PM-Rtree: A Highly-Efficient Crash-Consistent R-tree for Persistent Memory

Brandon Lavinsky
School of Engineering and Computer Science
Washington State University
Vancouver, WA, USA
brandon.lavinsky@wsu.edu

Xuechen Zhang
School of Engineering and Computer Science
Washington State University
Vancouver, WA, USA
xuechen.zhang@wsu.edu

ABSTRACT

Persistent R-trees are important data structures for indexing large-scale spatial datasets using persistent memory (e.g., Intel Optane DIMMs). Existing persistent R-trees (e.g., FBR-tree) suffer from four major issues. (1) Node updates cause unnecessary writes to persistent memory, leading to high latency. (2) The locking overhead is high under high thread concurrency. (3) They support a limited number of maximum bounding rectangles on each node. (4) The persistent overhead of managing its bitmaps in persistent memory is high for repeatedly cache line reflushing.

In this paper, we propose a novel data structure **P**ersistent **M**erged R-tree (PM-Rtree) for high-efficient insert, delete, and search operations for high-dimensional datasets using persistent memory. It is a partitioned data structure where its non-leaf nodes are stored in DRAM and leaf nodes are stored in persistent memory. In addition, we use an interleaved mapping approach. This approach maps contiguous data blocks in persistent memory to interleaved bits in bitmap groups in different cache lines to reduce cache line reflushes. Finally, PM-Rtree supports lock-free insertion using persistent multi-word compare and swap operations to eliminate locking overhead. Our experimental results show that PM-Rtree reduces the latency of insertion by up to 77.6% and 80% for the uniform and zipfian datasets respectively compared to the state-of-the-art persistent R-trees while maintaining crash consistency. It reduces the search time by 19.2% compared to FBR-tree. It achieves better scalability for both insertion and search up to 32 threads.

CCS CONCEPTS

• **Computer systems organization → Processors and memory architectures**; • **Information systems → Data structures**.

KEYWORDS

R-tree, Persistent Memory, Lock-free

ACM Reference Format:
Brandon Lavinsky and Xuechen Zhang. 2022. PM-Rtree: A Highly-Efficient Crash-Consistent R-tree for Persistent Memory. In *34th International Conference on Scientific and Statistical Database Management (SSDBM 2022), July 6–8, 2022, Copenhagen, Denmark*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3538712.3538713

1 INTRODUCTION

Spatial database systems have received considerable attention over the years because of their many applications in geographical information systems [7], computer-aided design [27], and computer vision [29]. A rectangle tree (R-Tree) is a popular data structure used for indexing spatial data in spatial database systems. The existing volatile R-trees [2, 12] designed for indexing spatial data, were proposed based on the assumption that computer memory is DRAM. Because it has become increasingly difficult to scale DRAM to higher density, we must seek more cost-effective solutions to extend memory capacity. Emerging byte-addressable persistent memory (PM) is widely adopted to build persistent in-memory databases for its low latency, large capacity, and data persistence. For example, the recently released Optane DC Persistent Memory (abbreviated as "Optane PM" in the rest of the paper) is capable of achieving comparable performance to DRAM [5].

Most of the existing R-tree data structures did not explore non-volatility of PM. Recently, FBR-tree was proposed to store and manage the spatial data in PM [6]. It can enforce crash consistency upon failures by setting a limit on the size of its metadata structure (e.g., bitmaps), so that failure-atomic instructions can be used in its insert and delete operations. However, our research shows that FBR-tree only achieves suboptimal performance for the following reasons.

First, FBR-tree insert and delete operations may incur high latency. PM write latency is 2.5X higher than that of DRAM [17, 28, 31]. The insert and delete operations of R-trees may need to recursively update R-tree nodes. FBR-tree places the whole data structures in PM, which may expose the long write latency to R-tree operations and result in a significant loss of performance.

Second, FBR-tree uses mutex locks to support concurrent insert and delete operations. However, our study shows that locking overhead accounts for up to 54% and 33.4% of the execution time of insert and delete operations, respectively, when thread concurrency is high.

Third, managing FBR-tree is expensive because of its large height. The size of the metadata structure in a node of FBR-tree is limited to 8 B. Thus, it only supports 55 maximum bounding rectangles (MBRs) per node. For large-scale datasets (e.g., PostalPoints [24]), the height of the FBR-tree is 5. Therefore, managing the FBR-tree requires updating 78.1% more nodes than R-trees supporting a large number of MBRs (e.g., 247+) per node.

Last but not least, managing the bitmaps in FBR-tree may cause cache line reflushes. A typical size of CPU cache line is 64 B [25]. The size of a bitmap is 8 B for the nodes of FBR-trees. When the bitmap is updated repeatedly during insertion, the same cache line should be flushed for persistence. The latency of cache line reflush is 7.5X higher than the latency of writes [5]. As a result, updating bitmaps of FBR-tree nodes causes degraded performance.

In the paper, we propose a new persistent R-Tree variant, named PM-Rtree. Its design emphasizes on efficiently reducing the number of writes to PM, alleviating locking overhead, and eliminating cache line reflushes. Specifically, first, PM-Rtree is a partitioned data structure. Its non-leaf nodes are stored in DRAM while its leaf nodes are stored in PM. Thus, PM-induced additional memory latencies can be effectively shielded. Second, PM-Rtree is a lock-free data structure. We design a new insert operation using the PMwCAS library [32] to support atomic updating of bitmaps in the nodes of PM-Rtree. Third, we replace a single bitmap with a bitmap group (named G-bitmap) on each node. Each bitmap in G-bitmap is stored in a CPU cache line and its maximum size is 8 B. Thus, they can be updated using atomic CPU instructions for data persistence. At any particular time point, only one bitmap in G-bitmap can be updated by using the PMwCAS-based concurrency control mechanism. Using this design, PM-Rtree supports a variable number of MBRs in each node for optimal performance. Finally, PM-Rtree uses an interleaved memory mapping from data blocks to their corresponding bits in bitmaps to avoid accessing the same CPU cache line repeatedly.

We analyze the space and time cost of PM-Rtree. We compare it with other state-of-the-art persistent R-trees on a real machine equipped with Intel Optane 3D-XPoint persistent memory. Our experimental results show that PM-Rtree outperforms FBR-tree by 84.1% and 19.2% for insertion and search. It achieves 77.8% better scalability than FBR-tree. In summary, we made the following contributions.

- We conduct empirical study on the performance of persistent R-tree indexes using Intel Optane PM. The results show that they may cause a large number of writes and repeated cache line reflushes in PM. Furthermore, mutex locking is a performance bottleneck under high thread concurrency.
- We design a new persistent R-tree index PM-Rtree. (1) It stores non-leaf nodes in DRAM to reduce PM writes. (2) It is a lock-free data structure, which uses PMwCAS to support atomic updating of bitmaps for insertion and deletion. (3) It uses an interleaved memory mapping for node metadata management to alleviate cache line reflushes.
- We implement a software prototype of PM-Rtree and evaluate it using real-world spatial datasets (e.g., FireStat and Postal). Our experimental results show that PM-Rtree significantly improves the insertion/deletion and search performance compared to the state-of-the-art FBR-tree.

The rest of the paper is organized as follows. Section 2 explains the performance issues of the existing persistent R-tree index. In Section 3 we introduce related work. Section 4 describes how to build PM-Rtree to minimize the number of writes to PM, provide lock-free thread management, and reduce cache line reflushes. Section 5 describes and analyzes experimental results. And Section 6 concludes the paper.

2 BACKGROUND AND MOTIVATION

2.1 Persistent R-Tree Index

R-trees are designed to index multi-dimensional data sets [12]. Ordinarily, R-trees are ephemeral data structures in DRAM indexing data stored on slow hard disks or flash disks. They have many variants (e.g., R*-tree [2], PR-tree [1], and NIR-tree [16]). Each of them uses its own approach to achieve node division/merging and rebalance when a point is inserted or deleted. Each R-tree node can contain at most M entries and with the exception of the root node, which contains at least m entries ($m \geq M$). The entry of leaf nodes contains a Minimum Bounding Rectangle (MBR) and a data object (or its reference). The entry of non-leaf nodes contains an MBR which fully contains all downward data objects and a pointer to its child node. R-trees are height-balanced because all of their leaf nodes have the same height $O(log_m n)$, where n is the total number of objects.

Persistent R-tree index [6] supports the same basic functionality as R-tree, which divides a 2D/3D space into a set of possibly overlapped rectangles. In FBR-tree, each node consists of both metadata and data spaces. The data space consists of a set of MBRs and pointers to child nodes. In the metadata space, it uses bits to indicates whether its corresponding MBR and its pointer in the node are valid. We typically use a bitmap to manage these bits. When a bit is 0, its corresponding MBR and pointer in the node is free and can be allocated for serving insert operations.

A persistent R-tree needs to enforce crash-consistency so that both data and metadata in PM can be used to rebuild a consistent R-tree upon failure recovery. For example, FBR-tree implemented failure-atomic insert and delete operations based on the observation that a partially updated MBR does not affect the correctness of FBR-tree after failure recovery. FBR-tree also atomically updates data (e.g., node entries) and metadata (e.g., version numbers and bitmaps) for in-place split without triggering the overhead of copy-on-write or logging. In this paper, we have the same assumption as FBR-tree. We address many performance issues discovered in our empirical study without compromising invariants of the FBR-tree index.

2.2 Empirical Study on Optane PM

We conducted an in-depth performance review of existing persistent R-trees (i.e., FBR-tree) and observed some interesting findings. In our evaluation, each write operation is followed by $clwb$ and $mfence$ instructions, ensuring that the written data has reached Optane PM. We use a synthetic dataset which is comprised of 3 dimensional rectangles. The position value of rectangles are randomly generated within a pre-defined space (details of the experimental configuration are given in Section 5).

(1) *The insert operations cause a large number of writes in PM.* In Figure 1, we measure the total number of PM writes as we increase the number of insert operations from 50K to 1000K. We observe that the number of PM writes is increased from 2122219 to 5074735. Each insert operation causes 4.7 PM writes on average. This is

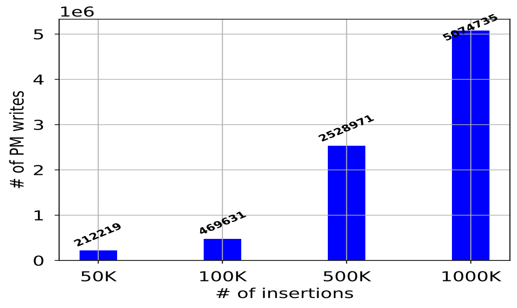


Figure 1: The number of PM writes as we increase the number of insertions from 50K to 1000K.

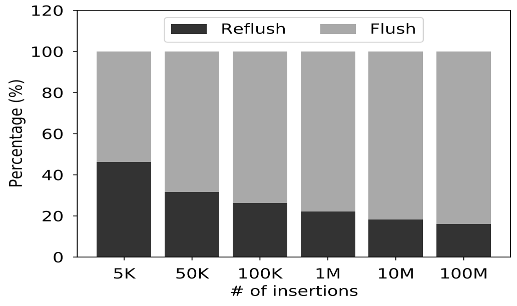


Figure 2: The ratio of the number of cache line reflush.

because FBR-tree limits the number of MBR-pointer entries to 55, which triggers a large number of node-splitting writes to PM.

(2) *The insert operations incur repeated cache line reflushes in PM.* Given a sequence of cache lines (1, 2, 3, 4, 1) that are flushed consecutively, the reflush distance of cache line 1 is 3. In this paper, we assume a cache line reflush occurs when its reflush distance is no larger than 1. Otherwise, a regular flush occurs. To study the number of cache line reflushes during insertion, we run FBR-tree and increase the number of insert operations from 5K to 100M. The ratio of reflush and flush operations are shown in Figure 2. When the number of insertion is 5K, the ratio of reflush is 53.6%. It is decreased to 16.2% as we have more insertions. This is because the positions of MBRs are randomly generated. Having a larger number of insertions reduces the possibility of repeatedly inserting KVs to the same nodes. The results show that flush operations used in implementing persistent R-trees may lead to poor performance of insert and delete because the latency of cache line reflush can be 7.5X higher than the latency of writes [5].

(3) *Mutex locking/unlocking is a performance bottleneck under high thread concurrency.* The existing persistent R-trees use mutex locks for thread synchronization. We study the scalability of FBR-tree in Figure 3(a). We execute 5M insertions with different number of threads in the experiments. The execution time speedup is decreased to 0.46 when the number of threads is 32. We observe that performance bottleneck is caused by mutex locks. As shown in Figure 3(b), the ratio of execution time spent on locking is increased from 0.1% to 54% as the number of threads is increased from 2 to 32.

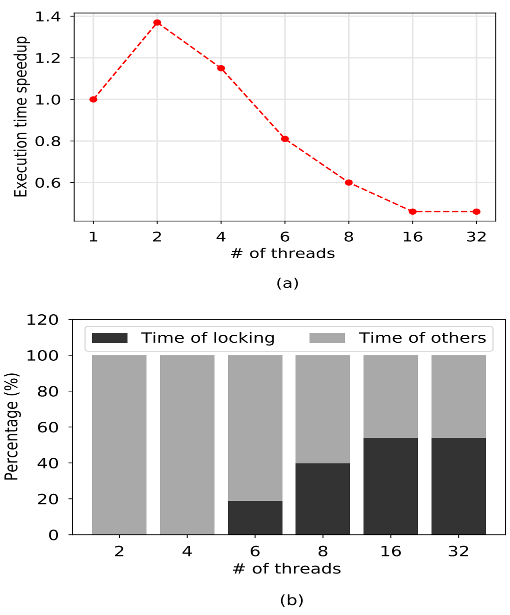


Figure 3: (a) Execution time speedup of FBR-tree and (b) time breakdown with mutex locks.

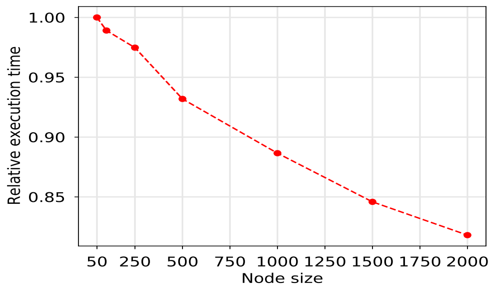


Figure 4: The Impact of the number of MBR-pointer pairs per node on execution time.

(4) *Performance of persistent R-trees is affected by its node size.* Persistent R-trees support a limited number of MBR-pointer entries per node (called node size in the paper), affecting its height. For failure atomicity, the node size of FBR-tree is 55. Here, we study the impact of node size on its execution time using a regular persistent R-tree. The relative execution time of 1M insertions is shown in Figure 4. The execution time is decreased by 18.2% while the node size is increased from 50 to 2000. The results show that the FBR-tree achieves failure atomicity at the price of insertion performance because of its increased tree height and the number of node splitting and merging for insertion and deletion respectively.

In summary, although the existing persistent R-trees can enforce crash consistency, they only achieve suboptimal performance because they did not fully consider the unique characteristics of PM.

Name	*Failure atomicity*	*Lock-free insertion*	*Lock-free search*	*Reducing reflushing*	*Partitioned*
Flash R-tree [33]	No	No	No	No	No
Hybrid R-tree [11]	No	No	No	No	Yes
FBR-tree [6]	Yes	No	Yes	No	No
PM-Rtree	Yes	Yes	Yes	Yes	Yes

Table 1: Comparison of PM-Rtree with existing persistent R-trees.

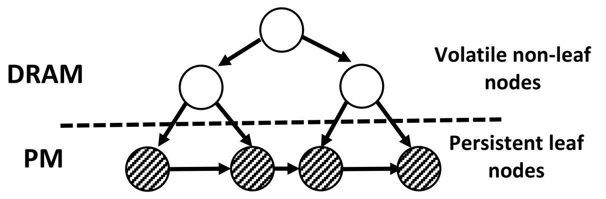


Figure 5: An illustration of PM-Rtree.

3 RELATED WORK

To extend memory space using PM for in-memory database applications, a wide range of persistent data structures (e.g., R-tree, B-tree, Octree, and graph) have been developed to address performance and consistency issues caused by PM. The work closely related to PM-Rtree is discussed below.

3.1 Persistent R-trees

Flash R-tree was designed to reduce write amplification in flash storage [33] with Rtree-aware FTL mapping algorithm. Hybrid R-tree was proposed to store R-trees in both flash and PM [11]. It is a partitioned data structure in which high-frequently accessed MBRs are stored in PM and the rest is stored in flash storage. FBR-tree was the first R-tree solution that provides failure atomicity and lock-free search for performance optimization. It limits the number of MBR entries per node to 55 so that atomic instructions can be used when flushing data to PM. Furthermore, it uses version numbers for thread synchronization. Compared to FBR-tree, we implement lock-free insertion using the PMwCAS library to further reduce locking overhead. We use a partitioned data layout with which non-leaf nodes are stored in DRAM so that the number of PM writes is significantly reduced. We also use interleaved mapping of bitmaps to reduce the number of cache line reflushes. Table 1 summarizes the unique characteristics of PM-Rtree compared to the existing persistent R-trees.

3.2 Other Works in Context

Reducing locking overhead: Lock-free data structures have been studied for PM. FBR-tree [6] uses version numbers to implement lock-free search. It optimistically accesses each node of R-trees and rolls back when a data inconsistency is detected later. Wang et al. implemented doubly-linked skip list [26] and Bw-tree [18] using PMwCAS for lock-free indexing in PM [32]. Although PMwCAS has been used elsewhere, PM-Rtree is the first to apply it in the implementation of persistent R-trees synergistically combing with other optimizations.

Partitioned persistent data structures: Partitioned data structures have been well studied to optimize data layout of B-trees, Octrees, and graphs. BPTree stores non-leaf nodes in DRAM and leaf nodes in PM [14]. It has buffer nodes on top of leaf nodes for batch insertion. DPTree consists of a buffer tree in DRAM and a base tree in PM [34]. Writes are first served by the buffer tree and then merged with the base tree to reduce write amplification in PM. FlatStore can be used to implement partitioned B+-tree [5]. It consists of volatile indexes in DRAM and compacted logs in PM. The latter is designed to serve small writes in batches. PM-Octree and DPM-Octree were designed as persistent octrees to support large-scale meshing [22, 23]. They are partitioned data structures because their hot octants are stored in DRAM to reduce the number of writes to PM. NVGraph also partitions a graph data structure so that popular vertices and edges are stored in DRAM for better scalability [19, 20]. Similar to these data structures, PM-Rtree has a partitioned data layout to hide write latency of PM.

Reducing cache line reflushes: wB$^+$-Tree uses a small bitmap to reduce the number of PM writes for updating index entries, thus reducing the number of cache line reflushes. FAST&FAIR uses failure-atomic shift and failure-atomic in-place rebalance to reduce the number of flushes [15]. The buffer tree of DPTree serves small writes sequentially in write-ahead logs and merges them with the base tree in batches to reduce flush operations [34]. LB$^+$ is designed to pack metadata and data in one CPU cache line so that they can be updated in one flush rather than two separated flushes [21]. FlatStore [5] uses a compacted log structure to reduce the number of writes in PM. Most recently, NValloc [10] uses an interleaved tcache layout to map contiguous memory blocks to interleaved sub-tcaches in different cache lines. Thus, the issue of repeated cache line reflushes can be alleviated. PM-Rtree uses an interleaved mapping scheme in bitmaps to reduce the number of cache line reflushes.

4 DESIGN OF PM-RTREE

This paper proposes PM-Rtree, a persistent R-tree with three key design principles: minimizing the number of writes to PM, lock-free thread management, and reducing cache line reflushes.

4.1 Overview

As shown in Figure 5, PM-Rtree is a hybrid data structure. Its non-leaf nodes are stored in DRAM to hide PM-induced latency. Its leaf nodes are stored in PM. We can rebuild all of the volatile non-leaf nodes using persistent leaf nodes upon failure recovery. All leaf nodes are connected by pointers for efficient failure recovery.

Node structure. The existing persistent R-trees limit the number of MBR entries in a node. For example, FBR-tree uses $mfence$ and $clflush$ CPU instructions to atomically update metadata of nodes. It can only support 55 entries for failure atomicity because it needs to limit the size of metadata to 8 B. Our study shows that for large-scale datasets this limitation can result in 11.9% and 32.1% performance loss for insert and search operations respectively.

In this paper, we design a node structure to support a various number of entries in a node. Figure 6 shows the node structure of PM-Rtree. Its node consists of metadata and data areas. The data

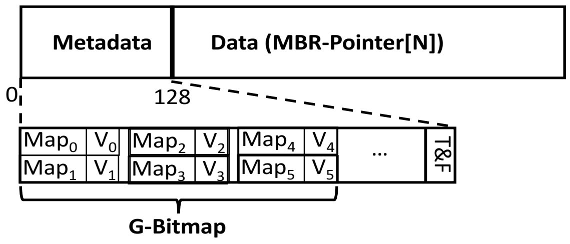


Figure 6: Node structure with G-bitmap.

area stores an array of MBR-and-pointer pairs. The MBRs contain all downward nodes and pointers pointing to its respective child nodes in DRAM or PM. In the metadata area, T uses the last 1 bit to indicate the node type (i.e., non-leaf or leaf nodes). F uses the second to the last bit to indicate the fullness of the node. Besides, it stores a bitmap group (named G-bitmap). The G-bitmap is a set of map-and-$version$ (MV) pairs. Map of an MV pair is a bitmap which has 56 bits and indicates whether an entry (MBR-and-pointer pair) is allocated (1: allocated; 0: free). $Version(V)$ is a version number (8 bits) used to implement lock-free search. The size of each MV pair is 8 B so that it can be atomically updated in persistent memory using $mfence$ and $cflush$. Each map can manage 56 entries.

The MV pairs are mapped to different cache lines. We use 2 cache lines as an example for illustration in Figure 6. Because a typical size of CPU cache line is 64 B [25], the size of its metadata area is 128 B. In the example, the metadata area includes 6 MV pairs. And its last two bits are used for type and fullness of the node. For the 6 MV pairs, they are mapped to two cache lines. Specifically, $< Map_0, V_0 >$, $< Map_2, V_2 >$, and $< Map_4, V_4 >$ are stored in the first 64 B and mapped to cache line 0. $< Map_1, V_1 >$, $< Map_3, V_3 >$, and $< Map_5, V_5 >$ are stored in the second 64 B and mapped to cache line 1. Entries in the data area are contiguously allocated but mapped to interleaved MV pairs using the algorithm described in Section 4.2 to reduce repeated cache line reflushes. The node structure in Figure 6 can support up to 336 entries in the data area.

After updating an entry (MBR-and-pointer pair), we will change its bit and write the incremented version number in its corresponding MV pair together followed by $mfence$ and $cflush$. We use its sequence number to index each MV pair. During insertion, the bits in an MV pair are contiguously allocated.

Proof of correctness: For correctness, all the search operations for non-empty entries should scan the bitmaps from the left-most MV pairs. To obtain the latest version number, if there are empty entries in the node, the MV pair having least number of 0s and largest sequence number has the latest version number. If a node is full, its latest version number is stored in the MV pair who has the largest sequence number. Neither delete or search operations update version number (explained in FBR-tree).

For each insertion, we only atomically update a single MV pair using $mfence$ and $cflush$, and no two MV pairs on a persistent leaf node can be updated simultaneously because we use PMwCAS to provide thread-level concurrency control. Therefore, PM-Rtree can enforce the same level of crash consistency as FBR-tree on persistent leaf nodes.

4.2 Interleaved Mapping of Bitmaps

The existing persistent R-trees use bitmaps to manage the data area. They sequentially map bits to entries of MBR-pointer arrays. As shown in Figure 7(a), MBR-pointer[0], MBR-pointer[1], and MBR-pointer[2] are sequentially mapped to $B0$, $B1$, and $B2$ in the bitmaps as those used in FBR-tree. After writing each bit in the bitmap, the metadata area needs to be flushed for failure atomicity, leading to repeatedly reflushing cache line 0 in persistent memory.

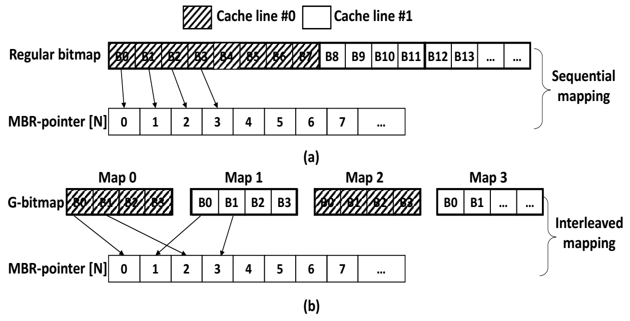


Figure 7: Sequential and interleaved mapping from bitmap to MBR-pointer entries.

In this paper, we use an interleaved mapping approach to reduce cache line reflushing. Specifically, we partition G-bitmap into groups, each of which occupies a single CPU cache line. The size of each group is capped by the cache line size (64 B). Then, consecutive MBR-pointer entries are mapped to different groups in different cache lines. Algorithm 1 describes the algorithm for interleaved mapping of bitmaps. PM-Rtree selects a bitmap and a bit in G-bitmap (line #9-10). Then it sets the data area, updates the bitmap and version number (line #11-13). It writes the data area to PM (line #15). Then it atomically writes the bitmap and version number in the metadata area to PM (line #16). If the current selected bitmap group is full, it identifies a new bitmap group to set (line #23-25).

We use Figure 7(b) to illustrate this algorithm. Group 1 includes Map_0 and Map_2, which are stored in cache line 0 and Group 2 includes Map_1 and Map_3, which are stored in cache line 1. MBR-pointer[0] and MBR-pointer[2] are mapped to $B0$ and $B1$ in Map_0 respectively. MBR-pointer[1] and MBR-pointer[3] are mapped to $B0$ and $B1$ in Map_1 respectively. There will be no cache line reflush when MBR-pointer[0] and MBR-pointer[1] are consecutively allocated because their corresponding bits are located in different cache lines. If both Map_0 and Map_1 are full, Map_2 and Map_3 are allocated in an interleaved manner for insertions.

4.3 Basic Operations of PM-Rtree

Lock-free updates using PMwCAS. The existing persistent R-trees use mutex locks for thread synchronization. The code snippet for updating node metadata of R-trees using mutex locks is shown in Figure 8(a). In PM-Rtree, we replace the mutex locks using the persistent_cas() function provided in the PMwCAS library [32]. The code snippet using PMwCAS is shown in Figure 8(b). We apply this lock-free code in all the operations of PM-Rtree requiring modification of node metadata.

Algorithm 1 Interleaved Mapping of Bitmaps

```
Function SetInterleavedBitmap(targetNode, newEntry) :
Node n = targetNode;
Entry data = newEntry;
startMap = 0;
checkBit = 0;
foundSlot = false;
if (n not full) then
    while (foundSlot == false) do
        for (i = startMap; i <= startMap + 1; i++) do
            if n−> Meta.map[i].Bit(checkBit) not set then
                n−> Meta.map[i].branch[checkBit] = data;
                n−> Meta.map[i].SetBit(checkBit);
                n−> Meta.map[i].UpdateVersion();
                if n is leafNode then
                    persist(n, data);
                    persist(n, meta);
                end if
                foundSlot = true;
                break;
            end if
        end for
        checkBit++;
        if (checkBit == 56) then
            checkBit = 0;
            startMap = startMap + 2;
        end if
    end while
end if
```

```
1. lock(n);
2. persistent (n->metadata);
3. CLWB(n->metadata);
4. SFENCE;
5. unlock(n);
```

(a) With mutex lock

```
1. old_meta = new(n->metadata);
2. new_meta = update(old_meta);
3. persistent_cas (n->metadata,
                   old_meta,
                   new_meta);
```

(b) Lock-free

Figure 8: Lock-free using PMwCAS.

Lock-free insertion. For inserting a new object, we need to recursively traverse PM-Rtree from root node to leaf node to find a candidate child node. We use least enlargement algorithm [12] if the MBR of the chosen child node is not completely overlapped with the new object to insert. Similar to the insertion of FBR-tree, we do not use logging in the node enlargement process because subsequent queries will still find and visit the desired child node if the node is only partially updated [6] upon failures.

Algorithm 2 PM-Rtree Insertion

```
1:  Function Insert(node, parent, data) :
2:  if (node → Meta.nodeType() == leaf) then
3:      if (node → Meta.isFull() == false) then
4:          Node prev = null;
5:          prev = node;
6:          SetInterleavedBitmap(node, data);
7:          persistent_cas(&(node → Meta), prev → Meta,
                                            node → Meta);
8:          splitNode = null;
9:          return splitNode;
10:     else
11:         splitNode = SplitNode(node, parent, data);
12:         return splitNode;
13:     end if
14: else
15:     idx = chooseChildPosition(node, data);
16:     Node child = node → child[idx];
17:     child.mbr = calculateNewMbr(child.mbr, data);
18:     splitNode = Insert(child, node, data);
19:     if (splitNode ≠ null) then
20:         siblingNode.mbr = splitNode.mbr;
21:         if (node → Meta.isFull() == true) then
22:             splitNode = SplitNode(n, splitNode, parent);
23:         else
24:             Data indexData;
25:             indexData.mbr = siblingNode.mbr;
26:             indexData.ptr = &siblingNode;
27:             Node prev = null;
28:             prev = node;
29:             SetInterleavedBitmap(node, indexData);
30:             regular_cas(&(node → Meta),
                            prev → Meta,
                            node → Meta);
31:             splitNode = null;
32:         end if
33:         return splitNode;
34:     else
35:         splitNode = null;
36:         return splitNode;
37:     end if
38: end if
```

Algorithm 2 describes the insertion algorithm of PM-Rtree. The algorithm handles insertion in leaf nodes (line #2-13) and non-leaf nodes (#15-37) separately. Persistent_cas() is implemented using PMwCAS [32] and regular_cas() is implemented using MwCAS [13]. For leaf nodes, if they are not full, we will insert the MBR and set the metadata directly (line #4-8). If the insertion causes an overflow, we will use the in-place split algorithm [6] to add a new entry to the split nodes (line #11). Assume that we need to in-place split node x to add a new node y and their parent node is node p. (1) We need to allocate node y, set the version number of x to 0, and copy half of the entries in x to y. (2) We add the address and MBR of y

to p. (3) We atomically update the version number and G-bitmap of p. (4) We update MBR of x in p to reflect the new space size for the remaining MBRs in x. (5) We update the version number and G-bitmap of x atomically. For non-leaf nodes, we recursively update their MBR entries until reaching leaf nodes.

Lock-free deletion. For deleting an existing object, we need to merge two nodes when underflows happen. Specifically, assume that we need to in-place merge node x and y, where x is the underflow node, y is the sibling node of x, and p is the parent node of node x and y. We copy the MBR entries of y to x. Then, we need to atomically update the G-bitmap and version number of x. Next, we need to update MBR of p to include all the MBRs merged from y. In the end, we need to atomically update the G-bitmap and version number of p to invalidate the entry of y in p. We do not show the deletion algorithm in the paper because of its similarity to the insertion algorithm and space limit.

Search. We revised the search algorithm used in FBR-tree so that it works with the interleaved mapping of bitmaps as discussed in Algorithm 1.

Failure recovery. We can rebuild PM-Rtree by rebuilding non-leaf nodes in DRAM while traversing the linked list of leaf nodes in PM. To address the issue of memory leakage, we adopt the chunk-based allocation strategy [4] to service the allocation/deallocation requests in PM. Any leaf nodes that are not reachable during the recovery process are reclaimed and used for future memory allocations in the memory pool.

5 EVALUATION

We conduct an extensive performance study for PM-Rtree to experimentally answer the following questions.

- Is PM-Rtree effective and scalable for synthetic and real datasets with diverse query patterns?
- What is the impact of each optimization used in PM-Rtree on the reduction of query execution time?
- Is it effective on the future eADR platform?

5.1 Experimental Setup

Experimental platform. We run the experiments on a Linux server with a 16-core Intel Xeon Silver 4215 CPU, 32 GB DRAM and 2 Intel Optane DCPMMs (128 GB per DIMM). The Optane PM is mounted with the Ext4-DAX file system and configured in App Direct mode. To avoid the NUMA effects, we use the numactl utility to bind every thread to one core in the first socket. All source codes are compiled with g++7.5 with -O3 optimization. All presented measurements represent arithmetic means of three runs.

Data Sets. We use three datasets in the evaluation. (1) *Random:* This is a synthetic 3D dataset. It consists of 20 M randomly generated rectangles defined by its minimum points (min-x, min-y, and min-z) and maximum points (max-x, max-y, and max-z). Min-x/min-y/min-z are random numbers between 0 and 1. Max-x/max-y/max-z are random numbers between 1 and 2. (2) *FireStat:* This is a dataset consisting of 329,448 latitude & longitude coordinates of wildfire ignitions points in the United States. It is published by USDA Forest Service [30]. (3) *Postal:* This is a 2D dataset of postal code boundaries. It consists of over 19 M boundary coordinate locations for postal code areas. It is published by SpatialHadoop [24].

Target comparisons. (1) *R-tree*: This is an R-tree designed for DRAM [2]. It does not enforce crash consistency. Therefore, it is not recoverable upon failures. (2) *PMDK R-tree*: This is an R-tree designed using the transactional model to enforce crash consistency. We implement the insert and delete operations using the transactional model provided by Intel PMDK [8]. Failure recovery is supported by rollback using logs. (3) *FBR-tree*: This is the state-of-the-art persistent R-tree [6]. It provides failure atomicity by limiting the metadata size to 8 B and updating the node metadata using $mfence$ and $clfush$. The node size is 55, which is the default setting of FBR-tree as described in its paper. (4) PM-Rtree: This is a persistent R-tree optimized for PM using techniques discussed in Section 4 while still enforcing crash consistency. By default, G-bitmap is stored in two CPU cache lines with a support of up to 784 MBR-pointer pairs per node.

5.2 Overall Performance

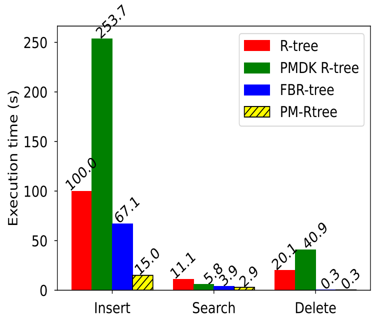
	Random	FireStat	Postal
Insert	5 M	330 K	10 M
Search	100	1 K	10 M
Delete	1 K	200 K	100 K

Table 2: Experimental setting.

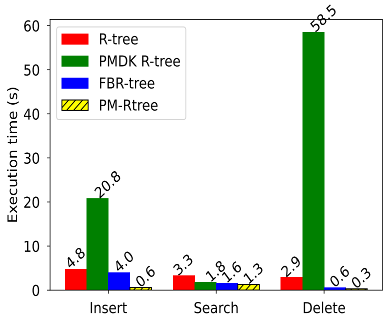
In this section, we study the overall performance of PM-Rtree compared to other persistent R-tree solutions. We measure the execution of the insert, search, and delete operations with three datasets including Random, FireStat, and Postal. The number of insert, search, and delete operations are listed in Table 2 for each dataset. We use 16 threads in the experiments. The results are shown in Figure 9. We have three observations.

First, PM-Rtree improves the performance of insert by up to 13.3X, 64.8X, and 9.8X compared to R-tree, PMDK R-tree, and FBR-tree, respectively. Using R-tree designed for DRAM directly in PM results in 88.4% more execution time than PM-Rtree because of its locking overhead. PMDK R-tree has the worst performance because it uses log-based transactional model for failure atomicity. The logging overhead dominates the execution time during insertions. The performance of FBR-tree is degraded compared to PM-Rtree because it incurs 63.4% more writes in PM, locking overhead, and extra latency caused by cache line reflushes when allocating bitmaps. Similar to insert, PM-Rtree improves the performance of delete by up to 67X, 195X, and 3.2X compared to R-tree, PMDK R-tree, and FBR-tree, respectively.

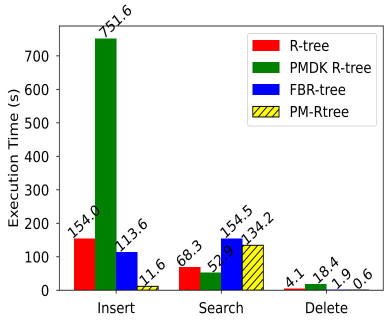
Second, the search performance of PM-Rtree is comparable to other persistent R-trees. The write latency of PM is 2.5X longer than DRAM, while its read latency is comparable to that of DRAM. Therefore, PM-Rtree is heavily optimized for the insert and delete operations. The new design also benefits the search operations. This is because we use a partitioned data layout. The search operations will reduce the number of random reads in PM, thus leading to less read amplification [3, 21] and improved performance for the Random and FireStat datasets.



(a) Random



(b) FireStat



(c) Postal

Figure 9: Overall performance with insert, search, and delete operations.

Third, the performance improvement of PM-Rtree is varied for different datasets. This is because the sequentiality of data directly impacts the time in which it takes to select a location for a new entry in the list of MBRs within non-leaf nodes. For example, the Postal dataset is highly squential and when compared to the Random dataset on time spent selecting the location of new entries, the Postal dataset spent on average 76.2% more time per selection. Similarly, the FireStat dataset, which is less squential than the Postal dataset, spent 66.5% more time per selection compared to Random dataset but 43.6% less time compared to the Postal dataset. This discrepancy in slection time is due to the fact that, new random data entries usually fit into existing MBRs without the need to be enlarged, whereas sequential data triggers more MBR enlargements adding more computation time.

Because we have shown that PM-Rtree and FBR-tree generally perform better than R-tree and PMDK R-tree, we will only compare PM-Rtree to FBR-tree in the following experiments.

5.3 Concurrent Insert and Search

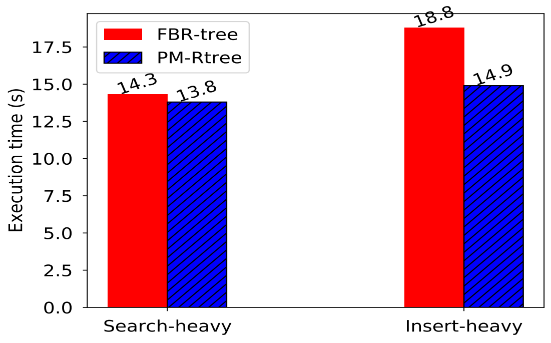
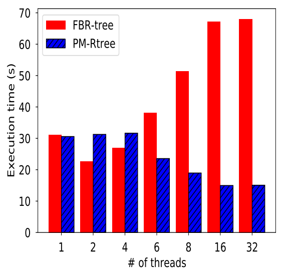


Figure 10: Performance with concurrent insert and search operations.

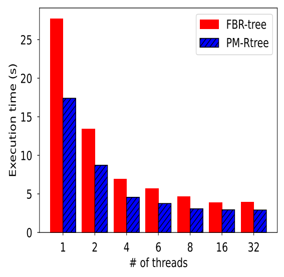
In this section, we evaluate the performance of PM-Rtree with concurrent insert and search operations. We set up two types of mixed workloads. The first one is *search-heavy*. For this workload, we first perform 5 M insert operations. Then, we perform 10 K insert and 300 K search operations concurrently using 4 threads. The second workload is *insert-heavy*. For this workload, we first perform 1 M insert operations. Then, we perform 4 M insert and 1 K search operations concurrently using 4 threads. In the experiments, we use the Random dataset and measure the execution time of the two workloads. The results are shown in Figure 10.

We have three observations from the figure. First, PM-Rtree works effectively for both search-heavy and insert-heavy workloads. It reduces the execution time by 3.5% and 20.7% for the search-heavy and insert-heavy workloads respectively compared to FBR-tree. Second, the optimizations of PM-Rtree benefits more for the insert-heavy workload because it reduces the number of writes in PM and the number of cache line reflushes, which are caused by the insert and delete operations. Third, it improves the performance of the search-heavy workload because the lock-free insert operations reduce locking overhead.

5.4 Scalability



(a) Insert



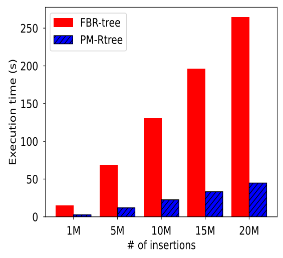
(b) Search

Figure 11: Performance when scaling up the number of threads at a fixed dataset size.

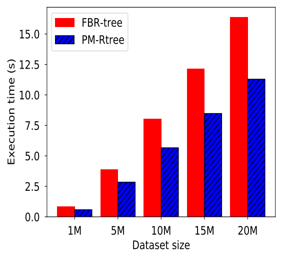
In this section, we study the scalability of PM-Rtree. In the experiment, we increase the number of threads from 1 to 32 at a fixed dataset size of 5 M. We execute 5 M insert and 100 search operations on the Random dataset. The results are shown in Figure 11.

We have the following observations. First, as shown in Figure 11(a), when the number of insertion threads is smaller than 4, FBR-tree performs slightly better than PM-Rtree. This is because the locking overhead is not the bottleneck in this scenario and PMwCAS incurs 13% overhead. Second, as the number of threads is increased from 6 to 32, PM-Rtree reduces the execution time by 38.1%, 63.1%, 77.7%, and 77.8%, respectively. It shows that PM-Rtree achieves significantly improved scalability when the thread concurrency is high and mutex locking overhead dominates the execution time during insertions. The results suggest we need to selectively apply PMwCAS to reduce overhead. We plan to implement it in the future work. Third, as shown in Figure 11(b), both PM-Rtree and FBR-tree achieve a good scalability for search because they do not use locks in the search operations.

5.5 Impact of Dataset Size



(a) Insert



(b) Search

Figure 12: Performance when scaling up the data set size with 16 threads.

Here, we study the impact of dataset size. In the experiments, we use 16 threads. We first increase the number of insert operations from 1 M to 20 M using the Random dataset. We can observe that PM-Rtree outperforms FBR-tree by up to 5.9X from Figure 12(a). The improvement ratio of PM-Rtree is consistent as we increase the number of insert operations, leading to a larger dataset size. Second, we study the search performance of PM-Rtree by executing 100 search operations. As shown in Figure 12(b), PM-Rtree improves search performance by 1.5X. The improvement ratio for search does not change as we increase the size of dataset.

5.6 Impact of Each Optimization

In this section, we study the impact of each optimization discussed in Section 4 using the Random dataset with 6 threads. We enable each optimization incrementally and show the results in Figure 13. *+Lock-free* denotes PM-Rtree using only PMwCAS for thread synchronization. All the non-leaf and leaf nodes are stored in PM. *+Partitioned* denotes PM-Rtree using a partitioned data layout and PMwCAS. Its non-leaf and leaf nodes are stored in DRAM and PM respectively. *+All* denotes PM-Rtree with all the three optimizations enabled including partitioned data layout, lock-free thread synchronization using PMwCAS, and interleaved mapping of bitmaps.

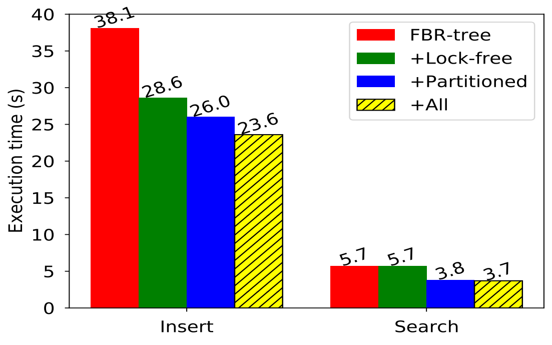


Figure 13: Impact of each optimization used in PM-Rtree.

We have the following observations from the figure. First, lock-free thread synchronization reduces the execution time of insert by 25% compared to FBR-tree. It is the most beneficial optimization because locking overhead is the major issue for FBR-tree. Second, *+Partitioned* further reduces the execution time by 5% because it reduces the number of PM writes by 63%. Third, the execution with *+All* is 9.2% smaller than that with *+Partitioned*. This is because the number of cache line reflushes is reduced by 8.2%. Finally, the performance of search is not affected by lock-free optimization since no mutex locks are used in search. *+Partitioned* and *+All* reduce the search time by 33% and 35% respectively. This is because both the increased node size and partitioned data layout reduce the number of reads in PM.

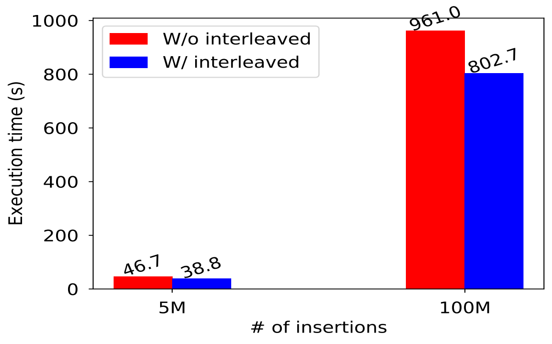


Figure 14: Impact of interleaved mapping of bitmaps.

Next, we study the impact of interleaved mapping with different dataset size. We run PM-Rtree without the interleaved mapping and with interleaved mapping in the experiments. We increase the number of insertions from 5 M to 100 M to increase the dataset size. From Figure 14, we observe that the execution time of the insert operations is reduced by 17% on average and it is not sensitive to the dataset size.

5.7 Performance on Emulated eADR Platform

eADR (extended ADR) is a new feature supported in the 3rd generation Intel Xeon Scalable Processors, which ensures CPU caches are

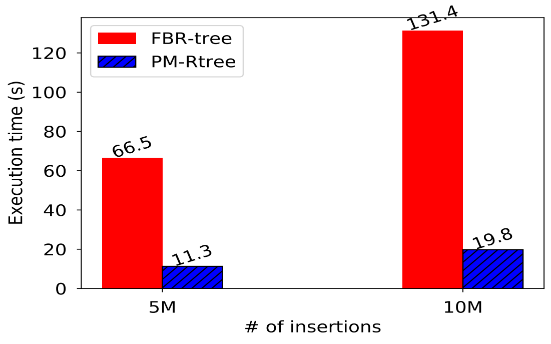


Figure 15: Performance on the emulated eADR platform using 16 threads.

in the power fail protected domain [9]. Explicit cache line refulshes are not necessary on the eADR platform. Since the eADR platform is still not available to us, we use the ADR platform to emulate it by removing the flush instructions in the programs. We only test the insert operation because the search operation does not need to write to PMs.

In the experiments, we use the Random dataset with 16 threads. The results are shown in Figure 15. Without the flush instructions, the performance of PM-Rtree is further improved by 8.2% because the cache line reflushes are eliminated since no flushes are needed. Furthermore, we observe that PM-Rtree outperforms FBR-tree by 6.6X on the eADR platform. This is because on the eADR platform locking overhead still accounts for up to 82% of the execution time of the insert operations. Finally, for both small and large datasets with 5 M and 10 M insertions, PM-Rtree achieves consistently better performance than FBR-tree.

5.8 Failure Recovery

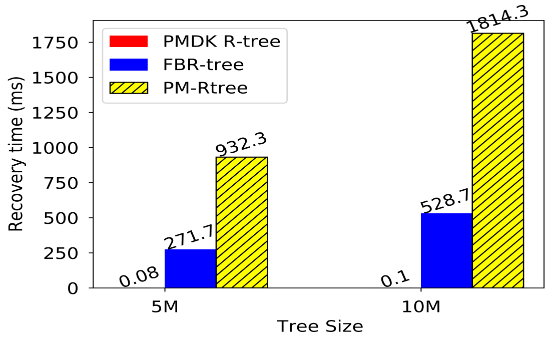


Figure 16: Failure recovery time.

In this section, we measure the time of failure recovery with PMDK R-tree, FBR-tree, and PM-Rtree. In the experiments, we kill the database server process after inserting 5 M and 10 M MBRs and then restart the simulation. We measure the restart time and show it in Figure 16. PMDK R-tree has a small failure recovery time because upon failure recovery the PMDK library simply adjusts PM pool addresses in the process address space and reassigns the root pointer of R-tree in PM. For both FBR-tree and PM-Rtree, they need to retrieve the root node and check the retrievability of PM area allocated before the failure by traversing the tree. PM-Rtree spends 2.4X more time on rebuilding the non-leaf nodes for failure recovery. Even though it takes more time for PM-Rtree to recover from failures, its performance is still acceptable because we assume failures are rare events compared to other events (i.e., insert, search, and delete).

5.9 Skew Test

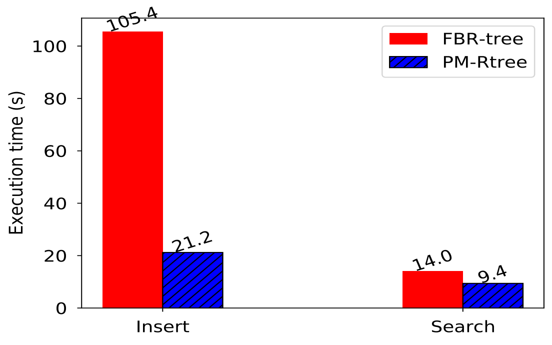


Figure 17: Performance with skew workloads.

We produce a skew workload with a Zipfian key distribution. We set $\alpha = 1.5$ and $n = 1000$ as its shape parameters. We execute 10 M insert operations and 1000 search operations respectively with 16 threads and measure their execution time. The results are shown in Figure 17. We have two observations. First, PM-Rtree works effectively for the skew workload. It reduces the execution time of insert and search operations by 80% and 33% respectively. Second, compared to the random distribution used in the Random dataset, the execution time of PM-Rtree is reduced by 19% and 38% for insert and search respectively with the Zipfian distribution. This is because the Zipfian distribution improves data locality and reduces the number of PM writes and reads for serving the insert and search requests.

6 CONCLUSION

In this paper, we design and implement a crash-consistent PM-aware R-tree, PM-Rtree, which reduces lock contention for concurrent insert operations while achieving high efficiency of PM accesses. Specifically, (1) PM-Rtree uses PMwCAS to implement lock-free insertion/deletion. (2) It reduces PM writes by placing non-leaf nodes in DRAM. (3) It alleviates cache line reflushes by using interleaved mapping of bitmaps in its metadata area. Compared to other persistent R-trees, PM-Rtree improves the execution time by 96.6% for insertions while maintaining crash consistency. It reduces the search time by 19.2% compared to FBR-tree. We hope the proposed techniques in PM-Rtree will facilitate the future design of persistent indexes.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful comments and feedback. We also thank Jia Yu for providing spatial datasets used in the evaluation and Carl Zimmerman for the discussion on this work. This research was supported by US National Science Foundation under CNS 1906541. This work was also funded in part by WSU Vancouver Research Mini Grant.

REFERENCES

[1] Lars Arge, Mark De Berg, Herman Haverkort, and Ke Yi. 2008. The Priority R-Tree: A Practically Efficient and Worst-Case Optimal R-Tree. *ACM Trans. Algorithms* 4, 1, Article 9 (mar 2008), 30 pages. https://doi.org/10.1145/1328911.1328920

[2] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD '90)*. Association for Computing Machinery, New York, NY, USA, 322–331. https://doi.org/10.1145/93597.98741

[3] Shimin Chen and Qin Jin. 2015. Persistent B+-Trees in Non-Volatile Main Memory. *Proc. VLDB Endow.* 8, 7 (feb 2015), 786–797. https://doi.org/10.14778/2752939.2752947

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

[5] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1077–1091.

[6] S. Cho, W. Kim, S. Oh, C. Kim, K. Koh, and B. Nam. 2021. Failure-Atomic Byte-Addressable R-tree for Persistent Memory. *IEEE Transactions on Parallel & Distributed Systems* 32, 03 (mar 2021), 601–614. https://doi.org/10.1109/TPDS.2020.3028699

[7] Richard L Church. 2002. Geographical information systems and location science. *Computers & Operations Research* 29, 6 (2002), 541–562.

[8] Intel Corporation. 2020. Persistent Memory Development Kit. http://pmem.io/.

[9] Intel Corporation. 2021. eADR: New Opportunities for Persistent Memory Applications. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html

[10] Zheng Dang, Shuibing He, Peiyi Hong, Zhenxin Li, Xuechen Zhang, Xian-He Sun, and Gang Chen. 2022. NVAlloc: Rethinking Heap Metadata Management in Persistent Memory Allocators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 115–127.

[11] Athanasios Fevgas, Leonidas Akritidis, Miltiadis Alamaniotis, Panagiota Tsompanopoulou, and Panayiotis Bozanis. 2019. A Study of R-tree Performance in Hybrid Flash/3DXPoint Storage. In *2019 10th International Conference on Information, Intelligence, Systems and Applications (IISA)*. 1–6. https://doi.org/10.1109/IISA.2019.8900716

[12] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD '84)*. Association for Computing Machinery, New York, NY, USA, 47–57. https://doi.org/10.1145/602259.602266

[13] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. A Practical Multi-Word Compare-and-Swap Operation. In *Proceedings of the 16th International Conference on Distributed Computing (DISC '02)*. Springer-Verlag, Berlin, Heidelberg, 265–279.

[14] Chenchen Huang, Huiqi Hu, and Aoying Zhou. 2021. *BPTree: An Optimized Index with Batch Persistence on Optane DC PM.* 478–486. https://doi.org/10.1007/978-3-030-73200-4$_3$2

[15] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies (FAST)*. 187–200.

[16] Kyle Langendoen, Brad Glasbergen, and Khuzaima Daudjee. 2021. *NIR-Tree: A Non-Intersecting R-Tree.* Association for Computing Machinery, New York, NY, USA, 157–168. https://doi.org/10.1145/3468791.3468818

[17] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory As a Scalable Dram Alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*.

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

[19] Soklong Lim, Tyler Coy, Zaixin Lu, Bin Ren, and Xuechen Zhang. 2020. NVGraph: Enforcing Crash Consistency of Evolving Network Analytics in NVMM Systems. *IEEE Transactions on Parallel and Distributed Systems* 31, 6 (2020), 1255–1269. https://doi.org/10.1109/TPDS.2020.2965452

[20] Soklong Lim, Zaixin Lu, Bin Ren, and Xuechen Zhang. 2019. Enforcing Crash Consistency of Evolving Network Analytics in Non-Volatile Main Memory Systems. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 124–137. https://doi.org/10.1109/PACT.2019.00018

[21] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *Proc. VLDB Endow.* 13, 7 (2020), 1078–1090.

[22] Bao Nguyen, Hua Tan, Kei Davis, and Xuechen Zhang. 2019. Persistent Octrees for Parallel Mesh Refinement through Non-Volatile Byte-Addressable Memory. *IEEE Transactions on Parallel and Distributed Systems* 30, 3 (2019), 677–691. https://doi.org/10.1109/TPDS.2018.2867867

[23] Bao Nguyen, Hua Tan, and Xuechen Zhang. 2017. Large-scale Adaptive Mesh Simulations Through Non-volatile Byte-addressable Memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'17)*.

[24] Boundaries of postal code areas. 2022. http://spatialhadoop.cs.umn.edu/datasets.html.

[25] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. 2017. Memory Management Techniques for Large-Scale Persistent-Main-Memory Systems. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1166–1177.

[26] William Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (jun 1990), 668–676. https://doi.org/10.1145/78973.78977

[27] MMM Sarcar, K Mallikarjuna Rao, and K Lalit Narayan. 2008. *Computer aided design and manufacturing.* PHI Learning Pvt. Ltd.

[28] Suman Nath Shimin Chen, Phillip B. Gibbons. 2011. Rethinking Database Algorithms for Phase Change Memory. In *CIDR'11: 5th Biennial Conference on Innovative Data Systems Research.*

[29] Richard Szeliski. 2010. *Computer vision: algorithms and applications.* Springer Science & Business Media.

[30] FIRESTAT Fire Occurrence Yearly Update. 2022. https://data.fs.usda.gov/geodata/edw/datasets.php.

[31] 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 Stroage Technologies (FAST'11).*

[32] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. 2018. Easy Lock-Free Indexing in Non-Volatile Memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 461–472. https://doi.org/10.1109/ICDE.2018.00049

[33] Chin-Hsien Wu, Li-Pin Chang, and Tei-Wei Kuo. 2003. An Efficient R-Tree Implementation over Flash-Memory Storage Systems. In *Proceedings of the 11th ACM International Symposium on Advances in Geographic Information Systems (GIS '03)*. Association for Computing Machinery, New York, NY, USA, 17–24. https://doi.org/10.1145/956676.956679

[34] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. 2019. DPTree: Differential Indexing for Persistent Memory. *Proc. VLDB Endow.* 13, 4 (dec 2019), 421–434. https://doi.org/10.14778/3372716.3372717