3-tree is 2x faster than FAST and FAIR B-tree because the leaf page size of B^3-tree can be set to 8x larger than that of FAST and FAIR B-tree without degrading insertion performance. We also show that read transactions can access B^3-tree without acquiring a shared lock because B^3-tree remains always consistent while a sequence of 8-byte write operations are making changes to it. As a result, B^3-tree provides high concurrency level comparable to FAST and FAIR B-tree.

CCS Concepts: • **Information systems → Data structures**; **Storage class memory**;

Additional Key Words and Phrases: Non-volatile memory, data structure, persistent indexing

ACM Reference format:
Hokeun Cha, Moohyeon Nam, Kibeom Jin, Jiwon Seo, and Beomseok Nam. 2020. B^3-Tree: Byte-Addressable Binary B-Tree for Persistent Memory. *ACM Trans. Storage* 16, 3, Article 17 (July 2020), 27 pages.
https://doi.org/10.1145/3394025

This research was supported by National Research Foundation of Korea (No. 2018R1A2B3006681, Improving Data Processing Performance with Byte-Addressable Non-Volatile Memory) and National Research Foundation of Korea (No. 2016M3C4A7952587, PF Class Heterogeneous High Performance Computer Development), and Institute for Information & Communications Technology Promotion (IITP) (grant No. 2018-0-00549, Extremely Scalable Order-preserving Operating System for Manycore and Non-volatile Memory) funded by Ministry of Science and ICT, Korea, and NST (B551179-12-04-00, ETRI R&D grant 19ZS1220).

Authors' addresses: H. Cha, College of Information and Communication Engineering, Sungkyunkwan University, 2066 Jangan-gu, Suwon-si, Gyeonggi-do, South Korea; email: chahg0129@skku.edu; M. Nam and K. Jin, Ulsan National Institute of Science and Technology (UNIST), 50 UNIST-gil, Eonyang-eup, Ulju-gun, Ulsan Metropolitan City, South Korea; email: moohyeon.nam@gmail.com; J. Seo, Hanyang University, 222 Wangsim-ri, Seongdong-gu, Seoul, South Korea; email: seojiwon@hanyang.ac.kr; B. Nam (corresponding author), College of Computing, Sungkyunkwan University, 2066 Jangan-gu, Suwon-si, Gyeonggi-do, South Korea; email: bnam@skku.edu.

1553-3077/2020/07-ART17 $15.00
https://doi.org/10.1145/3394025

1 INTRODUCTION

Recent advances of byte-addressable persistent memories (PM), such as 3D Xpoint [12], phase-change memory [41], and STT-MRAM [9] are expected to open up new opportunities to transform main memory from volatile device to persistent storage. Due to its persistency and byte-addressability, PM can be used either as fast secondary storage via legacy block I/O interfaces or as slow but large main memory [3, 6, 7, 10, 15, 19, 27–29, 43]. In order to leverage high performance, persistence, and byte-addressability of PM, various opportunities are being pursued in numerous domains including database systems [15, 20, 33]. However, details on how PM will interact with existing systems are scarce at the moment and a possible role of PM is currently an open question. Recently developed Intel's Optane DC Persistent Memory extends the capacity of DRAM and enables a very large storage class memory. To manage data objects in such large space, the necessity of efficient indexing structures that take into account the properties of PM arises.

For block device storage systems, B+-tree variants have been widely used in the past decades to organize data blocks. B+-trees are designed to be updated in block granularity and as such copy-on-write and logging methods have been used to enforce failure-atomicity, consistency, durability, and even concurrency. On the other hand, as memory size has dramatically increased, various byte-addressable in-memory indexing structures such as T-tree, Radix-tree, Skip-List, CSS-tree [31], CSB-tree [32], and FAST (Fast Architecture Sensitive Tree) [14] have been developed. Since in-memory indexing structures are designed for volatile main memory, failure atomicity and recoverability were not of their primary concern.

Non-cache-conscious in-memory index such as T-tree, Radix-tree, and Skip List performed well in the past when CPU cache size was small in old processors. However, as CPU cache size has increased, B-tree-like in-memory indexes that take into account cache locality and arrange sorted keys in hierarchical blocks have been developed [14, 31, 32]. They were shown to outperform legacy in-memory indexes as they benefit from large-size CPU caches, ILP (instruction level parallelism), and MLP (memory level parallelism) of modern processors, i.e., they result in a fewer number of LLC and TLB misses than non-block-based index. That is, block-based data structures are no longer disk-based indexing structures but they have been shown to be efficient in-memory indexing structures as well.

For byte-addressable persistent memory, there have been various efforts to redesign B-trees [3, 11, 23, 37, 42]. One of the key challenges in designing byte-addressable B-tree for persistent memory is that it is not trivial to enforce failure-atomicity of page updates with a sequence of 8-byte atomic write operations. In particular, B-tree inserts or deletes a key in the middle of array to keep keys in sorted order. Such a sorting operation modifies a large portion of B-tree page, which results in transient inconsistent states [11]. Besides this, not all modern processors preserve *total-store ordering* and as such they often reorder memory writes to save the bandwidth [11]. Such memory access reordering can aggravate inconsistency problems. The memory access reordering is not a problem in volatile DRAM because volatile copies of B-tree pages are isolated from other transactions via a shared lock. That is, a write transaction copies B-tree pages to DRAM and makes changes to them. After the dirty B-tree pages become consistent, the modified pages are flushed to disk storage and become persistent. However, if memory is persistent, a volatile lock will be lost when a system crashes and partially updated dirty pages can be exposed to other transactions.

To resolve this problem, various B-tree variants, including NV-tree [42], FP-tree [30], wB+-tree [3], FAST and FAIR B-tree [11], have been proposed [23, 37, 42]. These works carefully use `clflush` and `mfence` instructions to update B-tree pages in a failure-atomic manner. In particular, NV-tree, FP-tree, and wB+-tree employ *append-only update* strategy to mitigate the sorting overhead. While wB+-tree performs expensive logging when a tree node splits, NV-tree and FP-tree

relax the consistency of internal nodes to avoid logging at the cost of reconstructing internal tree nodes upon a system failure. FAST and FAIR B-tree makes read transactions be aware of transient inconsistency and tolerate inconsistent B-tree pages such that it can keep records in sorted order and avoid the expensive tree reconstruction overhead. However, FAST and FAIR B-tree performs a large number of shift operations that can aggravate the wear-out problem of persistent memory.

In this work, we design and implement a novel variant of B-tree - *B^3-tree* (Byte-addressable Binary B-tree). In B^3-tree, keys stored in each page are represented in the form of a binary search tree, which allows records to be kept in sorted order without moving a large number of entries in a tree node. In that respect, B^3-tree is a hybrid index that combines B-tree and binary search tree. B^3-tree is write optimal as only two cacheline flushes are required to insert a new key-value pair into a B-tree node, i.e., one cacheline flush for storing the key-value record and the other cacheline flush for updating its parent node in binary search tree so that the key-value pair is stored in a sorted order. The update of the parent node in binary search tree behaves as a commit mark. With the binary search tree representation, the number of cacheline flushes is bounded by two as in most in-memory indexing structures. In addition to the number of reduced cacheline flushes, binary tree helps reduce the number of bit flips as it does not perform a large number of shift operations, which helps avoid cell wear-out and energy consumption.

While taking advantage of byte-addressable binary trees, B^3-tree does not lose the benefit of block-based B-tree index, i.e., locality in memory references, because similar keys are clustered in B^3-tree pages unlike binary trees that scatter each tree node across the entire physical memory space. Legacy binary trees fail to leverage memory locality and suffer from the fact that each binary tree node access results in accesses to different cachelines far from each other. In B^3-tree, we bound the memory space for each binary search tree to a single B^3-tree page so that nearby binary tree nodes are clustered in a few cachelines and the number of LLC misses is significantly reduced compared to legacy binary trees.

The contributions of this work are as follows:

- Persistent memory promises to encompass desirable features of byte-addressable memory and persistent storage systems. To benefit from such persistent memory, we design and implement a novel hybrid indexing structure B^3-tree that combines byte-addressable binary search trees and persistent B+-trees so that it reduces the number of dirty cacheline flushes.
- B^3-tree guarantees the failure-atomicity via fine-grained atomic 8-byte write operations. We develop a logging-less failure-atomic split and merge algorithms for B^3-tree to eliminate expensive logging.
- Through extensive performance study, we show that our B^3-tree outperforms other variants of persistent B-trees by a large margin in terms of insert, update, delete, and range query performance.

The rest of this article is organized as follows. In Section 2, we present previous works that are most relevant to ours. In Section 3, we present the design of B^3-tree. In Section 4, we evaluate the performance of B^3-tree against state-of-the-art byte-addressable persistent B-trees. In Section 5, we conclude the article.

2 BACKGROUND: RELATED WORK

As byte-addressable persistent memory is now on the horizon, numerous studies have been conducted to exploit new opportunities of its beneficial features in various domains including file systems and database management systems [1, 4, 7, 13, 15–17, 19, 21, 25–27, 33, 34, 37, 40, 42, 43]. In this section, we review some of these indexing work, which we deem most relevant to our study.

Systems on byte-addressable persistent memory have to safeguard against failures with fine-grained write atomicity. That is, in persistent memory, the granularity of failure-atomic writes is 8-bytes, or a cacheline if we use hardware transactional memory [20, 33]. In persistent memory, Such a small write granularity makes it difficult to guarantee consistency of various data structures including B+-trees because not all 8-byte writes can transform a consistent index into another consistent index. Moreover, independent store instructions can be arbitrarily reordered in modern processors. Therefore, a large number of expensive `clflush` and `mfence` instructions are required to guarantee the consistency of data structures [37].

To resolve this problem, Venkataraman et al. [37] proposed to use multi-versioning scheme so that we can roll back to previous consistent states. Version-based recovery methods, though, have a limitation in that it requires an expensive garbage collection.

Alternatively, Yang et al. [42] proposed NV-Tree that updates B-tree pages in *append-only* manner instead of making a large portion of data structures dirty so that it can reduce the number of calls to `clflush` and `mfence`. While the append-only update mitigates the overhead of write transactions, read transactions suffer from searching unsorted arrays. The append-only update scheme has been adopted by other persistent B-tree variants such as wB+-tree [3] and FP-tree [30]. wB+-tree proposed to use additional metadata to manage the ordering of keys in order to resolve the problem of unsorted keys. However, the additional metadata in wB+-tree requires additional cacheline flushes and puts a limitation on the number of entries in each tree node. Besides this, wB+-tree relies on expensive logging for tree rebalancing operations, which is sub-optimal.

FPTree, proposed by Oukid et al., is similar to NV-Tree in that it guarantees consistency only for leaf pages [30]. Different from NV-Tree, FPTree exploits hardware transactional memory to efficiently handle concurrency of internal tree page accesses and it reduces the cache miss ratio via *fingerprint*, which is one-byte hash value for keys stored in leaf page. While NV-Tree is designed for persistent memory-only systems, FPTree employs *selective persistence* where they store internal pages in volatile DRAM but leaf pages in persistent memory. Therefore, the internal pages in NV-Tree and FP-Tree may be lost upon system failure. Although the internal pages can be reconstructed from scratch using the leaf pages in persistent memory, the entire reconstruction process can hinder the instant use of the index.

FAST and FAIR B-tree proposed by Hwang et al. [11] does not employ append-only update strategy nor selective persistence. Instead, FAST and FAIR B-tree makes read transactions be aware of transient inconsistency and tolerate inconsistent B-tree pages such that it can keep records in sorted order and avoid the expensive tree reconstruction overhead. By making read transactions tolerate transient inconsistency, FAST and FAIR B-tree avoids expensive copy-on-write and logging. In addition, FAST and FAIR B-tree enables lock-free search since read transactions can tolerate transient inconsistent tree nodes while write transactions are making changes to them. However, a large number of shift operations employed in FAST and FAIR B-tree make the performance degrade as we increase the tree node size and aggravate wearing issues of persistent memory.

3 B^3-TREE: BYTE-ADDRESSABLE BINARY B-TREE

3.1 Node Structure of B^3-tree

B^3-tree is a hybrid index that combines binary search tree and B-tree to take advantage of both byte-addressability, balanced hierarchy, and cache locality. B^3-tree is a self-balancing binary tree index but it does not perform rotation operations unlike AVL-tree or red-black tree. Instead, it partitions the binary tree into fixed-sized sub-trees and performs rebalancing operations similar to that of B-tree. That is, B^3-tree groups a set of nearby BST nodes and stores them in a single

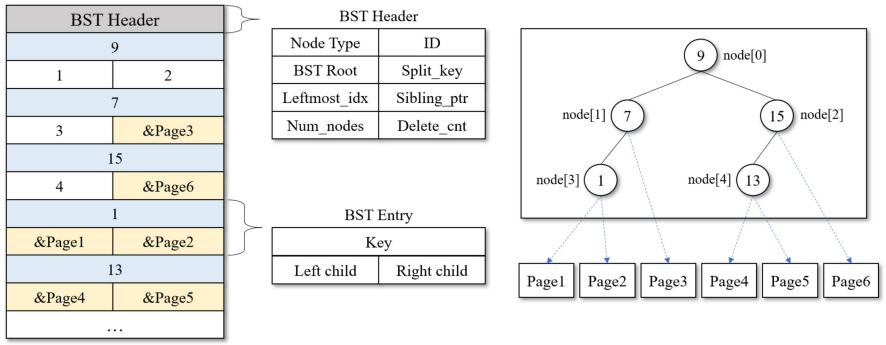


Fig. 1. Page Structure of B^3-tree.

B^3-tree node. I.e., a B^3-tree node uses a BST as its internal node representation. To avoid confusion, we need to distinguish a BST *node* that stores a single key and a child/value from a B^3-tree *node* that has multiple keys and child pointers/values. Therefore, we will refer to B^3-tree *node* as B^3-tree *page* and internal binary node as *BST node* hereafter.

Figure 1 illustrates the structure of B^3-tree page, which consists of a header and an array of keys and child pointers. The header stores three fields; (i) the *root offset* stores the offset of the root BST node, (ii) the *page type* field indicates whether the current page is a leaf page or an internal page, and (iii) the sibling pointer points to its external sibling page. We will describe the purpose of sibling pointer in Section 3.3.1.

The array of keys and child pointers stores an internal BST. In legacy BST, a child pointer is a memory address of another BST node. However, in our B^3-tree page, a child pointer is either the memory address of an external B^3-tree page or the index of another BST node in the same B^3-tree page. For example, `node[0]` (9) in Figure 1 has two BST child nodes. Thus, its child pointers are not memory addresses but indices of BST nodes.

In legacy BST, a tree node may have a single child. However, in our B^3-tree, all BST nodes have two children because BST nodes are created only when a B^3-tree page splits. For example, if `Page 3` splits, a new BST node will be created so that it points to `Page 3` and a new split page. And the new BST node is pointed by `node[1]`.

3.2 Failure-Atomic B^3-Tree Page Update

In this section, we discuss how B^3-tree achieves failure-atomicity for a single page update. For multiple page updates triggered by page split or merge, we defer our discussion to Sections 3.3.1 and 3.3.4.

3.2.1 Failure-Atomic Insertion. An insertion into a BST requires a single atomic 8-byte write operation since a new key is always added to a leaf node, which requires a single pointer update. In such a sense, BST is failure-atomic and write-optimal for insert operations.

Algorithm 1 shows the insertion algorithm of BST in B^3-tree. First, we check if the current B^3-tree page has available space for a new BST node (lines 1–8). If not, we check if there is any BST node that is not accessible from the BST root node. If an inaccessible BST node is found, we garbage collect it to make a space (line 2). If the current page is full, the BST in the page has to split (lines 3–5). If there is a space for a new data, the new data is written as a BST node in the found space (lines 9–11). Next, we increase the size of array (line 12). In line 15, we traverse the BST from root node to find the parent node and add the BST node to it. Depending on the key, the new BST

ALGORITHM 1: InsertBSTNode(key, left, right)

```
1:  if arraySize == MAX_COUNT then
2:      idx = lazyDefragmentation(this);
3:      if idx == -1 then
4:          return SPLIT_THIS_NODE;
5:      end if
6:  else
7:      idx = arraySize;
8:  end if
9:  node[idx].key = key;
10: node[idx].left = left;
11: node[idx].right = right;
12: arraySize++;
13: persist(&node[idx]);
14: persist(&arraySize);
15: parent = searchParent(key);
16: if key < node[parent].key then
17:     node[parent].left = idx;
18:     persist(&node[parent].left);
19: else
20:     node[parent].right = idx;
21:     persist(&node[parent].right);
22: end if
```

node becomes either a left or right child of the parent (lines 16–22). Note that the child pointer of the parent node stores the index of the new BST node.

Let us now discuss how B^3-tree tolerates various failures that can occur during the insertion algorithm that we described above. To persist memory writes, we call *persist()* function which calls a memory barrier and a cacheline flush instruction.

First, suppose a system crashes while we are looking for available space for a new BST node (lines 1–8). The failure will not result in inconsistency because we have not made any modification to the BST. Next, suppose a system crashes while a new BST node is being written (lines 9–11). Even if cacheline flush instruction is not explicitly called, the dirty cachelines can be flushed via cache replacement mechanisms. However, it still does not hurt consistency as the new BST node has not been added to the BST yet. Moreover, we have not increased the size of array. Hence, the partially written BST node will be ignored and overwritten by a subsequent insert transaction.

If a system crashes after we increase the size of array (line 12) and the updated size of array is flushed to persistent memory, the new BST node is no longer accessible because it is not pointed by any BST node but it is not released until a garbage collector reclaims it. We refer to such a node as *dead node*. Dead nodes waste memory space and hurt page utilization but they do not violate the invariants of index. Similarly, a system crash when calling *persist()* (lines 13–14) does not hurt the consistency of index. Note that the order of persisting the BST node and the size of array does not have to be preserved. Since dead nodes do not hurt the correctness of index, we garbage collect them in a lazy manner. When a B^3-tree page overflows, we check if there is any dead node that cannot be traversed from the the root node (line 2). If we find one, we use the dead space for a new BST node. If there is no such a dead node, we split the page (line 4). With such a lazy garbage collection scheme, we can postpone and reduce the overhead of searching an available free space for insertion. We note that writing a child pointer (index) in the parent

ALGORITHM 2: SearchBSTNode(key)

```
1: if isSiblingValid() && sibling→splitKey < key then
2:     return sibling→SearchBSTNode(key);
3: end if
4: idx = root;
5: while idx is a valid index do
6:     if key < node[idx].left then
7:         if node[idx].left is a BSTNode index then        // child is an index of another BSTNode
8:             idx = node[idx].left;
9:         else                                             // child is a pointer to a B3-tree page
10:            return node[idx].left;
11:        end if
12:    else
13:        if node[idx].right is a BSTNode index then       // child is an index of another BSTNode
14:            idx = node[idx].right;
15:        else                                             // child is a pointer to a B3-tree page
16:            return node[idx].right;
17:        end if
18:    end if
19: end while
```

BST node behaves as a commit mark of insert transaction (lines 16–22). Since the child pointer is 8 bytes long, the commit mark can be atomically written. Hence, the insert algorithm of B^3-tree is failure-atomic.

3.2.2 Search. Algorithm 2 shows how B^3-tree traverses a BST. As in legacy BST, we compare a given search key against the keys stored in BST nodes. However, since BST nodes are just metadata to keep child pointers in sorted order, `SearchBSTNode()` always returns a pointer to another B^3-tree page. That is, unlike BST, which stores user data in BST nodes, B^3-tree does not store user data in BST nodes but only in the leaf pages of B^3-tree. We note that, if a value stored in the child field is greater than the maximum size of array, the child is a pointer to a B^3-tree. Otherwise, the child is an index of another BST node.

For range query, once we find a leaf node by recursively calling `SearchBSTNode()`, we perform linear scanning since binary search traverses up and down the tree structure, which accesses the BST array irregularly and fails to leverage instruction-level parallelism and memory-level parallelism. Since a range query is likely to return a large number of data, brute-force linear scanning and filtering out non-overlapping data often performs better than traversing the tree structure up and down in sorted order.

3.2.3 Failure-Atomic Deletion. Deletion of a tree node in legacy BST is more complicated than insertion because the deletion of an internal BST node modifies multiple child pointers. In this section, we describe how we make the deletion of a BST node in an internal B^3-tree page failure-atomic.

In internal B^3-tree page, all BST nodes have two children, as shown in Figure 2, because a BST node is created when a B^3-tree page splits and every split creates a left and a right page. Note that we never delete a BST node if its both children are BST nodes. This is because we delete a BST node only when we delete a B^3-tree page. I.e., a BST node is deleted only when one of its child B^3-tree pages is deleted. Therefore, if a B^3-tree page is deleted, we simply delete the parent BST node by making the grandparent point to the sibling of the deleted page as shown in Figure 2. As

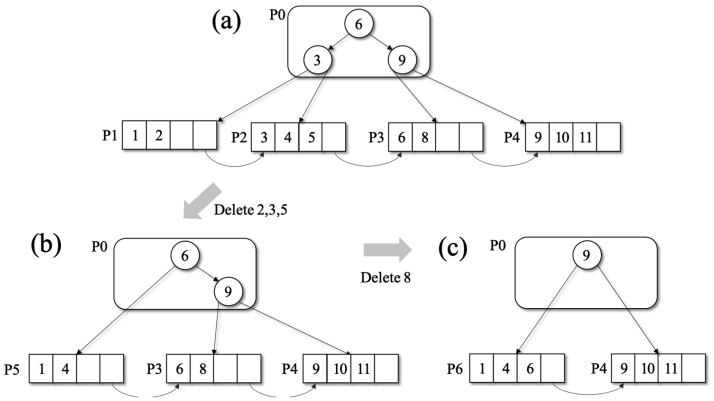


Fig. 2. Deletion in B^3-Tree Page.

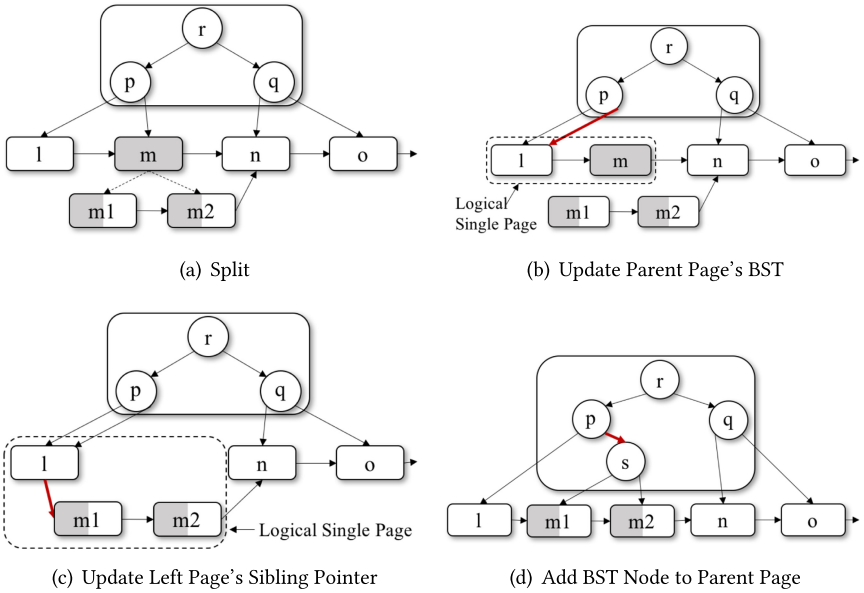
such, the delete operation in B^3-tree does not need complicated rotation operations unlike legacy BST. Since rotation operations require multiple pointer updates, self-balancing binary trees, such as AVL or red-black tree, are hard to make failure-atomic without logging.

3.2.4 Lazy Defragmentation. Note that the deletion algorithm of B^3-tree can leave a hole in the array of BST nodes. Such unused spaces in the array lower the page utilization and make subsequent queries access a larger number of cachelines. To mitigate the fragmentation problem, we let B^3-tree perform copy-on-write and garbage collect delete BST nodes in a lazy manner so that subsequent transactions can benefit from a fewer number of cacheline accesses. In lazy defragmentation, we allocate a new page and copy valid BST nodes from the fragmented page to a new page if there is no available space left on the page. During defragmentation, we not only delete invalid BST nodes but also reorganize the tree structure to build a complete binary search tree, thereby shortening the tree height. Even if a system crashes during defragmentation, it does not affect the correctness of the index because the incomplete modifications are made only in the new copy-on-write page. When the defragmentation is complete, we replace the original page with the new copy-on-write page. Note that lazy defragmentation trades off search performance for insertion performance. I.e., lazy defragmentation allows insert transactions to append new data without scanning free space in the array. However, read transactions suffer from a larger number of cacheline accesses. Although lazy defragmentation balances BSTs once in a while, as we will show in Section 4, lazy defragmentation improves the page utilization and balances the tree height.

3.3 Failure-Atomic Page Rebalancing

Insertions and deletions often result in page overflows and underflows, which require B^3-tree pages to split and merge respectively. Splitting and merging modify multiple pages to balance the tree height. However, multiple pages cannot be updated atomically. Therefore, legacy B-trees use expensive logging methods, which duplicate dirty pages and increase the write traffic.

Recently, several studies have been conducted to reduce or eliminate the logging overhead by employing byte-addressable persistent memory [11, 30, 42]. NV-tree [42] and FPtree [30] proposed *selective persistence*, which stores internal tree nodes in volatile DRAM but leaf nodes in persistent memory. If a system crashes, internal tree nodes can be reconstructed from leaf nodes in a bottom-up fashion so that logging can be avoided for internal nodes. However, such selective persistence is far from satisfactory because reconstruction of a large tree structure is expensive and it makes instant recovery almost impossible [11]. Instead, Hwang et al. [11] proposed the *Failure-Atomic*



(a) Split (b) Update Parent Page's BST

(c) Update Left Page's Sibling Pointer (d) Add BST Node to Parent Page

Fig. 3. Failure-Atomic Page Split in B^3-Tree.

In-place Rebalancing (FAIR) algorithm that eliminates the necessity of logging and performs in-place rebalancing operations. FAIR algorithm modifies the structure of FAST and FAIR B-tree in a predefined specific order that read transactions are aware of, thereby read transactions can ignore transient inconsistent tree structures.

Inspired by FAIR algorithm, we design a logging-less rebalancing algorithm for B^3-tree, i.e., we make B^3-tree split or merge in a particular order so that read transactions can tolerate transient inconsistent tree structures. Our rebalancing algorithm is different from FAIR algorithm in that FAST and FAIR B-tree shifts child pointers for rebalancing, but B^3-tree needs to update a BST of parent page.

As in B-link tree [22], every B^3-tree page has a *sibling pointer* so that all child pages of the same parent page can be managed as a linked list. When a page overflows, we allocate two new pages, connect them via a sibling pointer, redistribute the entries from the overflow page into two new pages. Then, we replace the overflow page with the two new pages. Since we cannot replace one page with two pages in a failure-atomic way, the sibling pointer in the new left page is used to combine the left page and the right page as in FAST and FAIR B-tree, so that they become a single logical page until their parent page adds a child pointer to the right page. When two pages merge, we merge them into a newly allocated page and replaces the underflow pages with the new one.

3.3.1 Page Split. Page split algorithm of B^3-tree is shown in Algorithm 3 and Figure 3 illustrates a working example. Suppose an insertion causes page *m* overflows. First, we allocate two new pages (*m1* and *m2*), and construct a balanced complete BST in each page using a half of page *m*'s key-value entries. Next, we update the sibling pointers of page *m1* and *m2*, as shown in Figure 3(a). We note that the new pages *m1* and *m2* are not added to the tree structure yet because no existing pages in B^3-tree has stored the addresses of the new pages. Hence, the two new pages cannot be accessed by other transactions and they are vulnerable to memory leak problems. We manage such potential memory leak using the *tri-state* scheme in user-level heap manager as was proposed by Kim et al. [15]. We defer a detailed discussion on this design to Section 3.4.1.

ALGORITHM 3: SplitPage(parent, n)

```
1: n1 = alloc();
2: n2 = alloc();
3: m = findMedian(n);
4: copyEntries(n, 0, m, n1);
5: copyEntries(n, m, ∞, n2);
6: n1.sibling = &n2.id;
7: n2.sibling = n.sibling;
8: persist(&n1);
9: persist(&n2);
10: leftSibling = findLeftSiblingPage(node);
11: swapAndPersist(parent, n, leftSibling);
12: leftSibling→sibling = n1;
13: persist(&leftSibling→sibling);
14: InsertBinaryTreeNode(m, n1, n2);
```

Second, we replace the pointer to the overflowing page in the parent page's BST with the left sibling of overflowing page as shown in Figure 3(b). In the example, the BST node *p*'s left and right child pointers point to the same page *l*. Although we removed the pointer to the overflowing page *m*, we can still access page *m* in this state. This is because we consider page *l* and *m* as a single logical page. In particular, we make transactions visit the right sibling page if the parent BST node has the same left and right child pointers and the right child pointer is chosen. Alternatively, we can make transactions follow the right sibling pointer if a given search key is greater than the largest key in the page.

Next, we set the sibling pointer of page *l* to the address of page *m1* as shown in Figure 3(c). By making the left sibling page point to the new split pages, we can atomically remove the overflowing page *m* and add the two new split pages *m1* and *m2*. We note that three pages *l*, *m1*, and *m2* are a single logical page.

Finally, we add a new BST node for page *m1* and *m2* in the parent BST. In the example shown in Figure 3(d), we create a BST node *s* that has page *m1* and *m2* as its left and right child. Then, we atomically replace the right child pointer of BST node *p* with the index of BST node *s*. This completes the split operation.

3.3.2 Crash Recovery during Page Split. While a page is splitting, various system failures can occur. Suppose a system crashes while copying data from the overflow page to new pages. Unless the original overflow page is not modified, B^3-tree is in consistent state.

Suppose a system crashes after we make the left and right child of the parent BST node point to the same left sibling page, as shown in Figures 3(b) and 3(c). A recovery process will detect the parent BST node has the same child pointers while scanning the index. This transient inconsistent state can be easily fixed by traversing right sibling pages and adding them to BST so that they can be directly accessed without using sibling pointers. We note that this recovery algorithm does not need a separate log file.

In addition, it is noteworthy that read transactions always succeed finding a key even if a node split has not completed. For example, if a query looking for a key in *m2* is submitted when *m2* is not added, as shown in Figure 3(c), the query will access *l* first, *m1* next, and then *m2*.

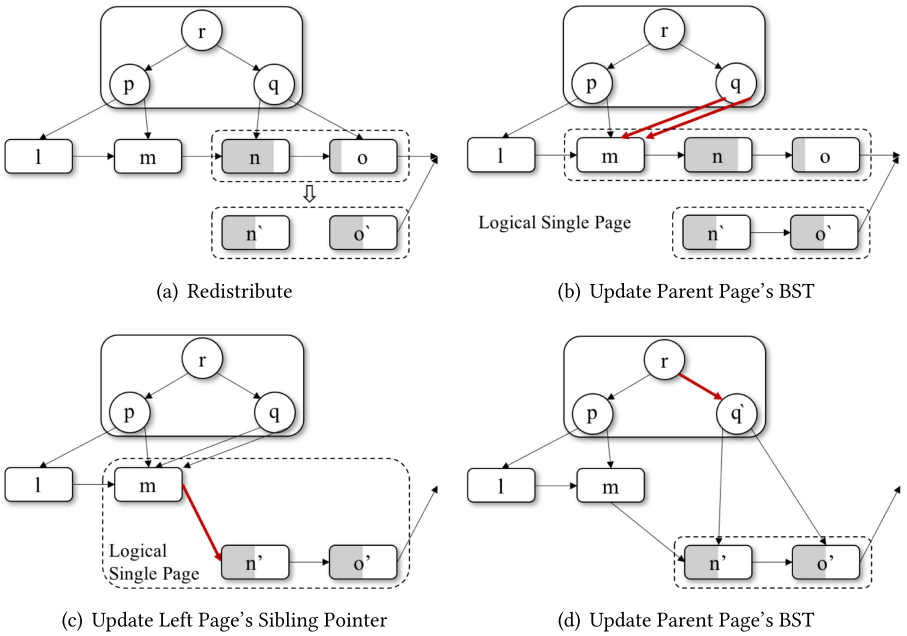
3.3.3 Failure-Atomic Page Merge. Page merge algorithm is similar to the page split algorithm, but the order of operations is reversed. The detailed page merge algorithm of B^3-tree is shown in Algorithm 4.

ALGORITHM 4: MergeTreePages(leftPage, rightPage)

```
1:  lp = parentBSTNode(leftPage);          // lp is a parent BSTNode that points to leftPage to be deleted
2:  rp = parentBSTNode(rightPage);         // rp is a parent BSTNode that points to rightPage to be deleted
3:  mergedPage = merge(leftPage, rightPage);   // insert data from underutilized pages into a new page
4:  leftSibling = findLeftSiblingPage(leftPage);          // find a left sibling of mergedPage
5:  lp→swapAndPersist(leftPage, leftSibling);             // make lp point to leftPage
6:  rp→swapAndPersist(rightPage, leftSibling);            // make rp point to rightPage
7:  leftSibling→sibling = mergedPage;                     // set mergedPage as its right sibling
8:  persist(&leftSibling→sibling);
9:  if lp == rp then                                      // if lp and rp are the same BSTNode
10:    grandParent = parentBSTNode(lp);
11:    grandParent→swapAndPersist(lp, mergedPage);        // remove lp and rp
12: else if lp.left == leftPage then                      // rp is the leftmost child of right sibling sub-tree
13:    rp.left = mergedPage;                              // make rp point to mergedPage (need to remove lp)
14:    persist(&rp.left);
15:    if node→root == lp then                            // lp to be deleted is the current root
16:       currentPage→root = lp.right;                    // make the right sibling sub-tree a new root node
17:       persist(&currentPage→root);
18:    else
19:       grandParent = parentBSTNode(lp);                // remove lp
20:       grandParent→swapAndPersist(lp, lp.right);
21:    end if
22: else if lp.right == leftPage then                     // rp is the leftmost child of its parent's right sub-tree
23:    lp.right = mergedPage;                             // make lp point to mergedPage (need to remove rp)
24:    persist(&lp.right);
25:    if node→root == rp then                            // rp to be deleted is the current root
26:       currentPage→root = rp.left;
27:       persist(&currentPage→root);
28:    else
29:       grandParent = parentBSTNode(rp);                // remove rp
30:       grandParent→swapAndPersist(rp, rp.left);
31:    end if
32: end if
33: node→delete_cnt++;
```

The simplest case is when we merge two pages that are pointed by the same BST node (lines 9–12). In this case, we make their grand parent BST node point to the merged page. A working example of this case can be illustrated by reversing the order of the steps shown in Figure 3. First, we remove the BST node for underutilized pages from the parent page's BST as shown in Figure 3(c). This will make *l*, *m1*, and *m2* a single logical page. Then, we allocate a new page (*m* in the example) and copy entries from underutilized pages (*m1* and *m2*) to the new page *m*. Note that we sort the keys and construct a complete BST in the merged page. When the merged page is ready, we atomically update the sibling pointer of the left sibling page (*l* in the example) as shown in Figure 3(b). Finally, we update the parent page's BST so that the merged page (*m*) is pointed by the parent BST. I.e., updating a child pointer of a grandparent BST node behaves as a commit mark of the merge operation, and it can be done via a single 8-byte atomic write, hence it is failure-atomic.

A more complicated case is when we merge two pages that are pointed by different BST nodes. Due to the pairwise split and merge algorithm, the parent of adjacent page must be the leftmost (or



(a) Redistribute (b) Update Parent Page's BST

(c) Update Left Page's Sibling Pointer (d) Update Parent Page's BST

Fig. 4. Failure-Atomic Redistribution.

rightmost) BST node in the right (or left) sibling sub-tree. Suppose we merge page l and $m1$ in the example shown in Figure 3(d). In this case, we delete the parent BST node of left page (BST node p in the example) (lines 13–21). If the parent BST node of left page is the root node, we choose a new root node. Otherwise, we remove the parent BST node by updating the child pointer of grandparent BST node (r in the example). We omit the discussion of other cases since they are all symmetric cases. But the detailed algorithm and comments are shown in Algorithm 4.

3.3.4 Failure-Atomic Redistribution. If the utilization of two sibling pages drops below a threshold value (e.g., 50% in our implementation), we perform merge operation. However, if the utilization of a single page drops below a threshold value, most of B-tree variants perform redistribution, that is, the underutilized page borrows some entries from its sibling page.

Redistribution is similar to the merge algorithm. We use the example depicted in Figure 4 to walk through the detailed workings of failure-atomic redistribution. First, we allocate two new pages (n' and o') and redistribute entries from the two underutilized pages, as shown in Figure 4(a). Next, we remove the pointers to underutilized pages (n' and o') from the parent BST, as shown in Figure 4(b). Updating the two child pointers of q does not have to be atomic but its left child pointer must be updated before the right child pointer. Making the left child pointer point to page m will make page m and n a single logical page. Then, making the right child pointer point to page m will make three pages — m, n, and o —a single logical page. We note that these operations do not affect the invariants of the index, hence they are failure-atomic. In the next step, we update the sibling pointer of the leftmost page in the single logical page (m in the example), as depicted in Figure 4(c)), so that the underutilized pages are replaced by the redistributed pages — n' and o'. Finally, we replace BST node (q) in the parent page with a new BST node (q' in the example), as shown in Figure 4(d). Updating the right child pointer of r via atomic 8-byte write operation commits the redistribution.

3.3.5 Lazy BST Rebalancing. BST always adds a new tree node as a child of a leaf node. Therefore, a single failure atomic cache line flush is sufficient to insert a new node. Although BST is good for failure-atomic insertion, BST suffers from the notorious *skew* problem. However, the number of BST nodes in B^3-trees and the height of skewed BST is bounded by the page size. When the page size is 4 KB, maximum 168 8-byte keys can be indexed in our implementation. Therefore, the height of BST can be no higher than 168 in the worst case.

Although the skewness of B^3-tree page is bounded, skewed BSTs can degrade the page access performance in terms not only of search but also of insertion as well. To mitigate the skew problem of BST, we make B^3-tree reconstruct a complete BST when a page is split or when underutilized pages merge. To convert a skewed BST into a complete BST, we in-order traverse the skewed BST and store the result in the BST node array. This optimization takes $O(n)$ time where n is the number of BST nodes. This rebalancing optimization not only shortens the tree height but also improves cacheline locality. Note that splitting or merging modifies a large portion of pages. Hence, we let B^3-tree perform copy-on-write instead of in-place updates. As such, the overhead of calling a large number of `clflush` for rebalancing BSTs is masked by the copy-on-write overhead.

3.4 Discussion

3.4.1 PM Allocator. The persistent memory allocator should provide a way of guaranteeing the failure atomicity for memory allocation and deallocation as well. Intel Persistent Memory Development Kit (PMDK) is a collection of libraries that provide general facilities for persistent memory programming for Optane DC Persistent Memory (DCPM). In *app direct mode* of DCPM, PM is accessed by the operating system as memory-mapped files called *pools*. Applications do not have to create a memory-mapped file using `mmap`, but it is created via PMDK API function `pmemobj_create()`. With the memory region mapped, an object in the pool can be accessed by a 16-byte *persistent pointer*, which consists of *pool id* and *offset* in the pool. The persistent pointer in PMDK raises a challenge in persistent indexes. That is, the size of the persistent pointer is 16 bytes, which cannot be updated by a single store instruction. To avoid this problem, we need to use a single pool ID for the index so that we can atomically update 8-byte offsets only. One of the challenges in dynamic index is that it is hard to know how large an index can grow or shrink. If we allocate a too large PM space for a small index, the unused space will be wasted and the utilization of PM will be low. To walk around this problem, PMDK provides the auto-growing directory-based poolset. If a directory is specified as the path of a pool, multiple memory mapped files of 128 MBytes are dynamically created in the directory and the pool grows in 128 megabyte increments on demand. With the auto-growing directory-based poolset, a dynamic index can be mapped to a single pool ID. Therefore, when a tree structure is rebalanced, only the 8-byte offset of each page needs to be updated. As such, B^3-tree is failure-atomic. It is noteworthy that we observed the performance overhead of directory-based poolset is at most 3% higher than that of a single file-based pool in our experiments.

3.4.2 Updates and Variable KV Support. Our implementation of B^3-tree stores values not in internal pages but only in leaf pages of B^3-tree. In internal pages, each BST node stores a key and two child offsets or pointers. In leaf pages, we store values along with keys. However, B^3-tree supports only 8-byte values because of update operations. If a write transaction updates a value of an existing data item in B^3-tree, the in-place update must atomically update the value. However, if the value size is greater than 8-bytes, values cannot be atomically updated in-place. Therefore, if the value size is greater than 8-bytes, we store values in a separate log-structured object space and store its pool offset (pointer) in a B^3-tree leaf page. We note that storing the pool offset in a B^3-tree leaf page behaves as a commit mark.

Consider a system crashes after we store the value in a separate log-structured object but before we add its offset to a leaf page. Since the value is not exposed to other transactions, we need to garbage collect the object from the log-structured object space. There can be various ways of garbage-collecting uncommitted values. Simply, we can take a brute-force method, i.e., we scan leaf nodes and check which values in the log-structured object space are not pointed. Alternatively, we can take a logging approach, i.e., we keep a list of key-value pairs for uncommitted values (garbage-collection log). That is, before we add a value to the log-structured object space, (step 1) we store its key and the value to the garbage collection log. Then, (step 2) we store its value to the log-structured object space. Next, (step 3) we add its key and 8-byte offset to the index as described earlier, which commits the transaction. Then, (step 4) we delete the key and value from the garbage collection log.

When a system crashes, we search the index using the garbage collection log. If a key is not found in the index but its value is found at the end of log-structured object space, we delete the value from the log-structured object space. If both a key and its value are not found, we simply ignore the garbage-collection log. If both a key and its value are found, no recovery is necessary.

3.5 Concurrency Control

With the growing number of cores in modern computer architectures, non-blocking access to concurrent data structures is drawing more attention in the community. To improve concurrent access to the index, FAST and FAIR B-tree [11] eliminates the necessity of read locks by making read transactions tolerate transient inconsistent status.

Similar to FAST and FAIR B-tree, a sequence of 8-byte store instructions used by B^3-tree also guarantees the consistency of data structures as described earlier. Hence, access to the shared B^3-tree does not have to be serialized and enables lock-free search as in FAST and FAIR B-tree. That is, even if a write thread fails or a system crashes while making changes to B^3-tree, no read thread will ever access inconsistent tree nodes because every single store instruction in B^3-tree does not affect the invariants of index and guarantees correct search results. Therefore, read operations do not have to wait for write transactions to finish updates and to release exclusive locks.

On the other hand, if multiple write threads update the same tree nodes simultaneously, B^3-tree suffers from write-write conflicts. Suppose a write thread is about to add a BST node to a leaf node and another thread is trying to delete the leaf node at the same time. If the insertion succeeds right before the leaf node is deleted, the newly inserted node will not be a part of the index, which leaves the index inconsistent. Therefore, B^3-tree does not allow concurrent modifications to the same page. A write thread must acquire an exclusive lock when modifying a page.

However, in various applications including enterprise database systems, read transactions are much more popular than write transactions. Hence, lock-free search can significantly improve the overall throughput. However, we note that the lock-free search algorithm can not guarantee the serializability of concurrent transactions. I.e., without read locks, transactions may suffer from dirty reads and phantom reads [11]. To guarantee the serializability of concurrent transactions, if a transaction reads or writes multiple data items in leaf pages, read/write locks must be used only for leaf nodes, as was proposed by FAST and FAIR B-tree [11]. Since read-write conflict more in upper-level tree structures rather than leaf pages, employing read-write locks in leaf pages does not significantly hurt the concurrency level.

3.5.1 Page Deallocation with Lock-Free Search. It is noteworthy that memory management of lock-free data structures is quite challenging. For example, deallocated pages can be still being read by other read transactions. To deal with memory reclamation for lock-free data structures, various solutions, such as *quiescent-state-based reclamation*, *epoch-based reclamation*, and

hazard-pointer-based reclamation, have been proposed in the literature [2, 5, 8]. In our implementation of B^3-tree, we employ a simple version of epoch-based reclamation. That is, we do not deallocate tree pages immediately but wait for concurrent transactions to finish in order to make sure no other transaction accesses the logically deleted page. I.e., when a page is to be deleted, we do not physically deallocate it, but add it to a list of logically deleted pages. A logically deleted page is not reused until all the queries that might access the page complete. To figure out which queries were active before a page is logically deleted, we assign each query a monotonically increasing ID and manage a list of IDs of currently active queries. We note that the list is usually small and does not incur significant performance overhead unless there are a very large number of concurrent queries. When a tree page is removed from the index, we tag the page with the largest ID (the latest ID) of the currently active queries and add it to the list of logically deleted pages. These logically deleted pages are physically deleted only if the smallest query ID (the oldest ID) of the currently active queries is greater (newer) than the ID of logical deleted page. I.e., all the current queries were submitted after the page was removed from the index. Therefore, none of the current queries can access the page and it is safe to physically delete it. In our implementation, the garbage collection occurs when a subsequent query tries to allocate a new page. We also note that B^3-tree design is independent of lock-free page deallocation schemes. That is, other page reclamation schemes such as quiescent-state-based reclamation and hazard-pointer-based reclamation schemes can be used for B^3-tree.

4 EVALUATION

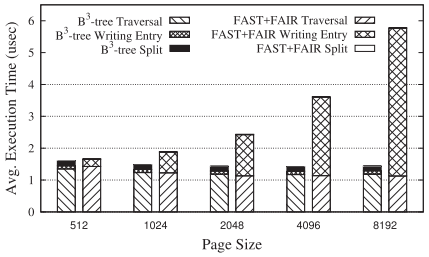
4.1 Experimental Setup

We run experiments on a workstation that has two Intel Xeon Gold 5125 processors (20 cores, 2.5 GHz, 20 x 32 KB instruction cache, 20 x 32 KB data cache, 20 x 1,024 KB L2 cache, and 13.75 MB L3 cache), 64 GB of DDR4 DRAM, and 128 GB Intel Optane DC Persistent Memory (DCPM). We implemented B^3-tree using the Persistent Memory Development Kit (PMDK), which is designed to facilitate programming for persistent memory. To make use of atomic 8-byte instructions, we create an auto-growing directory-based persistent memory poolset for the entire tree structure and call `pmemobj_alloc()` for each page to obtain 8-byte offset for each tree page. For failure-atomicity, we carefully enforce the ordering of `mfence` and `clwb` instruction instead of using the PMDK transaction APIs. We also ported wB+-tree and FAST and FAIR B-tree implementations to PMDK so that we can compare their performance against B^3-tree.

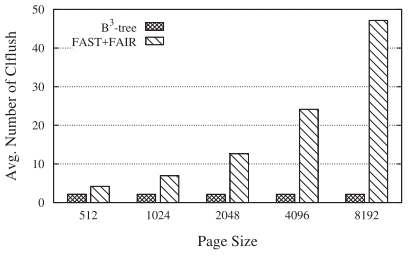
In addition to Intel Optane DCPM, which has been recently released on the market, other persistent memory devices such as phase-change memory (PCM) [41] and spin transfer torque MRAM [9] are also being considered as competitors to Optane DCPM. Since these emerging persistent memory devices are expected to have a lower access latency than Optane DCPM, we evaluate the persistent index by making use of a DRAM-based PM latency emulator - *Quartz* [18, 38] and varying the latency of persistent memory.

4.2 Performance Effect of Page Size

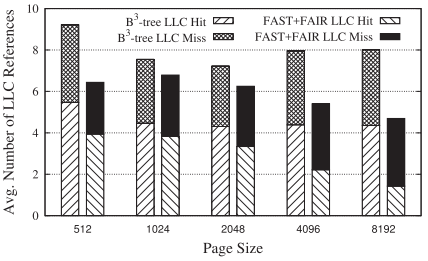
In the first set of experiments shown in Figure 5, we insert 10 million random keys and values of 8-bytes in uniform distribution into an index on DCPM. It is noteworthy that B+tree structure is relatively insensitive to key distribution compared to other indexes such as radix tree [20] because B+tree selects a median key as a separator when a node splits. For various key distributions, the median keys adapt to actual key distribution. In the experiments, we measure the insertion performance of B^3-tree and FAST and FAIR B-tree while we vary the page size. As we increase the page size from 512 Bytes to 8 KBytes, each B^3-tree page can hold 18 (512 bytes), 40 (1 KB),



(a) Insertion Time per Query



(b) Number of clflush per Query



(c) Number of LLC Accesses per Query

Fig. 5. Insertion Performance with Varying Page Sizes (Optane DCPM).

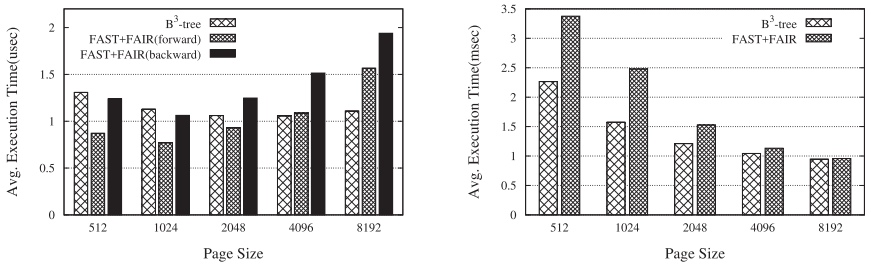
82 (2 KB), 168 (4 KB), and 338 (8 KB) key-value pairs while FAST and FAIR B-tree can hold 30, 62, 126, 254, and 510 key-value pairs. We do not show the performance of wB+-tree in this experiments because wB+-tree does not allow us to change the page size. Due to the 8-byte bitmap, wB+-tree can hold maximum 63 entries in a page.

As shown in Figure 5(a), the insertion time of B^3-tree is insensitive to the page size and it outperforms FAST and FAIR B-tree. As we increase the page size, the insertion time of FAST and FAIR B-tree suffers from a large number of shift operations but B^3-tree writes only three cachelines no matter how large the page size is, i.e., B^3-tree increases the size of BST, appends a new BST leaf node, and updates its parent node. While B^3-tree performs a constant number of write operations regardless of the page size, a large page size may increase the BST traversal time, but read operations often benefit from cache hits and our lazy rebalancing optimization keeps the BST height close to $O(logk)$, where k is the number of BST nodes in a page.

Figure 5(b) shows the average number of `clflush` instructions. Because of a large number of shift operations, FAST and FAIR B-tree calls at least a twice and up to 23 times larger number of `clflush` instructions than B^3-tree. Note that, FAST and FAIR B-tree shows a comparable performance with B^3-tree when the page size is 512 Bytes, but FAST and FAIR B-tree calls a larger number of `clflush` instructions. This is because FAST and FAIR B-tree benefits from memory locality and memory prefetching whereas B^3-tree accesses non-contiguous cachelines due to the binary search. However, since the write latency of persistent memory is much higher than read latency, non-contiguous cacheline access does not significantly affect the insertion performance.

Figure 5(c) shows the number of LLC references, i.e., LLC hits and LLC misses, per insertion query. B^3-tree accesses a larger number of cachelines than FAST and FAIR B-tree due to the non-contiguous cacheline accesses. Therefore, the average number of LLC references of B^3-tree is higher than that of FAST and FAIR B-tree.

In the experiments shown in Figure 6, we measure the search performance of B^3-tree with varying the page size. FAST and FAIR B-tree uses a *scan direction flag* to guide in which direction



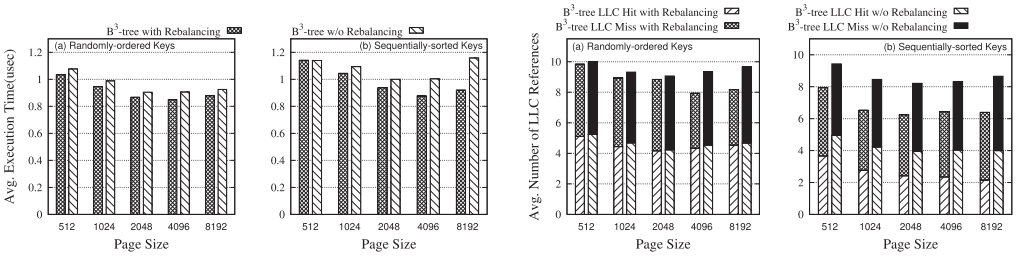
(a) Exact Match Search Time per Query (b) Range Search Time per Query

Fig. 6. Search Performance with Varying Page Sizes (Optane DCPM).

subsequent read transaction should scan a tree node. This flag is required because a read transaction must not miss any array elements while they are being shifted to right or left. I.e., if a sorted array is shifted from right to left by a delete transaction, subsequent read transactions need to read the array in descending order. The scan direction flag not only increases the complexity of search algorithm implementation but also hurts the memory access performance [36]. Therefore, we show the exact match search performance of FAST and FAIR B-tree with two configurations. In the first configuration, denoted as `FAST+FAIR(forward)`, we set the flag of all tree pages to the forward direction. This is the optimal case for FAST and FAIR B-tree since it benefits from memory level parallelism and hardware prefetechers. Data Cache Unit (DCU) prefetcher in Intel Xeon CPUs, also known as the streaming prefetcher, prefetches data from PM 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. As a result, `FAST+FAIR(forward)` shows the higher search performance than `FAST+FAIR(backward)`.

Due to the larger number of LLC references, the exact match search performance of B^3-tree is about 40% worse than that of `FAST+FAIR(forward)` B-tree when the page size is 512 Bytes. However, the performance of `FAST+FAIR(backward)` is much worse than that of `FAST+FAIR(forward)` and similar to that of B^3-tree. It is also noteworthy that, as we increase the page size, the search performance of B^3-tree improves while FAST and FAIR B-tree suffers from a larger number of cacheline accesses due to linear scanning. Considering that the insertion performance of B^3-tree is insensitive to the page size, the best page size for B^3-tree is 4 KBytes while FAST and FAIR B-tree shows the best performance when the page size is 512 Bytes. With these two different page size configurations, the exact match search performance of B^3-tree is up to 15% slower than FAST and FAIR B-tree.

Figure 6(b) shows that the range query execution time of both persistent indexes. When the page size is 512 Bytes, the range query execution time of FAST and FAIR B-tree is about 1.5x higher than that of B^3-tree. As we increase the page size, the range query execution times of both indexes decrease because both indexes benefit from locality and hardware prefetching. This result shows that there is a trade-off in choosing a page size for FAST and FAIR B-tree. That is, when the page size is as small as 512 Bytes, FAST and FAIR B-tree shows good insertion and exact match search performance but range query suffers from poor locality. On the contrary, B^3-tree is rather insensitive to the page size for insertion and exact match search performance, but it benefits from a larger page size for range query. Considering that the tree page size of persistent index is not statically determined by disk page size as in legacy disk-based indexes, but it is just a performance tuning parameter in PM, we choose to use 4 KBytes page size for B^3-tree for the rest of experiments unless stated otherwise. For FAST and FAIR B-tree, we use 512 Byte page size since it shows the



(a) Search Time per Query (b) Number of Cache Line Accesses per Query

Fig. 7. Search Performance: Performance Effect of Lazy Rebalancing (Optane DCPM).

fastest insertion and exact match query performance although the range query performance is sub-optimal.

Figure 7 shows how the rebalancing optimization helps improve the exact match query performance by decreasing the tree height of BSTs in B^3-tree. The performance of BST is often affected by the distribution of keys and insertion order. If we insert keys in sorted order, a BST is completely skewed and the tree height can be as high as the number of BST nodes ($O(n)$), which is the worst case in BST. Therefore, we evaluate the performance of B^3-trees using two types of workloads—random keys and monotonically increasing keys.

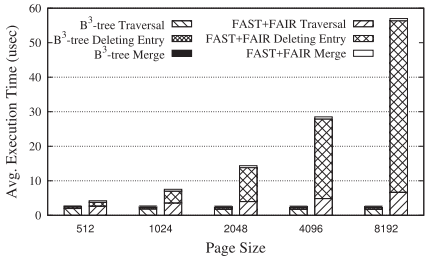
If we disable the lazy rebalancing optimization and insert monotonically increasing keys, the height of BST becomes linearly proportional to the page size. However, Figure 7 shows that search performance of B^3-tree improves as we increase the page size. This is because the tree height of B^3-tree decreases as the page size increases. I.e., although a larger page size aggravates the skew problem of BST, the size of BST in B^3-tree is often very small and does not significantly affect the overall tree traversal time. Our lazy rebalancing optimization helps rebalance internal BSTs and reduce the number of accessed cachelines and the number of LLC misses as shown in Figure 7. As a result, lazy rebalancing improves the search performance by up to 25.8%. We note that the performance gain becomes larger as the page size increases.

In the experiments shown in Figure 8, we evaluate the deletion performance. Similar to the insertion and search performance, the deletion time of B^3-tree is also insensitive to page size. Figure 8(b) shows that B^3-tree calls about 2.5 cacheline flushes per deletion on average while FAST and FAIR B-tree calls up to 67 cacheline flushes. Although the page merge algorithm of B^3-tree requires multiple cacheline flushes due to the copy-on-write, most deletions require only a single cacheline flush to reset the parent node's child offset. Therefore, the average number of cacheline flushes is much smaller than that of FAST and FAIR B-tree that shifts a large number of cachelines.

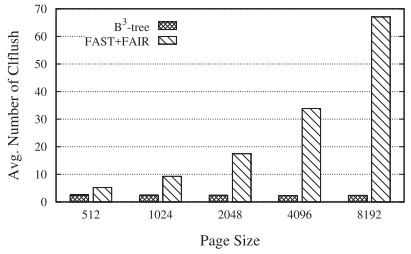
4.3 Performance Effect of Index Size

In the experiments shown in Figure 9, we vary the number of indexed key-value pairs and measure the insertion and search throughput. For FAST and FAIR B-tree, we set the page size to 512 Bytes as it shows the fastest insertion and search performance. Since the page size of wB+-tree cannot be controlled, we set its page size to 1 KBytes and its bitmap to 8 bytes so that it can hold maximum 64 entries. For NV-tree, we set its page size to 4 KBytes as it shows the fastest insertion and search performance. For B^3-tree, we set the page size to 4 KBytes as it shows the fastest range query performance. Again, it should be noted that there is no reason to use the same page size for all persistent indexes since the page size is just a tuning parameter.

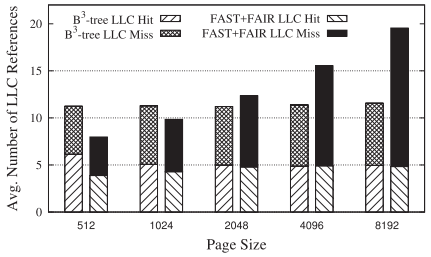
In terms of the insertion throughput shown in Figure 9(a), B^3-tree consistently outperforms all other indexes. Note that B^3-tree shows about 1.5x and 3x higher insertion throughput than FAST



(a) Deletion Time per Query

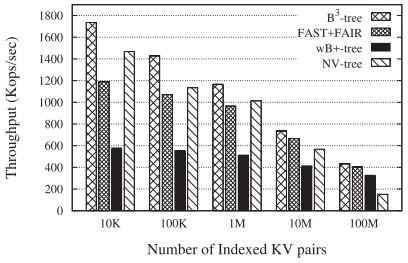


(b) Number of Cache Line Flushes

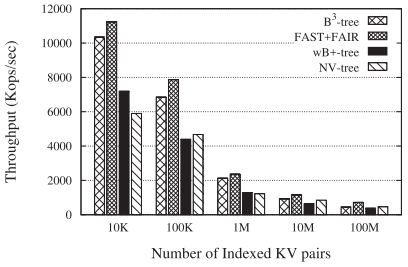


(c) Number of Cache Line Accesses per Query

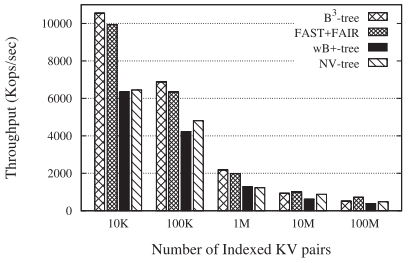
Fig. 8. Deletion Performance with Varying Page Sizes (Optane DCPM).



(a) Insertion Throughput



(b) Search Throughput



(c) Search Throughput After Deletion

Fig. 9. Throughput Comparison with Varying Number of Indexed Data (Optane DCPM).

and FAIR B-tree and wB+-tree, respectively, when we index 10,000 data. As we index a larger number of data, the performance gap between B^3-tree and FAST and FAIR B-tree decreases due to the split overhead. When a page splits, B^3-tree performs the expensive copy-on-write for two new sibling pages so that it can rebalance BSTs. However, unlike B^3-tree, FAST and FAIR B-tree creates only one sibling page and performs the in-place update for the overflowing page. As a result, FAST and FAIR B-tree modifies a fewer number of cachelines than B^3-tree when a split occurs. wB+tree also creates one sibling page when a page splits. However, its insertion performance is worse than B^3-tree because it calls at least four cacheline flush instructions while B^3-tree calls only two. I.e., wB+tree (i) writes a key-value record, (ii) invalidates slot-array by updating the bitmap, (iii) updates the slot-array, and (iv) and validates the slot-array by updating the bitmap. Besides, wB+tree requires expensive logging for tree rebalancing, which aggravates its poor insertion performance. NV-tree shows high insertion throughput when the number of indexed data because it also requires only two cacheline flushes per insertion. However, the insertion throughput of NV-tree degrades sharply as we index more data. This is because each split of internal node results in the reconstruction of the entire internal nodes since NVTree requires all internal nodes to be stored in consecutive memory blocks to exploit cache locality.

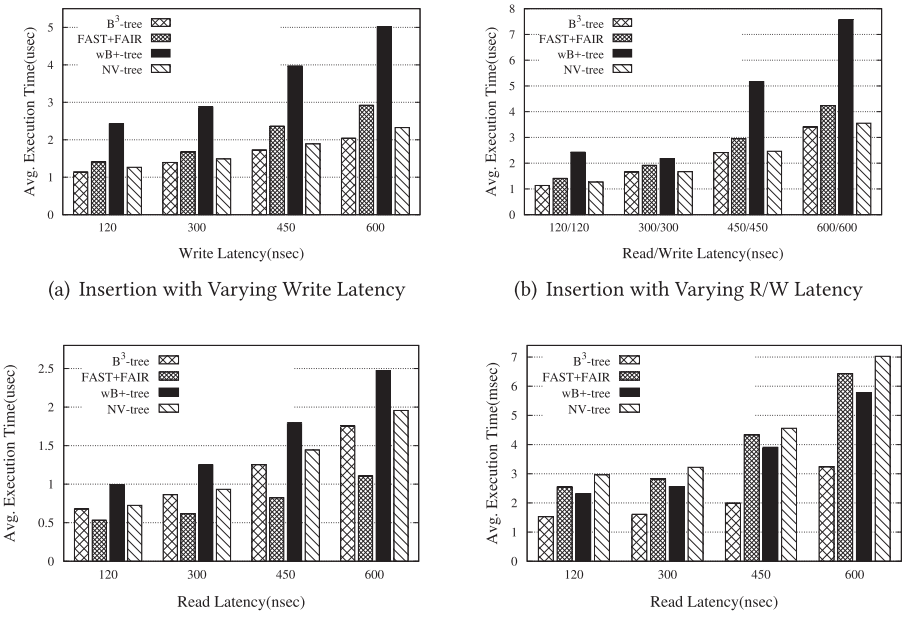
As for the search throughput, shown in Figure 9(b), B^3-tree is slightly outperformed by FAST and FAIR B-tree because of a larger number of LLC misses. In this experiments, we do not submit any delete query. Therefore, all tree pages are accessed in ascending order, which is optimal for FAST and FAIR B-tree. As a result, FAST and FAIR B-tree outperforms *B*3-tree since B^3-tree performs binary search and each query jumps around cachelines and it fails to benefit from prefetching and memory locality.

In the Figure 9(c), we deleted 10% of records after we create indexes. As a result, FAST and FAIR B-tree accesses about 10% of tree pages in descending order. As a result, FAST and FAIR B-tree is outperformed by B^3-tree when the number of indexed key-value records is smaller than 10 millions. However, as the index size increases, B^3-tree suffers more from a larger number of LLC misses and it is outperformed by FAST and FAIR B-tree.

4.4 PM Latency Effect

In this section, we evaluate the performance of persistent indexes using a DRAM-based PM emulator - Quartz. As such, we do not use Intel's PMDK library for the experiments shown in Figure 10. Although Optane DCPM is the first commercial PM product on the market, other persistent memory devices, such as STT-MRAM and PCM, are expected to be available as DIMM devices and they are expected to provide various latencies. Quartz consists of a kernel module and a user-mode library that models application-perceived latency by injecting stall cycles into each predefined time interval, called *epoch*. We configure minimum and maximum epochs to 5 us and 10 us for our experiments. We adjust read latency of PM from 300 ns up to 600 ns. We note that 300 ns is the minimum latency that we can configure in our testbed environment. To emulate write latency, we inject stall cycles after each `clflush` instructions, as was done by [10, 15, 20, 33, 39]. Note that PM write latency is often hidden by CPU cache. Hence, we do not add the software delay to `store` instructions. As for the PM bandwidth, we assume that PM bandwidth is no different from that of DRAM since Quartz does not allows us to emulate both latency and bandwidth at the same time.

We note that we run this experiments on a different testbed since Quartz is not supported by the latest Gold processors that supports Optane DCPM but only by old Haswell processors. Therefore, we run Quartz experiments on a workstation that has four Intel Xeon Haswell-EX E7-4809 v3 processors (8 cores, 2.0 GHz, 8x32 KB instruction cache, 8x32 KB data cache, 8x256 KB L2 cache, and 20 MB L3 cache) and 64 GB of DDR3 DRAM.



(a) Insertion with Varying Write Latency

(b) Insertion with Varying R/W Latency

(c) Search with Varying Read Latency

(d) Range Search with Varying Read Latency

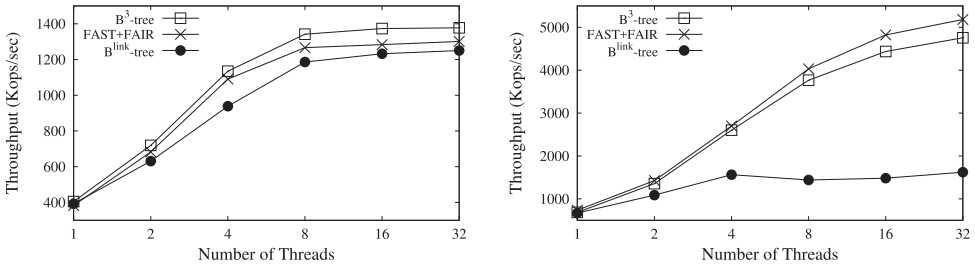
Fig. 10. Performance Comparison with Varying Latencies (Quartz Emulator).

In the experiments shown in Figure 10(a), we index 10 million random keys and measure the average insertion time while varying the write latency of PM. We set the read latency of PM to that of DRAM but we vary the write latency of PM. As we increase the write latency of PM, FAST and FAIR B-tree suffers from a larger number of cacheline flushes due to shift operations. Therefore, its insertion time increases at a faster rate than that of B^3-tree.

In the experiments shown in Figure 10(b), we increase both read and write latency. Similar to the results shown in Figure 10(a), B^3-tree consistently outperforms the other three indexes, but the performance gap between B^3-tree and FAST and FAIR B-tree is not as significant as the results shown in Figure 10(a) because FAST and FAIR B-tree benefits from prefetching and memory locality, thus it is less affected by the high read latency of PM. Figures 10(c) and 10(d) show the performance of exact match query and range query of persistent index. As we increase the read latency of PM, the performance gap between persistent indexes widens for both type of queries.

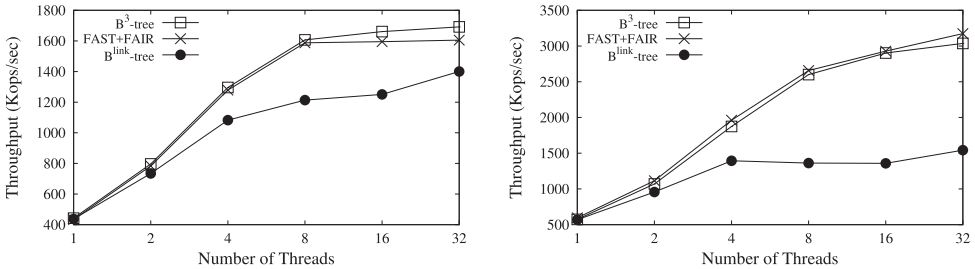
4.5 Concurrency

In the experiments shown in Figures 11 and 12, we evaluate the performance of multi-threaded versions of B^3-tree and FAST and FAIR B-tree. We do not evaluate the performance of wB+-tree in this experiments because wB+-tree is not designed for concurrent queries. However, wB+-tree can employ read/write locks or crabbing protocol [35] to handle concurrent queries. Instead of implementing a multi-threaded wB+-tree with crabbing protocol, we present the performance of *B-link tree* because it is known to outperform the crabbing protocol [35]. We note that our B-link tree is not designed to provide the failure-atomicity in Optane DCPM, i.e., it does not have additional consistency overhead, such as logging or metadata updates, unlike other persistent index. Therefore, the presented performance of multi-threaded B-link tree should be higher than the performance of wB+-tree that employs the crabbing protocol.



(a) Insertion Throughput (b) Search Throughput

Fig. 11. Throughputs with Varying Number of Threads (Optane DCPM).



(a) Insert 75%, Search 25% (b) Insert 25%, Search 75%

Fig. 12. Performance Comparison Using Mixed Workload (Optane DCPM).

In Figure 11, we measure the insertion and search throughput with varying the number of concurrent threads. We insert 25 million data into the index that already stored 25 million data. We note that both B^3-tree and FAST and FAIR B-tree enable lock-free search. Therefore, they show comparable concurrency level, but B^3-tree shows slightly higher insertion throughput than FAST and FAIR B-trees due to its superior insertion performance. However, for search throughput, B^3-tree is slightly outperformed by FAST and FAIR B-trees because of irregular cacheline accesses. Interestingly, B-link tree is outperformed by B^3-tree and FAST and FAIR B-tree. Although B-link tree does not have the consistency overhead and all three indexes require write locks to serialize write operations, B-link tree has to acquire read locks while traversing the index whereas the other two persistent index do not need read locks. As a result, read locks often block write transactions and insertion throughput of B-link tree degrades.

In the experiment shown in Figure 12, we measure the query processing throughput when the workload alternates between insert and search queries while varying the number of read and write queries. When 75% of queries are search, B^3-tree shows similar performance with FAST and FAIR B-tree. Note that although B^3-tree is slow for search queries, unless more than 75% queries are search queries, B^3-tree outperforms FAST and FAIR B-tree in terms of overall query processing throughput. If read queries account for less than 75%, B^3-tree shows higher throughput than FAST and FAIR B-tree.

4.6 Range Query

Hash indexes are known to perform faster than B-tree variants for exact match queries, and several hash-based indexes have been proposed for persistent memory [27, 44]. However, what makes B-tree stand out from other hash-based indexes is that B-tree variants allow range queries while hash-based indexes do not. Therefore, although hash-based indexes are faster than B+-tree variants

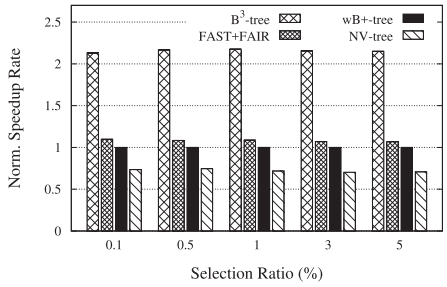


Fig. 13. Range-Query Speedup Rate (Optane DCPM).

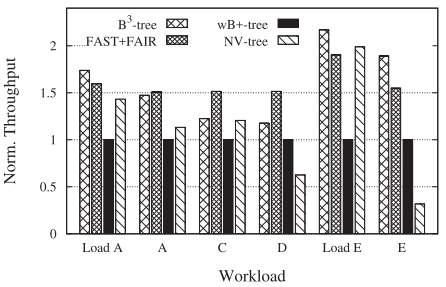


Fig. 14. YCSB Throughput (Optane DCPM).

for exact match queries, hash-based indexes are mostly used as a supplementary index of B-tree index.

In the experiments shown in Figure 13, we measure the range query performance of persistent indexes and show their relative performance improvement over wB+-tree with various selection ratios. The selection ratio is the proportion of selected keys to the number of indexed keys. I.e., if a query selects five keys out of a hundred key-value pairs in the index, its selection ratio is 5%. For the experiments, we index 10 million key-values in each persistent index and we submit range queries that select different numbers of records. Overall, B^3-tree is up to 2x, 2.15x, and 3x faster than FAST and FAIR B-tree, wB+-tree and NV-tree, respectively, in terms of query response time. It is noteworthy that B^3-tree shows worse search performance than FAST and FAIR B-tree for exact match queries. However, B^3-tree outperforms FAST and FAIR B-tree for range queries. This is because the page size of B^3-tree is much larger than FAST and FAIR B-tree (512 Bytes) and wB+-tree (1 KBytes). That is, B^3-tree benefits from sequential data access and prefetching for range queries since our range query implementation performs linear scanning of BST nodes instead of performing binary search.

4.7 YCSB Results

In the experiments shown in Figure 14, we measure the performance of persistent indexes using YCSB, which is widely used to evaluate the performance of key-value stores. In the experiments shown in Figure 14, we run six YCSB core workloads in Zipfian distribution and measure the throughput of persistent indexes. For ease of readability, we show the throughput normalized to wB+-tree.

`Load A` and `Load E` are write-only workloads. Therefore, B^3-tree outperforms FAST and FAIR B-tree and wB+-tree. This result is consistent with the microbenchmark results that we presented above. For read-intensive workloads (`Workload C` and `Workload D`), FAST and FAIR B-tree shows higher throughput because binary search of B^3-tree results in more cacheline accesses. For `Workload A`, which performs 50% reads and 50% writes, both FAST and FAIR B+-tree and B^3-tree show comparable performance. In `Workload E`, clients submit range queries instead of exact match queries. In this range query-only workload, B^3-tree benefits from its large page size and it shows 89% and 22% higher throughput than wB+-tree and FAST and FAIR B+-tree, respectively.

4.8 Bit Flips

In this experiments, we measure the number of flipped bits to evaluate the wearing issues of persistent memory. Figure 15 shows the number of flipped bits while varying the number of indexed keys. Overall, as the tree size grows, each write transaction is likely to change only leaf nodes

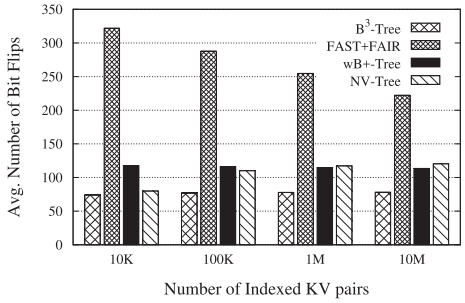


Fig. 15. Number of Flipped Bits with Varying Number of Indexed Key-Value pairs.

without rebalancing the whole tree. Therefore, the number of bit flips decreases as we insert more data.

B^3-tree consistently outperforms FAST and FAIR B-tree, wB+-tree and NV-tree because it benefits from its binary tree structure, which requires only a few number of bits to be flipped to persist write transaction. However, FAST and FAIR B-tree shifts a large number of data, which results in a significantly higher number of bit flips. Although wB+-tree outperforms FAST and FAIR B-tree, it has to update its metadata (a bitmap and a sorted indirection array) per transaction, thus it flips more bits than B^3-tree. The number of flipped bits in NV-tree increases as the index size grows because of the reconstruction of internal pages and parent leaf pages.

4.9 Fully Persistent Index vs. Recoverable Volatile Index

In the final set of experiments, we compare the performance of persistent indexes against FP-tree. While arguable, FP-tree and other persistent indexes are different in philosophy. FAST and FAIR B-tree and B^3-tree are fully persistent index that store all indexing components in PM, while FP-tree stores most of its indexing components in volatile DRAM, which need to be reconstructed upon system failures. We believe these indexes have different purposes and it is questionable whether comparing them directly is fair. If a system crashes, only raw data in leaf nodes of FP-tree (although they are sorted) will remain on persistent storage. With the persistent raw data, we can reconstruct any kind of volatile index such as binary search tree from scratch. Then, the question is whether we can classify those volatile index including legacy binary search as tree persistent index. We believe recoverable volatile index such as FP-tree serve different purposes from fully persistent indexes.

Since both DRAM and PM need to be installed on a fixed number of DIMM slots, the DRAM size becomes smaller as we install a larger number of PMs. With small DRAM and large PM, it will become problematic if we store all internal tree nodes on DRAM. In certain applications such as database systems, a query may access a large number of B+tree tables and a large number of queries may access different database tables, which could easily make working sets of processes exceed the memory capacity. FAST and FAIR B-tree and B^3-tree can be used for such database applications, but FP-tree cannot.

While the scope of application of FP-tree is rather limited compared to fully persistent index, FP-tree takes advantage of faster DRAM avoiding various limitations of newly developed PM technology. One of the critical limitations of PMDK is that a pool or a directory-based poolset must be entirely on a single socket. Therefore, a B^3-tree is confined to a single PM on a socket, threads running on another socket suffer from low NUMA bandwidth.

In the experiments shown in Figures 16 and 17, we compare the performance of single-threaded and multi-threaded FP-tree and fully persistent indexes. Figure 16 shows that the performance of

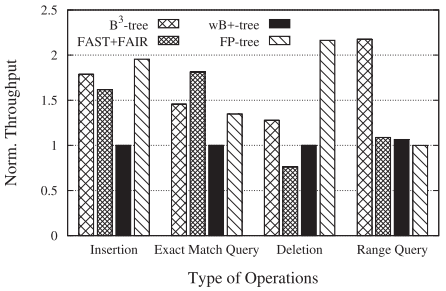


Fig. 16. Single-Threaded Indexing Performance Normalized to wB+-tree (Optane DCPM).

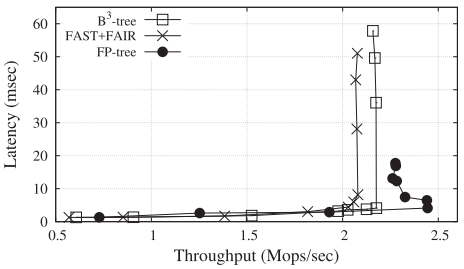


Fig. 17. Throughput vs. 99th Percentile Latency of 50:50 R/W Ratio (Optane DCPM).

indexes normalized to wB+-tree. FP-tree shows the highest insertion and deletion performance and outperforms B^3-tree because FP-tree eliminates the necessity of expensive logging and copy-on-writes. Also it does not need to call expensive cacheline flush instructions for tree rebalancing operations since its internal nodes are all volatile. As for the superior deletion performance, it is because FP-tree does not merge nor re-distribute key-values even if a page is underutilized. It simply updates the bitmap of each leaf page to remove a key-value record. As for the exact match search performance, FP-tree is outperformed by FAST and FAIR B-tree. Although FP-tree benefits from faster DRAM performance, its exact match query performance is similar to B^3-tree. This is because FP-tree does not sort key-value records in leaf pages. I.e., it has to read the entire leaf page for every query. In terms of range query performance, B^3-tree outperforms all other indexes because its 4x larger page size benefits from memory level parallelism and hardware prefetching.

For the experiments shown in Figure 17, we measure the 99th percentile query latency and query processing throughput while increasing the number of concurrent threads up to 32, which alternate insert and search queries. When the number of threads is smaller than the number of cores in a single socket, all three indexes scale well and the throughput of FP-tree peaks at 2.5 Mops/sec while B^3-tree peaks at 2.15 Mops/sec. However, if the number of threads exceeds the number of cores of a single socket, all three indexes suffer from low NUMA bandwidth and the 99th percentile tail latency increases rapidly. The 99th percentile latency of B^3-tree is about 2.5x higher than FP-tree. This is because FP-tree benefits from DRAM page migration for internal node accesses while the other indexes suffer from remote PM accesses. With the current Optane DCPM and PMDK design, pages are not allowed to be migrated to remote PM devices since app-direct mode PM uses memory-mapped files. We note that Lersch et al. [24] has reported that FP-tree scales well up to 47 threads in NUMA architecture. This is because Lersch et al. [24] stored leaf pages of FP-tree in multiple pools (i.e., multiple PM devices) while our implementation of FP-tree stores all leaf pages in a single pool. Since internal pages of FP-tree are volatile, FP-tree can store leaf pages in multiple pools and store 16-byte persistent pointers in volatile internal nodes in a non-failure atomic manner. However, in FAST and FAIR B-tree and B^3-tree, all tree pages must be stored in a single pool so that we can atomically update 8-byte offsets with a single store instruction. We note that this is one of the most critical limitations of fully persistent index.

5 CONCLUSION

In this work, we presented B^3-tree—a persistent index that combines byte-addressable BST and persistent B-tree to leverage the byte-addressability and persistency of PM. With the binary tree structure, B^3-tree avoids expensive shift operations and reduces the number of calls to expensive

cacheline flush and memory fence instructions. B^3-tree organizes the binary tree into hierarchical blocks so that it can take advantage of balanced tree height. To enable the failure-atomic updates to hierarchical blocks with fine-grained atomic write operations, we propose to use sibling pointers. Using the sibling pointer, we can rebalance the tree structure via 8-byte store instructions without expensive logging while preserving the invariants of index. Our performance study shows that B^3-tree outperforms the state-of-the-art persistent index—FAST and FAIR B-tree, wB+-tree and NV-tree in terms of insertion, delete, and range query performance at the cost of slightly low exact match query performance.

REFERENCES

[1] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. 2015. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 707–722.

[2] Trevor Brown. 2015. Reclaiming memory for lock-free data structures: There has to be better way. In *Proceedings of the 34th ACM Symposium on the Principles of Distributed Computing (PODC'15)*.

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

[4] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. Optimistic crash consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*.

[5] Nachshon Cohen and Erez Petrank. 2015. Efficient memory management for lock-free data structures with optimistic access. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'15)*.

[6] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, and D. Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*.

[7] Ru Fang, Hui-I Hsiao, Bin He, C. Mohan, and Yun Wang. 2011. High performance database logging using storage class memory. In *Proceedings of the 27th International Conference on Data Engineering (ICDE)*. 1221–1231.

[8] Thomas E. Harta, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. 2007. Performance of memory reclamation for lockless synchronization. *J. Parallel and Distrib. Comput.* 67 (2007), 1270–1285.

[9] Yiming Huai. 2008. Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects. *AAPPS Bulletin* 18, 6 (2008), 33–40.

[10] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. 2014. NVRAM-aware logging in transaction systems. *Proceedings of the VLDB Endowment* 8, 4 (2014).

[11] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable transient inconsistency in byte-addressable persistent b+-trees. In *Proceedings of the 16th USENIX Conference on File and Storage (FAST)*.

[12] Intel. 2018. Intel and Micron produce breakthrough memory technology. https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology.

[13] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-Ri Choi. 2019. SLM-DB: Single-level key-value store with persistent memory. In *Proceedings of the 17th USENIX Conference on File and Storage (FAST)*.

[14] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. 2010. FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

[15] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. 2016. NVWAL: Exploiting NVRAM in write-ahead logging. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[16] Wook-Hee Kim, Beomseok Nam, Dongil Park, and Youjip Won. 2014. Resolving journaling of journal anomaly in android I/O: Multi-version B-tree with lazy split. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*.

[17] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-performance transactions for persistent memories. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 399–411.

[18] HP Enterprise Lab. 2018. Quartz. https://github.com/HewlettPackard/quartz.

[19] Eunji Lee, Hyokyung Bahn, and Sam H. Noh. 2013. Unioning of the buffer cache and journaling layers with non-volatile memory. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*.

[20] 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 *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*.

[21] Wongun Lee, Keonwoo Lee, Hankeun Son, Wook-Hee Kim, Beomseok Nam, and Youjip Won. 2015. WALDIO: Eliminating the filesystem journaling in resolving the journaling of journal anomaly. In *Proceedings of the 2015 USENIX Anual Technical Conference*.

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

[23] Victor Leis, Alfons Kemper, and Thomas Neumann. 2014. Exploiting hardware transactional memory in main-memory databases. In *Proceedings of the 30th International Conference on Data Engineering (ICDE)*.

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

[25] Sparsh Mittal and Jeffrey S. Vetter. 2016. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel and Distributed Systems* 27, 5 (May 2016), 1537–1550.

[26] Sparsh Mittal, Jeffrey S. Vetter, and Dong Li. 2015. A survey of architectural approaches for managing embedded DRAM and non-volatile on-chip caches. *IEEE Transactions on Parallel and Distributed Systems* 26, 6 (June 2015), 1524–1537.

[27] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-optimized dynamic hashing for persistent memory. In *Proceedings of the 17th USENIX Conference on File and Storage (FAST)*.

[28] Gihwan Oh, Sangchul Kim, Sang-Won Lee, and Bongki Moon. 2015. SQLite optimization with phase change memory for mobile applications. *Proceedings of the VLDB Endowment (PVLDB)* 8, 12 (2015), 1454–1465.

[29] Jiaxin Ou, Jiwu Shu, and Youyou Lu. 2016. A high performance file system for non-volatile main memory. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys 16)*.

[30] 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 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

[31] Jun Rao and Kenneth A. Ross. 1999. Cache conscious indexing for decision-support in main memory. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*.

[32] Jun Rao and Keneeth A. Ross. 2000. Making B+-trees cache conscious in main memory. In *Proceedings of 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

[33] Jihye Seo, Wook-Hee Kim, Woongki Baek, Beomseok Nam, and Sam H. Noh. 2017. Failure-atomic slotted paging for persistent memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[34] Kai Shen, Stan Park, and Meng Zhu. 2014. Journaling of journal is (almost) free. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*.

[35] Abraham Silberschatz, Henry Korth, and S. Sudarshan. 2005. *Database Systems Concepts*. McGraw-Hill.

[36] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. 2007. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proceedings of IEEE 13th International Symposium on High Performance Computer Architecture (HPCA)*.

[37] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*.

[38] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. 2015. Quartz: A lightweight performance emulator for persistent memory software. In *Proceedings of the 15th Annual Middleware Conference (Middleware'15)*.

[39] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[40] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. 2018. Barrier-enabled io stack for flash storage. In *Proceedings of the 11th USENIX Conference on File and Storage (FAST)*.

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

[42] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, and Khai Leong Yong. 2015. NV-Tree: Reducing consistency const for NVM-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*.

[43] Yiying Zhang and Steven Swanson. 2015. A study of application performance with non-volatile main memory. In *Proceedings of the 31st International Conference on Massive Storage Systems (MSST)*.

[44] Pengfei Zuo and Yu Hua. 2017. A write-friendly hashing scheme for non-volatile memory systems. In *Proceedings of the 33rd International Conference on Massive Storage Systems and Technology (MSST)*.

Received March 2019; revised January 2020; accepted April 2020