Master's Thesis

Failure-Atomic Byte-Addressable R-tree for Persistent Memory

Soojeong Cho

Department of Computer Science and Engineering

Graduate School of UNIST

2020

Failure-Atomic Byte-Addressable R-tree for Persistent Memory

Soojeong Cho

Department of Computer Science and Engineering

Graduate School of UNIST

Failure-Atomic Byte-Addressable R-tree for Persistent Memory

A thesis/dissertation
submitted to the Graduate School of UNIST
in partial fulfillment of the
requirements for the degree of
Master of Science

Soojeong Cho

12/16/2019

Approved by

[illegible]

Advisor
Young-ri Choi

Failure-Atomic Byte-Addressable R-tree for Persistent Memory

Soojeong Cho

This certifies that the thesis/dissertation of Soojeong Cho is approved.

12/16/2019

signature

[illegible]

Advisor: Young-ri Choi

signature

[illegible]

Beomseok Nam: Thesis Committee Member #1

signature

[illegible]

Myeongjae Jeon: Thesis Committee Member #2

three signatures total in case of masters

Abstract

With the emergence of persistent memory (PM), which enables byte-addressable, 64-byte cacheline flush and 8-byte store instructions are expected to be used as persist and failure-atomic write operations, respectively, instead of calls from `fsync()` and `write()` system. The granularity of such a small atomic write represents a challenge to the crash consistency of *in-PM* data structure. In addition, even if the process does not explicitly invoke clflush, a dirty cachelines can be flushed to PM unexpectedly and exposed to other concurrent processes when the system recovers. These challenges occur in PM, but high-density PM is attractive in enabling multi-dimensional indexing to navigate efficiently through large scientific datasets due to their high performance, durability and large capacity.

This work proposes a Fail-atomic Byte Addressable R-tree (FBR tree) that utilizes byte-addressability, persistence and high performance of PM while ensuring crash consistency. We carefully control the order of the store and cashline flush instruction and prevent any sinble store instruction from making the FBR tree inconsistent and unrecoverable. We also develop a non-blocking lock-free range query algorithm for the proposed FBR-tree. Since FBR-tree allows read transactions to detect and ignore any transient inconsistent states, multiple read transactions can access tree nodes concurrently without using shared locks while other write transactions make changes to them. Our performance study shows that FBR-tree successfully reduces legacy logging overhead, and the lock-free range query algorithm shows up to 9.4x higher query processing throughputs than the shared lock-based crabbing concurrency protocol.

Contents

List of Figures

I Introduction

Recent advances in byte-addressable persistent memories (PM) such as 3D Xpoint [1], phase-change memory [2], and STT-MRAM [3] are expected to open up new opportunities to transform main memory from volatile device to persistent storage [4–12]. Due to its persistency and byte-addressability, PM can be used either as slow but large main memory or as fast secondary storage via legacy block I/O interfaces [4, 13–20]. To leverage the high performance, persistence, and byte-addressability of PM, various opportunities are being pursued in numerous domains, including operating systems and database systems [16, 21, 22]. However, how PM will interact with existing systems has not been thoroughly investigated, and a possible role for PM in high-performance computing is currently an open question.

To manage data objects in PM, the properties of PM must be considered. Although we can employ traditional techniques, such as logging and shadowing when designing data structures for PM, logging and shadowing unnecessarily duplicate unmodified portions of data structures, which incurs significant overhead due to additional memory writes and cache line flush instructions. Implementing byte-addressable data structures for PM has very different characteristics from disk-based data structures as well as in-memory data structures. First, there is no guarantee that when modified dirty cachelines will be written back to PM. Even if we do not explicitly call `clflush` instructions, dirty cache lines can be flushed by cache replacement mechanisms. Such a premature cacheline flush can make data structures in PM inconsistent, and such inconsistency becomes permanent and exposed to other processes when a system crashes. Second, when processors store data in PM, they guarantee at most 8 bytes of data to be written atomically. Most data structures consist of logical blocks of composite data types; thus, failure-atomicity at a granularity of 8 bytes is not sufficient to guarantee crash consistency. Third, multiple write operations may not occur in the same order as they are written by a process running on the CPU. Such reordering of memory writes does not harm volatile memory because applications that make changes to volatile memory can protect the inconsistent data via locking mechanisms, and partially written data will be lost when a system crashes. However, if we store data items on PM, such transient inconsistent data written in an arbitrary order will persist across system failures.

To address these challenges associated with a fine-grained write unit in PM, various block-based indexing structures, such as B+-trees [10, 13, 23, 24] and hash tables [25–27] have been redesigned. However, to the best of our knowledge, no previous study has attempted to design multidimensional indexing structures for byte-addressable persistent memory.

Multidimensional range queries are an important class of problems in high-performance computing [28–39], In many scientific domains, the size of data files produced by scientific applications continue to grow, and petabytes or exabytes of data will soon be common. Typically, the content of scientific data files are a collection of multidimensional arrays along with the associated spatio-temporal coordinates [35–37,40]. To help navigating through large scientific datasets, various multidimensional indexing techniques, such as R-trees [41,42], have been developed and used widely to allow for direct access to particular datasets [28,29,43–46]. For example, multi-physics oil reservoir simulation [47], spatial modeling of the brain [39], and disease transmission analysis [40] employ multidimensional indexes to accelerate range query processing performance.

For large-scale scientific datasets, persistent memory offers the ability to bring persistent data closer to the CPU and to extend the capacity of main memory. We note that Intel's latest Optane DC Persistent Memory [1] extends the capacity of DRAM and enables very large storage class memory, i.e., 3 TBytes of memory capacity per server CPU socket.

While R-trees are primarily designed for block-based devices, such as hard disk drives, we have designed a variant of an R-tree for byte-addressable PM. Byte-addressable PM raises new challenges in the use of R-trees because legacy R-tree operations are based on the assumption that block I/O is failure-atomic and memory operations are volatile. However, in PM, each memory operation at the granularity of a word, e.g., 64 bits, must be failure-atomic, i.e., data must be consistent for each store instruction. Otherwise, partially updated inconsistent tree nodes can be exposed to other transactions upon system failures. To prevent the reordering of memory write operations and to ensure that each cacheline flush to PM does not compromise the consistency of R-tree structures, we carefully redesign R-tree algorithms to control the order of memory writes and cacheline flushes so that R-tree can tolerate *transient inconsistency* [24] caused by incomplete write transactions.

The key contributions of this work are as follows.

- We carefully analyze the insert, delete, split, and merge R-tree algorithms and show how byte-addressable updates via a sequence of 8-byte store instructions can avoid unnecessary duplication of data items while guaranteeing the failure-atomicity of R-tree.
- We present two failure-atomic tree rebalancing algorithms for the split and merge operations - *copy-on-write (CoW) based rebalancing* and *in-place rebalancing* with byte-addressable metadata-only logging. Both algorithms eliminate the need for explicit logging of entire dirty pages.
- We also show that fine-grained control of failure-atomic 8-byte store instructions enables the lock-free search for R-trees on PM. Non-blocking queries on R-trees significantly improve the concurrency level and transaction throughput.

The remainder of this paper is organized as follows. In Section II, we present the challenges involved in designing a R-tree index on PM. In Section III, we present the design and implementation of failure-atomic and byte-addressable persistent R-tree (FBR-tree). In Section IV, we discuss concurrency and consistency issues associated with an FBR-tree. In Section V, we evaluate the performance of FBR-trees. Conclusion of this paper is presented in Section VI.

II Challenges in Design of Indexing Trees for Persistent Memory

R-tree structures are similar to B-tree structures in that both structures are balanced search trees and are designed for block device storage where data items are organized in pages. Although, to the best of our knowledge, there exists no prior work that studies multidimensional indexing trees on PM, various B-tree variants for PM have been proposed to resolve the challenges of PM and benefit from its high-performance [13, 23, 24, 48].

In legacy B-trees, key-value pairs are stored in a sorted order, which entails a large number of shifts and cache line flushes. In the legacy disk-based index, the overhead of a large number of shifts and cacheline flushes is negligible due to incomparably large disk I/O overhead. However, in high-performance PM, the overhead of memory barriers and cacheline flushes is known to be the dominant performance factor [10,13,21,23,24,48]. To avoid such a large number of shifts and cacheline flushes, NV-tree [23], FP-tree [48], and wB+-tree [13] have been proposed to append unsorted key-value pairs into the array. The append-only update strategy has been shown to improve write performance, but at the cost of higher lookup overhead because it requires linear scanning of all unsorted keys.

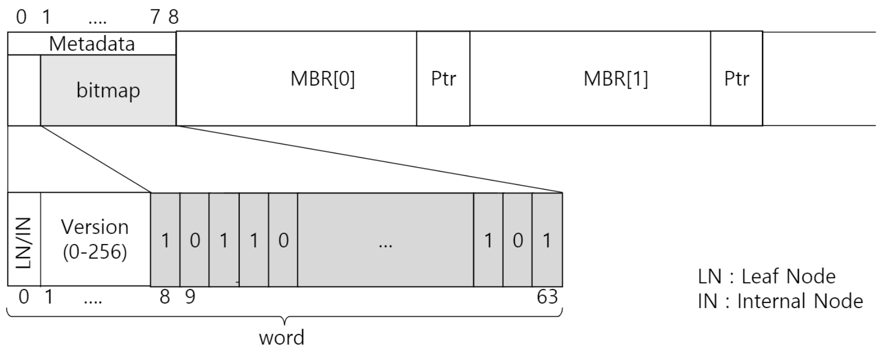
In R-trees, an internal node stores a set of Minimum Bounding Rectangles (MBR) and pointers to the corresponding child node. A leaf node of R-tree stores a set of spatial objects. When a range query is performed, it must be determined whether or not any MBR in a node overlaps the query range. For all overlapping MBRs, the corresponding child node must be visited in a recursive manner until all overlapping nodes have been traversed. When a leaf node is reached, the spatial coordinates of objects are compared against the query range, and their objects are put into the result set if they lie within the search range. Since an R-tree query scans all MBRs in a visited page, the ordering of MBRs in a tree node does not affect the performance of an R-tree search.

When rebalancing a tree-structured index, recovery methods, such as logging and CoW, make the structure recoverable by writing a consistent copy elsewhere prior to updating the tree structure. However, in byte-addressable PM, such per-node logging or per-node CoW is known to be expensive and sub-optimal because it unnecessarily duplicates the entire tree node, including the unmodified portion of it. To resolve this problem, NV-tree [23] and FP-tree [48] use *selective persistence* that keeps leaf nodes in PM but internal nodes in volatile DRAM. This is because internal tree nodes can be reconstructed from scratch when a system restarts. Although the selective persistence makes logging unnecessary, it requires reconstruction of whole tree structures upon any system fault. In that regard, NV-tree and FP-tree are not persistent indexes in strict sense.

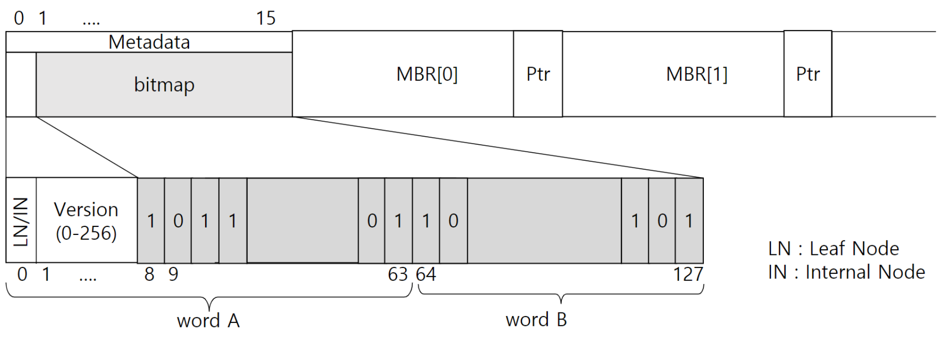
III Design and Implementation of FBR-tree for PM

In this section, we present the details of the tree node structure and tree operations of the proposed FBR-tree (Failure-atomic and Byte-addressable R-tree) optimized for PM with reduced consistency cost.

3.1 Node Structure



(a) Node Structure with 8 Byte Metadata

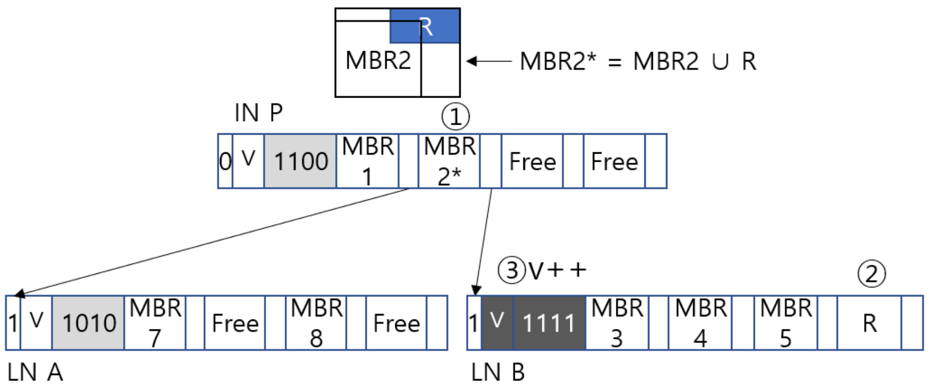


(b) Node Structure with Metadata of Unlimited Size

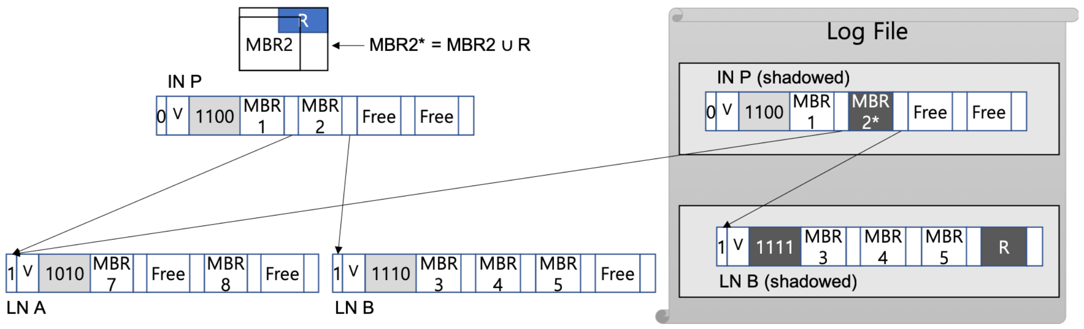
Figure 1: *Node Structure of FBR-tree*

Figure 1 shows the structure of FBR-tree nodes. If the metadata size is bounded by an 8-byte word, as shown in Figure 1(a), we can update the metadata in a failure-atomic manner with a single 8-byte store instruction. Each bit in the bitmap indicates whether its corresponding MBR and pointer are valid. If the k^{th} bit is 0, a pair of `MBR[k]` and its pointer in the node is a free space. Otherwise, the MBR and the pointer are valid and consistent. Therefore, the maximum number of MBRs we can store in a tree node with 8-byte metadata is limited to 55. Note that we use one bit to store the type of tree node (leaf (LN) or internal node (IN)) and one byte to store the version number, which is necessary to implement a lock-free search algorithm, which we will describe in Section IV. The rest of metadata is bitmap. Alternatively, bitmap size can be set arbitrarily large to have node degrees greater than 55, as shown in Figure 1(b). If we use metadata larger than an 8-byte word, write transactions must carefully order memory writes to enforce the consistency because the bitmap that spans across multiple 8-byte words cannot be modified atomically. That is, `word B` in Figure 1(b) can be unexpectedly flushed to PM while we are updating `word A` in CPU cache. To prevent such premature flush, we may employ PMwCAS (Persistent Multi-word Compare-And-Swap) [49] or hardware transactional memory to implement an atomic multi-word write function; however, both approaches incur additional overhead [24, 49].

3.2 Failure-Atomic Insertion


R
MBR2
MBR2* = MBR2 ∪ R
IN P
①
0 V 1100 MBR 1 MBR 2* Free Free
③V++
②
1 V 1010 MBR 7 Free MBR 8 Free
1 V 1111 MBR 3 MBR 4 MBR 5 R
LN A
LN B

(a) Inserting MBR 'R' into R-tree


R
MBR2
MBR2* = MBR2 ∪ R
IN P
0 V 1100 MBR 1 MBR 2 Free Free
1 V 1010 MBR 7 Free MBR 8 Free
1 V 1110 MBR 3 MBR 4 MBR 5 Free
LN A
LN B
Log File
IN P (shadowed)
0 V 1100 MBR 1 MBR 2* Free Free
1 V 1111 MBR 3 MBR 4 MBR 5 R
LN B (shadowed)

(b) Legacy Disk-based Logging

Figure 2: *Byte-addressable Insertion in FBR-tree*

When inserting a new spatial object into an R-tree, the tree is traversed recursively from the root node to a leaf node. At each node, a candidate child node is selected using a legacy heuristic, such as the *least enlargement algorithm* [41]. If the chosen MBR does not completely overlap the new spatial object, the MBR must be enlarged to contain the new object, as shown in Figure 2(a). In legacy disk-based R-trees, an MBR update requires CoW of the entire tree node to provide atomicity and crash consistency. For example, in the legacy disk-based write-ahead logging method, a new node is stored as a redo log first, as shown in Figure 2(b) and then checkpointed to the index later.

However, with byte-addressable PM, such per-node CoW incurs unnecessary memory copy overhead. For example, in the example shown in Figure 2(b), the undo logging method copies `MBR1` to the log so that it can be restored when a transaction aborts. To eliminate such per-node CoW overhead, FBR-tree employs fine-grained byte-addressable *metadata-only logging* and performs in-place updates without hurting the consistency and failure-atomicity.

Algorithm 1 Insert(Obj obj, Node *n, Node *parent)

```
1:  n → mutex.lock()
2:  if node is NOT leaf then
3:    locked = true;
4:    pos = PickChild(obj.r, n);
5:    n → child[pos].mbr = Combine(n → child[pos].mbr, obj.r);
6:    persist(n → child[pos]); // step 1 in Figure 2.(a)
7:    if n → child[pos] is NOT FULL then
8:      n → mutex.unlock(); locked = false;
9:    end if
10:   c_sibling = Insert(obj, n → child[pos].ptr, parent);
11:   if  c_sibling is NOT NULL  then
12:     child node has split, c_sibling is its sibling node
13:     Rect c_sibling_mbr = getMBR(sibling)
14:     if n is also full then
15:       create a splitLog (parent, current and a new sibling)
16:       persist(splitLog);
17:       sibling = split(n, c_sibling, parent);
18:       n → mutex.unlock();
19:     else
20:       n → child[free].mbr = c_sibling_mbr;
21:       persist(n → child[free]); // step 4 in Figure 4.
22:       n → child[pos].mbr = getMBR(n → child[pos]);
23:       increase n → version and update valid bit of free;
24:       persist(n → metadata); // step 5 in Figure 4.
25:       persist(n → child[pos].mbr); // step 6 in Figure 4.
26:       persist(n → child[pos].ptr→version = 1); // step 7
27:       sibling = NULL;
28:       n → mutex.unlock();
29:     end if
30:   else
31:     sibling = NULL;
```

```
        if locked is true then
            n → mutex.unlock()
        end if
    end if
    return sibling;
else
    if n → bitmap is FULL then
        create a splitLog (parent, current and sibling)
        sibling = split(n, obj, parent);
    else
        pos = n → getFreeSpace()
        n → child[pos] = obj;
        persist(n → child[pos]); // step 2 in Figure 2.(a)
        increase n → version and update valid bit of pos;
        persist(n → metadata); // step 3 in Figure 2.(a)
        sibling = NULL;
    end if
    n → mutex.unlock();
    return sibling;
end if=0
```

Algorithm 1 shows the insertion algorithm of FBR-tree. First, we select a child node and enlarge the MBR of the child node, as shown in Figure 2(a). Then, we call `mfence` and `clflush` to persist the updated MBR. Note that we perform in-place updates for the selected MBR (`MBR2` in the example) without logging. An MBR has multiple spatial coordinates, thus it cannot be updated in a failure-atomic manner. To guarantee failure-atomicity, we CoW the MBR such that we retain the old MBR while writing a new one. Then, we validate the new copy by atomically flipping the valid bit of the old copy and the new copy. However, we note that such a CoW is not necessary for MBR updates because insertions only enlarge the size of the MBRs. In other words, as long as the MBR contains all MBRs of child nodes, MBR updates do not have to be atomic and the write ordering of spatial coordinates does not violate the correctness of the index. For example, no matter whether the boundary of the first dimension or the second dimension is updated, the partially updated MBR will still include all child MBRs. Suppose a system crashes after only one of the boundaries is overwritten. Subsequent queries will still successfully find and visit the child node if their query ranges overlap the child node's MBR.

We note that partially enlarged MBRs can hurt the efficiency of the index because the partially enlarged MBR may unnecessarily overlap incoming queries due to the *dead space*, i.e., space that contains no data objects. However, such false positive results do not hurt the correctness, and the partially updated MBRs never return false negative results. We also note that such a dead space problem is not permanent since a dead space can be removed when a node splits or when underutilized nodes merge.

On the way down to a leaf node, we keep updating MBRs when necessary so that all ancestor nodes contain the new spatial object. Once we find a leaf node and insert a new spatial object, we search for free space by checking the bitmap in the leaf node and store the object's spatial coordinates in it (step 2 in Figure 2(a)). Then, we call `mfence` and `clflush` to persist the new object. In the next step (step 3 in Figure 2(a)), we increase the version number and update the bitmap to validate the new object. If the version number and bitmap are stored in an 8-byte word, they can be atomically updated and flushed. The version number update is necessary to enable a lock-free search, which we will discuss in Section IV. If a system crashes before the bitmap is updated, the written object will be ignored and considered as a free space when the system recovers, i.e., no recovery process is required. In such a sense, the insertion algorithm of the FBR-tree is failure-atomic although it does not perform logging.

3.3 Failure-Atomic Deletion

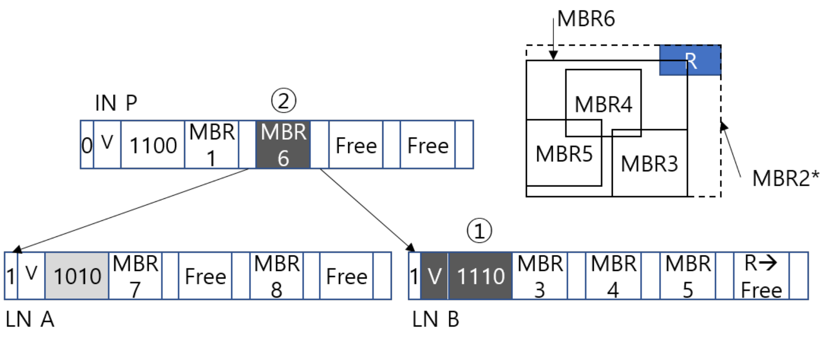


Figure 3: *Byte-addressable Deletion in FBR-tree*

When an indexed spatial object is deleted from a tree node, FBR-tree flips the valid bit of the object and flushes the bitmap to persist it, as shown in the first step of Figure 3. If the deleted object is entirely within the MBR of its leaf node, the MBR of the leaf node will not be modified. However, if the deleted object shares at least one boundary with the MBR of leaf node, the MBR of the leaf node needs to be shruken by the deletion. In such a case, we backtrack to the parent node and update the leaf node's MBR accordingly. To reconstruct the MBR, we select the minimum and maximum boundaries in each dimension and perform in-place updates to overwrite the existing MBR. Again, we note that shrinking the MBR also does not have to be atomic because partially updated MBR does not affect the invariants of the index. In other words, no matter what dimension has been updated and flushed to PM, when a system crashes, the partially updated MBR will still contain all spatial objects in the sub-tree, and it will guarantee correct search results. Therefore, the FBR-tree deletion algorithm is also failure-atomic and guarantees consistency. Note that such in-place updates greatly reduce the amount of I/O since transactions do not need to perform expensive logging. In addition, in-place updates simplify the lock-free search, as we will describe in Section IV.

3.4 Failure-Atomic Page Split

Insertions and deletions often result in node overflows and underflows, which respectively require nodes to split and merge such that the tree height becomes re-balanced. In disk-based R-trees and B-tree variants [13,23], *per-node* logging or journaling has been used because multiple tree nodes including a parent node need to be updated atomically. In legacy logging or journaling, unmodified portions of tree nodes are duplicated in a log or journal file because the minimum write granularity of disks is a disk page. Such legacy disk-based logging not only increases the write traffic but also blocks concurrent access to tree nodes. Unlike a disk-based R-tree, FBR-tree re-balances the tree height in a byte-addressable and failure-atomic manner without duplicating a clean portion of the tree nodes. We propose two methods: (1) byte-addressable CoW-based split and (2) in-place split with minimal metadata logging.

Byte-addressable Copy-on-Write Split

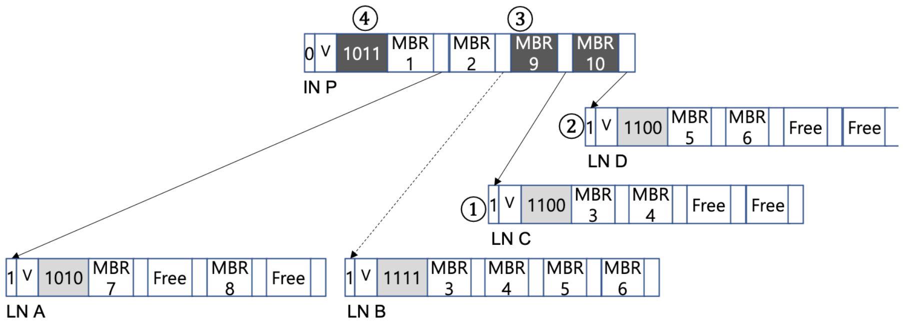

④
③
0 V 1011 MBR 1 MBR 2 MBR 9 MBR 10
IN P
② 1 V 1100 MBR 5 MBR 6 Free Free
LN D
① 1 V 1100 MBR 3 MBR 4 Free Free
LN C
1 V 1010 MBR 7 Free MBR 8 Free
LN A
1 V 1111 MBR 3 MBR 4 MBR 5 MBR 6
LN B

Figure 4: *CoW Split in FBR-tree*

Figure 4 shows the steps required to perform byte-addressable CoW when a node overflows. First, we allocate two new nodes and copy half of the entries to these nodes (steps 1 and 2 in the example). Then we insert their MBRs and pointers into the parent node `P`. Once we flush the new MBRs (step 3 in the example), we overwrite the bitmap via a single store instruction and call `clflush` to persist it.

If a system crashes before we add the new nodes to the parent, the index is consistent because nothing has been changed. If a system crashes after we add the two new nodes to the parent node but before we update the bitmap, the index is still consistent because the two child nodes will be considered invalid, i.e., free spaces. PM heap manager, such as Intel's PMDK [50] or HPE's NVMM [51] should be able to deal with memory leak problems. For example, a PM heap manager should create a log when it allocates a PM block. Then, the heap manager can check whether each PM block is being used by an application when a system recovers. If not, the block must be garbage collected. Note that this is not a requirement specific to our FBR-tree. All other PM-based data structures also require this feature to prevent memory leak problems. If a system crashes after the bitmap is updated, the index is in a consistent state; thus, recovery is not required.

CoW split updates three bits, two bits to validate new nodes and another bit to invalidate the overflow node. If the size of a bitmap is larger than 8 bytes, CoW split does not provide failure-atomicity because three bits can be in different words. Therefore, the maximum number of child nodes we can store in each tree node is limited to 55. If we want a tree node to have a larger number of child nodes, as shown in Figure 1(b), we must use explicit logging for a bitmap. However, the size of a log will still be much smaller than legacy per-node logging. If a system crashes before we put a commit mark in the bitmap log, the logged bitmap will be ignored, and the index will be in its previous consistent state. If a system crashes after we put a commit mark in the bitmap log, the index will be in a new consistent state. Therefore, our CoW split algorithm is failure-atomic.

A drawback of a CoW split is that we need at least two free spaces on the parent node. If there is only one last free space in the parent node, the parent node cannot accommodate two new child nodes; thus the parent node also has to split. For example, after we update the bitmap of parent node `P` in the example shown in Figure 4, `MBR2` will become a free space. However, if another child node splits again, node `P` must split even though there is a single free space. Such a premature split degrades node utilization.

Byte-addressable In-Place Split with Minimal Logging

We can also avoid such a premature split problem if we employ small metadata-only logging and perform in-place updates. Figure 5 shows the steps for the in-place split algorithm, and Algorithm 2 shows the in-place split algorithm.

First, when a node overflows, we create a log space and write minimal metadata to indicate which node is splitting, which node is created as a sibling, and which node is the parent node. In the example, we write the address of node `B`, `C`, and `P`. The split log stores the address of the overflow node as key and a pair of siblings and the parent node addresses as values. Then, we allocate memory space for a sibling node (`C` in the example) and copy half of the entries to this sibling node (step 2). For the other half, we reuse the overflow node (`B` in the example). Note that the order of the first two steps can be changed as they do not affect the invariants of the index.

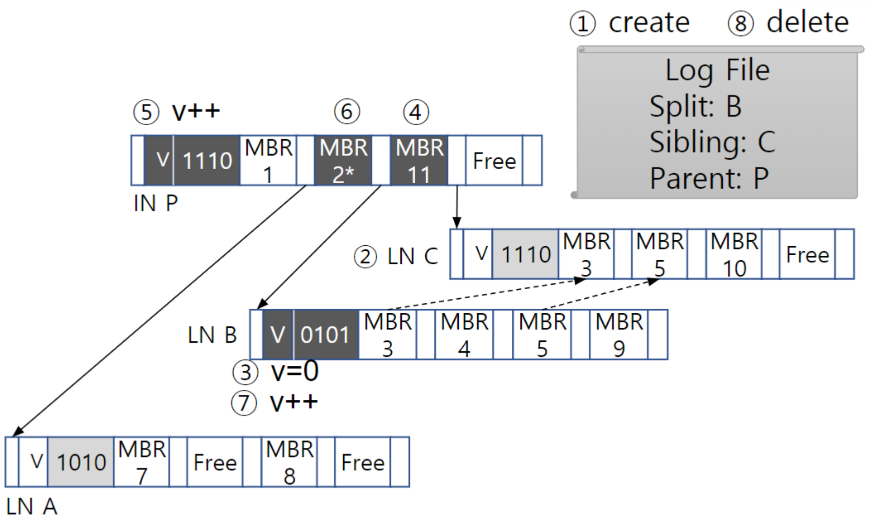


Figure 5: *In-Place Split in FBR-tree*

In the third step, we set the version of the overflow node to 0 and invalidates the migrated entries by overwriting the bitmap (step 3). Since we store the version and bitmap in the same 8-byte word, the version and bitmap are atomically updated. Version 0 indicates that the node is splitting. Any subsequent transaction that accesses a version 0 tree node must check its split log and access the newly created sibling node. Note that we have not added the new sibling node (`C`) to the parent node; however, the migrated entries are invalidated in the overflow node. At first glance, it appears as though the index is not in a consistent state because the migrated entries are not accessible from the root node. However, since subsequent transactions can find the sibling node from the split log, the correctness of search operations is not compromised.

In the next step (step 4), we add the address and MBR of the new sibling node to the parent (`MBR11`). Note that the MBR is not valid until the bitmap is updated in the next step (step 5). Again, we update the bitmap and version number atomically. After validating the new child node, we update the MBR of the overflow node to make it just small enough to include the remaining MBRs (step 6). The MBR of the overflow node must not be updated prior to validating a new child node. Otherwise, a query that searches for a migrated entry may fail to find it. For example, suppose a query is searching for `MBR5` in Figure 5. If we reduce the size of `MBR2` and `MBR5` does not overlap the reduced `MBR2`, the query will not visit node `B` and will never know `MBR5` has been moved to a new sibling node because node `C` is not added to the parent node. To avoid this problem, the order of each update must be strictly enforced. Otherwise, failure-atomicity and consistency will not be guaranteed. Therefore, we call `mfence` and `clflush` in each step. Finally, we increase the version of the overflow node to indicate the split process has completed (step 7), and we delete the log entry (step 8).

Suppose a system crashes before the version of the overflow node is set to zero (step 3). Since nothing has changed in the index, recovery is trivial. If a system crashes after the version of the overflow node has been set to zero but before the sibling node has been added to the parent (step 5), we can recover from the system failure by replaying the split log. If a system crashes after the sibling node has been added to the parent and validated but before the version of overflow node (step 7) has been increased, we will replay the log and determine that the sibling node has already been added to the parent node. Again, recovery is possible. If a system crashes after the version of the overflow node has been increased, the index is now in the next consistent state; therefore, recovery is unnecessary.

Algorithm 2 split(Node *n, Entry *entry, Node *parent)

```
 1: sibling = create a sibling node;
 2: cluster existing entries into group A and B
 3: B is the group that the new entry belongs to
 4: tmp_metadata.bitmap = n$\rightarrow$bitmap;
 5: for i = 0; i < size; i++ do
 6:    if n $\rightarrow$ branch[i] is in group B then
 7:       addEntry(sibling, n $\rightarrow$ child[i]);
 8:       tmp_metadata.bitmap[i] = 0;
 9:    end if
10: end for
11: sibling $\rightarrow$ version = 1;
12: persist(sibling);
13: tmp_metata.version = 0;
14: n $\rightarrow$ metadata = tmp_metdata;
15: persist(n $\rightarrow$ metadata);
16: if parent is NULL then
17:    n is the root node
18:    new_root = create a new node;
19:    write_log(new_root, n, sibling);
20:    AddEntry(new_root, n);
21:    AddEntry(new_root, sibling);
22:    initialize new_root.metadta;
23:    persist(new_root);
24:    root = new_root;
25:    return NULL;
26: end if
27: return sibling; =0
```

3.5 Failure-Atomic Node Merge

If deleting a key from a node causes an underflow, FBR-tree redistributes entries between sibling nodes. Such redistribution requires bitmap logging because multiple bitmaps must be updated atomically. If a sibling node also has the minimum number of keys, the two nodes are merged to improve node utilization. We propose and compare two merge operation methods, i.e., (1) byte-addressable CoW merge and (2) in-place merge with minimal metadata logging.

Byte-addressable Copy-on-Write Merge

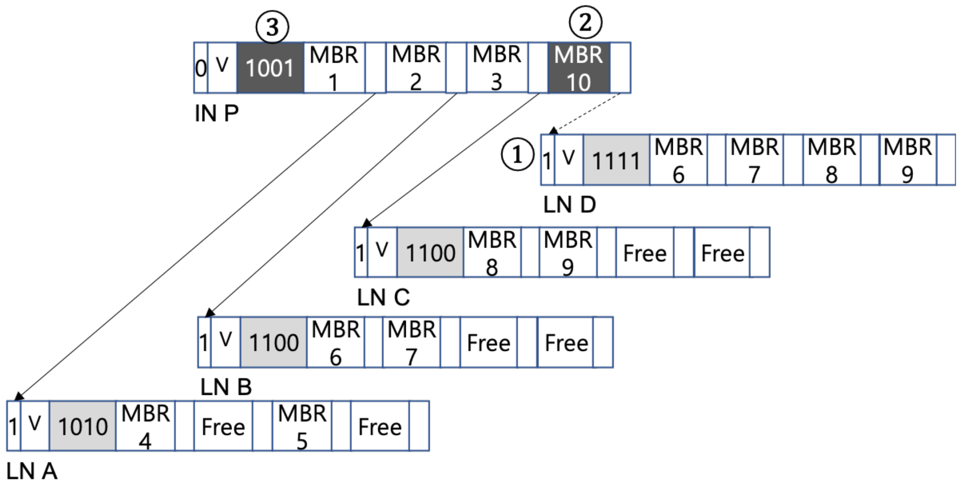

③
②
0 V 1001 MBR 1 MBR 2 MBR 3 MBR 10
IN P
①
1 V 1111 MBR 6 MBR 7 MBR 8 MBR 9
LN D
1 V 1100 MBR 8 MBR 9 Free Free
LN C
1 V 1100 MBR 6 MBR 7 Free Free
LN B
1 V 1010 MBR 4 Free MBR 5 Free
LN A

Figure 6: *CoW Merge in FBR-tree*

Figure 6 shows the steps required for a merge operation. In the walking example, we delete an entry from leaf node `B`, which causes an underflow. Since the leaf node `B` cannot borrow an entry from its sibling node `C`, nodes `B` and `C` need to be merged. Therefore, we allocate a new node `D` and copy `MBR6`, `MBR7`, `MBR8`, and `MBR9` to this new node. Then, we persist node `D` (step 1). In the next step, we add the MBR of node `D` to the parent node `P` as in normal insertion algorithm (step 2 and 3). Since the failure-atomic bitmap update will invalidate two under-utilized nodes and validate the new merged node atomically, the CoW merge algorithm is failure-atomic.

Byte-addressable In-Place Merge with Minimal Logging

Differing from CoW split, CoW merge does not cause the node utilization problem. However, CoW operations require more memory copy operations than in-place updates. Thus, we design and implemented the in-place merge algorithm.

As shown in Figure 7, when a node underflows, a merge log is created. This merge log stores the address of the underflow node as key and the addresses of the sibling node and parent node

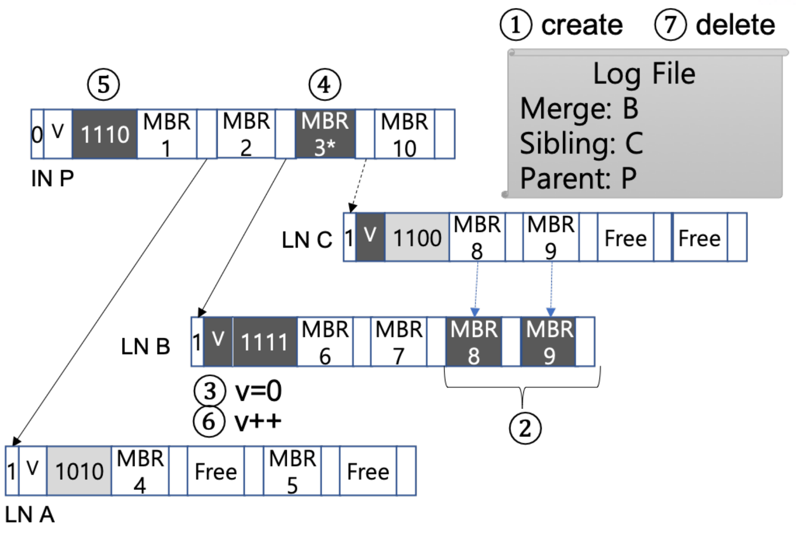


Figure 7: *In-Place Merge in FBR-tree*

as values (step 1). Then, we copy all entries from a sibling node to the underflow node (step 2), and update the bitmap to validate the migrated entries (step 3). Note that the sibling node must be also underutilized when merging. We also set the version of node `B` to 0 to indicate that the merge log must be accessed. If a system crashes at this point, the recovery process will read the log and replay the merge process. In the next step (step 4), the MBR of node `B` in the parent node is updated to include the entry migrated from an underutilized node. Then, the bitmap of the parent node and its version number are updated atomically to invalidate the underflow node `C` (step 5). Once the underflow node is removed from the parent node, the version of the merged node is increased (step 6) and the merge log is deleted (step 7). Note that we have not discussed crash consistency due to its symmetry with the in-place split algorithm.

IV Lock-Free Search

With the increasing prevalence of many-core systems, the importance of concurrent data structures also increases. One challenge for concurrent data structures is the lock contention between concurrent transactions. If a transaction accesses a data structure while it is being modified by another transaction, it may access the data structure in an inconsistent state. Various lock methods have been used to protect data structures from concurrent accesses. However, due to synchronization overhead, lock methods often degrade the concurrency level and degrade performance.

Optimistic synchronization has been proposed to reduce synchronization overhead [24,52,53]. With optimistic synchronization, search operations access data structures without acquiring shared locks. Instead, search operations optimistically access each node of the data structures and rollback when they later find that inconsistent nodes have been accessed.

FBR-tree takes this optimistic approach, and the non-blocking search operations of the FBR-tree guarantee system-wide progress and consistency. As described previously, the sequences of 8-byte store operations in the FBR-tree guarantee that the index is recoverable from system failures at any time. In other words, if every 8-byte store operation in write transactions guarantees recovery, and if a read thread knows when it needs to read a metadata log to avoid accessing a transient inconsistent tree, no read thread will ever access inconsistent tree nodes. Even if a write thread fails while making changes to tree nodes, subsequent transactions after recovery will be able to replay or rollback to a consistent state of the partially written index. Similarly, even if a read transaction accesses a tree node partially updated by a suspended write thread, it is guaranteed that read transactions can detect the partially written transient inconsistent tree nodes, and they can construct a consistent view of tree nodes without waiting for write transactions to release the exclusive lock.

In FBR-tree nodes, a version number is stored along with a bitmap in each node to indicate the node is in a transient inconsistent tree state, i.e., when an entry in a tree node is added or deleted, its version number is increased. When a search query visits a tree node for the first time, it remembers its version number. Later, when the query is done accessing the node, it verifies that the version number has not changed. If the version has changed, the query knows that the node has been changed and that it has to read the node again to guarantee serializability and strong consistency. Such a rollback operation is expensive as it may result in re-traversal of sub-trees. However, we note that the rollback operation is less expensive than the reader-writer lock mechanism, which requires write operations.
We note that our lock-free search algorithm always returns a result set even if the search algorithm does not verify the version of tree nodes. However, if the version metadata is not used,

the query result becomes vulnerable to dirty reads although it may return query results faster. I.e., if an application does not require serializability, our search algorithm may omit checking the version metadata and work in *read uncommitted mode* [54] to improve the query performance. If the version metadata and rollback method are employed, the presented lock-free search operates in *serializable mode* [54]. The detailed algorithm of lock-free search in FBR-tree is shown in Algorithm 3.

Algorithm 3 Search(Node *n, Query *q)

```
hitCount = 0;
if n != leaf then
  version = n → version;
  initialize 'childqueue';
  while true do
    for i=0; i<n → size; i++ do
      if branchOverlap(n→branch[i],q) && n→bitmap[i]==1 then
        childqueue.push( n→branch[i] );
      end if
    end for
    if version==n→version && version != 0 then
      while !childqueue.empty() do
        hitCount += Search(childqueue.front(), q);
        childqueue.pop();
      end while
      break;
    else if version!=n→version then
      // child was split and added branch
      remove all entries in childqueue
      version = n→version;
    else if version == 0 && n→version == 0 then
      // need to check a log
      logEntry=searchLog(n);
      if logEntry then
        for i=0; i<n → size; i++ do
          if branchOverlap(logEntry→branch[i], q) && logEntry→bitmap[i]==1 then
            childqueue.push(logEntry→branch[i]);
          end if
        end for
      else
        remove all entries in the childqueue
```

```
                version = n→version;
            end if
            while !childqueue.empty() do
                hitCount += Search(childqueue.front, q);
                childqueue.pop();
            end while
            break;
        end if
    end while
else if n is leaf then
    for i=0; i<size; i++ do
        if branchOverlap(n→branch[i], q) && n→bitmap[i]==1 then
            hitCount++;
        end if
    end for
end if
return hitCount =0
```

Note that FBR-tree insertion, deletion, split, and merge algorithms do not allow lock-free writes. Therefore, we use legacy exclusive locking to avoid write-write conflicts. This is because the FBR-tree must guarantee not only byte-addressable consistency but also durability as an additional challenge. In future, we intend to consider lock-free writes for a persistent index.

Consider the example shown in Figure 2. If a read transaction accesses leaf node `B` when another write transaction stores the MBR `R` into the same node prior to its bitmap being updated. If the read transaction returns without reading `R` because the version is not updated, the two transactions are serializable without incurring a consistency issue (read $\rightarrow$ write). If the write transaction updates the bitmap before the read transaction returns, the read transaction must rescan the leaf node and thus will read the new MBR `R`. Therefore, again the two transactions are serializable (write $\rightarrow$ read). We omit a discussion of deletions due to its symmetry with insertions.

Now, consider the example shown in Figure 4. Suppose a read transaction accesses the leaf node `B` and returns to its parent node `P` while another write transaction is making changes to `P` using CoW. If the node `P` version has not yet been changed, the read transaction does not have to read the spatial object written by the write transaction. Thus, the read transaction will successfully return, and the two transactions are serializable (read $\rightarrow$ write). If the write transaction updated the bitmap of node `P`, the read transaction will discard the results found in leaf node `B` and visit new child nodes `C` and `D`. Again, the two transactions are serializable (write $\rightarrow$ read).

As for the in-place split, we update versions three times as shown in Figure 5. If a read transaction accesses a leaf node and returns from its parent node prior to the version of an overflowing node being set, the transactions are serializable (read → write) and there is no consistency issue. However, a transaction can be suspended indefinitely, which can complicate the interaction between concurrent read and write transactions.

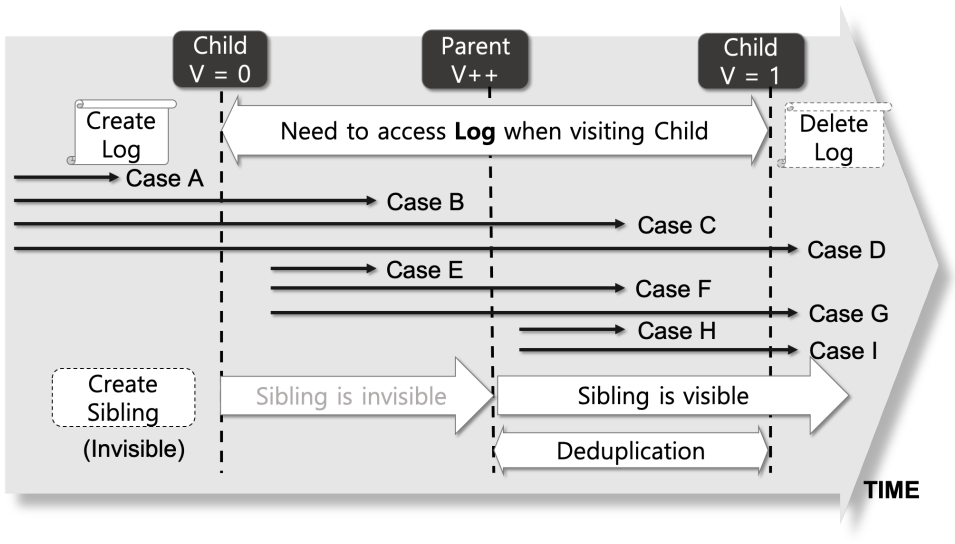


Figure 8: *Lock-Free Search with Concurrent In-Place Updates*

Figure 8 shows all possible execution orderings of concurrent read and write transactions. Vertical dash lines indicate when a write transaction updates each version number. The left end of the arrow indicates when a read transaction reads the version of a child node, and the right end of the arrow indicates when the read transaction verifies the version of a parent node.

(i) In the case of B in Figure 8, a read transaction accesses a leaf node before a write transaction migrates one half of the entries to a new sibling node and accesses its parent node before the sibling node is added to its parent node. Since the read transaction has already accessed all entries from the overflow node, the read transaction returns correct results.

(ii) In the case of C, a read transaction will determine if the parent node was modified by a write transaction, and the read transaction will discard the results found from the previous sub-tree traversal and read child nodes again. If the version of a child node is 0, it has to access the split log. Since the new sibling node is already pointed by its parent node, we make read transactions keep track of the version of visited tree nodes to avoid unnecessary multiple visits to the same node.

(iii) Case D is the same as case C except that the read transaction does not read the split log.

(iv) In case of E, a read transaction accesses a leaf node after the leaf node deletes migrated entries but before its sibling node is added to the parent node. Since we always check the split log if the version of a tree node is zero, the read transaction reads the split log to find the sibling node.

(v) Case F is similar to E. However, because the version of the parent node has been changed, the read transaction must rollback the previous sub-tree traversal, reread the parent node, and visit the child nodes. Although the sibling node can be pointed by both parent node and split log, we avoid visiting the same nodes unnecessarily by keeping track of the version of visited tree nodes.

(vi) Case G is similar to F except that the read transaction does not have to read the split log.

(vii) For cases H and I, a read transaction accesses a consistent tree structure. However, for cases H and I, the version of a leaf node is 0, which makes the read transaction access a split log and consider a new sibling node. However, our tree traversal algorithm allows us to avoid visiting the same nodes unless their version has changed. Therefore, the read transaction can return correct results.

As discussed above, our non-blocking range query algorithm can detect any transient inconsistency caused by a concurrent write transaction. Therefore, read transactions can access the FBR-tree in a non-blocking manner even if some nodes are being modified by concurrent write transactions.

V Evaluation

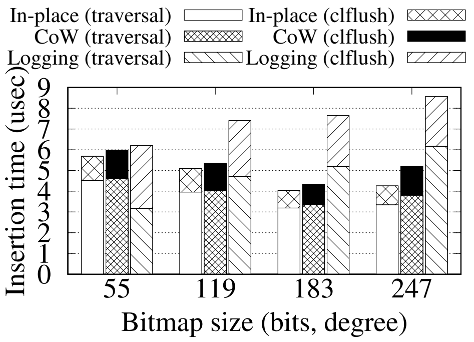
We designed and implemented variants of FBR-trees and evaluate their performance on a workstation that has two Intel Xeon Gold 6230 processors (20 cores, 2.1GHz, 20x32 KB instruction cache, 20x32 KB data cache, 20x1024 KB L2 cache, and 27.5 MB L3 cache), 375 GB of DDR4 DRAM and 1 TB Intel Optane DC Persistent Memory (DCPM). To create and manage FBR-trees in DCPM, we use Persistent Memory Development Kit (PMDK), which is designed by Intel to facilitate programming for persistent memory. To make use of atomic 8-byte instructions, we allocate a single large persistent memory pool for each index and call `pmemobj_alloc()` for each page, which returns an 8-byte pointer to the page. For failure-atomicity, we carefully enforce the ordering of `mfence` and `clwb` instruction rather than using the PMDK transaction APIs, which performs expensive logging.

Among numerous heuristics, we use the least enlargement algorithm [41] to select a child node when inserting spatial objects. In the evaluation experiments, we used two datasets. One is HDF-EOS L1B solar event transmission datasets that is collected by SAGE (Stratospheric Aerosol and Gas Experiment) III/ISS instrument mounted on the International Space Station. SAGE III/ISS dataset is being used to understand the Earth's atmosphere and ozone depletion. The other dataset is a time series multidimensional *taxi service trajectory* dataset that has more than 80 million polylines and a total of nine attributes [1] For both datasets, we generated synthetic range queries simulating a varying number of users posing queries to the index, modeled as a Poisson process. The workload generator creates range queries from various synthetic distributions such as uniform and zipfian. We present the performance results of query workloads in uniform distribution due to the lack of space, but the results of other workloads is not significantly different.

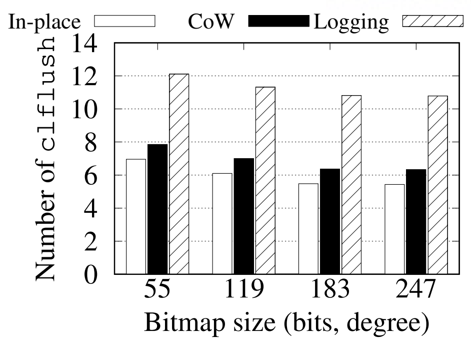
5.1 Node Size: Single vs. Multi-word Bitmap

In the first set of experiments shown in Figure 9, we insert 80 million Taxi trajectory polylines into the FBR-tree and evaluate performance while increasing the size of bitmaps. With 80 million polylines, the index size is approximately 10 Gbytes. Note that the tree node size in in-memory data structures is a performance tuning parameter that can be set arbitrarily, i.e., unlike disk-based data structures, each node size does not have to match the disk block size. In addition, the number of degrees (fan-outs) in FBR-tree nodes is determined by the bitmap size not the dimension. In legacy disk-based R-trees, the number of degrees is reduced as the number of dimensions in increased, which aggravates the well-known curse of dimensionality problem. However, the FBR-tree on PM alleviates the drawback of handling a small number of child nodes.

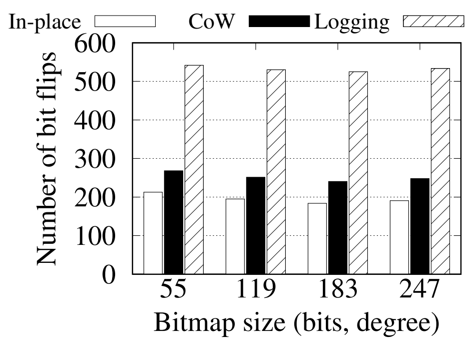
[1]The dataset is available at https://archive.ics.uci.edu/ml/machine-learning-databases/00339/Porto_taxi_data_test_partial_trajectories.csv.



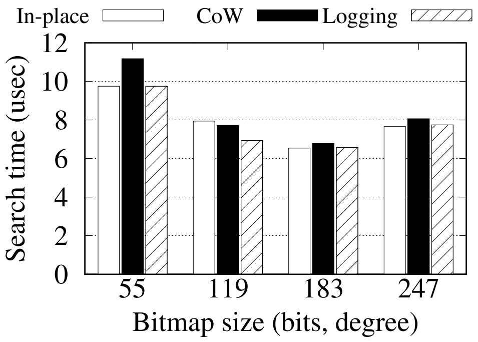
(a) Insertion Time



(b) Number of clflush



(c) Number of bit flips



(d) Search Time

Figure 9: Insertion Performance with Varying Node Size

When the bitmap size is 7 bytes, `CoW` does not require bitmap logging when rebalancing. Note that we reserve 1 byte for version and node type metadata. For larger bitmap sizes, `CoW` requires bitmap logging because the bitmap cannot be updated atomically without logging. For `In-place`, we require the split and merge log for any bitmap size is. However, since the bitmap and split/merge logs are so small in PM, they incur negligible performance overhead.

Overall, legacy logging, denoted as `Logging`, demonstrates the worst performance because it duplicates entire dirty nodes. `CoW` and `In-place` yield comparable performance, but the number of `clflush` required for `CoW` is approximately 12% and 42% greater than that of `In-place` and `Legacy logging` respectively. In particular, the insertion time involves of two parts, i.e., tree traversal and cacheline flush times. In terms of cacheline flush time, `In-place` is up to 19.8% faster than `CoW`.

Relative to tree traversal time, denoted as `traversal`, `In-place` is also approximately 5.1% faster than `CoW` because it spends less time selecting a child node. However, the difference in traversal time requires extensive investigation. A side effect of the `CoW` split algorithm is that it can compact MBRs and child pointers when new nodes are created. In contrast, `In-place` creates holes when migrating entries to a sibling node. Such fragmentation can increase the number of cacheline accesses and reduce search performance when MBRs are scanned. However, due to the difference in managing free space, the order of MBRs stored by `CoW` and `In-place` differs, which affects how quickly an insert query finds a completely overlapping MBR, i.e., if an insert query finds a completely overlapping MBR, it stops scanning the remaining MBRs and visits the corresponding child node. In other words, different MBR ordering in two schemes accounts for the difference in traversal time. We observe that `In-place` outperforms `CoW` and other times underperforms because the traversal time for insert transaction is primarily affected by how quickly an insert query finds a completely overlapping MBR, i.e., for `In-place` schemes, the fragmentation issue is not a critical performance problem relative to insertion queries.

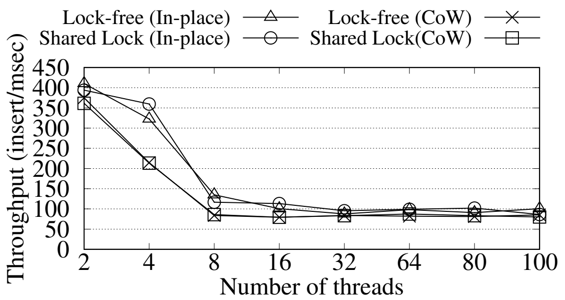
Figure 9(b) shows the number of bit flips. PM technologies only support a limited number of writes per cell, and bit flipping consumes most of the power required for PM; thus, the number of bit flipping is an important performance metric in PM systems. The results demonstrate that that `In-place` reduces the number of bits flipped by up to 1.3x and 2.85x over `CoW` and `Logging`, respectively.

Relative to range query performance, shown in Figure 9(d), we do not observe significant difference between `In-place`, `CoW`, and `Logging`. However, the 183-bit (three words) bitmap demonstrates the fastest search performance for all methods. With larger bitmap sizes, the tree node degree increases and tree height is reduced. However, a very large tree node requires more comparisons against MBRs in each node; therefore, search performance improvement is saturated when the degree is sufficiently large. For the rest of the experiments, we set the bitmap size to 23 bytes (183 bits) because this gave the fastest search performance and good insertion performance with all three schemes.

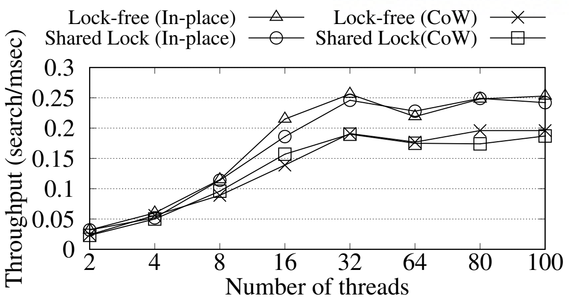
5.2 Concurrency and Recoverability

In the experiments shown in Figure 10, we evaluate the performance of multi-threaded FBR-trees on Optane DCPM. We implemented the lock-free search algorithm we described in Section IV and the *crabbing* protocol [54] as a baseline, which uses `std::shared_mutex` class in C++17.

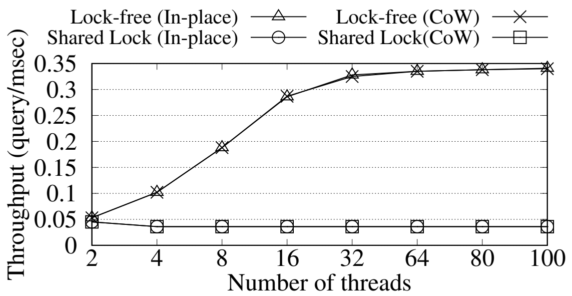
In the experiments shown in Figure 10(a), we increase the number of concurrent threads that insert 80 million polylines. As the number of concurrent insert transactions is increased, insertion throughput decreases due to lock contention. Note that the crabbing protocol we use for insert transactions must hold an exclusive lock on a parent node including the root node until its child node splits or it determines that its child node has sufficient free space such that a split is not required. Therefore, due to lock contention in upper-level tree structures, the crabbing protocol does not scale because it fails to benefit from high parallelism. As a result, for all schemes, insertion throughput decreases as the number of threads is increased. Note that the lock-free search implementation also uses the exclusive lock for insert queries. Therefore,



(a) Concurrent Insert Throughput



(b) Concurrent Search Throughput



(c) Concurrent Mixed Throughput

Figure 10: Performance with Varying Number of Concurrent Transactions (AVG. of 5 Runs)

the performance of lock-free search implementation, denoted as `Lock-free`, is not very different from the performance of shared lock implementation, denoted as `Shared Lock`. We note that the performance difference between in-place updates and CoW decreases as the number of threads increases. This is because as the number of threads increases, queries suffer from more serious lock contention and it becomes a dominant performance factor rather than the memory copy operations.

In the experiments shown in Figure 10(b), we measure the range query throughput with varying the number of concurrent range queries. In this experiments, there is no write transaction. Therefore, both the lock-free and shared lock implementations of `CoW` and `In-place` demonstrate good scalability up to 32 threads because the testbed machine has 40 cores. Note that performance scalability becomes saturated when we run more than 64 transactions. Interestingly, the lock-free implementation shows a similar performance with the shared lock implementation. This is because if there is no write transaction, shared lock implementation does not block any read transaction as in lock-free implementation. I.e., there is no lock contention between read transactions.

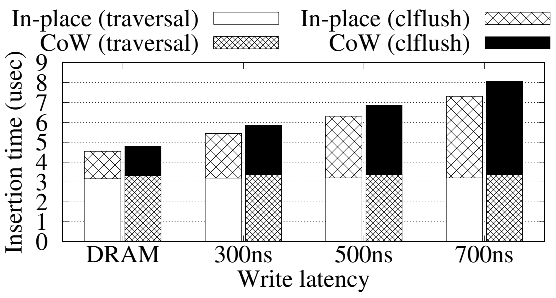
The range query throughput is much lower than the insertion throughput, although the read transaction does not include `clflush` overhead. This is because a multidimensional range query traverses a large number of tree nodes up and down, whereas the number of tree nodes that a write transaction visits is fixed as the tree height.

Figure 10(c) shows transaction throughput when the workload is mixed with both reads and writes, which is a more realistic scenario. In the mixed workload, our lock-free search algorithm shines. In the experiments, each thread alternates between three insert queries and seven search queries. The lock-free implementations of `CoW` and `In-place` gains up to 9.4x higher throughputs (338 txn/sec vs. 36 txn/sec), respectively, when the number of concurrent threads is 80. This is because a write transaction is blocked if any read transaction holds a shared read lock on the node that the write transaction wants to update. Since our workload runs multiple threads that submit queries in a batch, the blocked write transaction also blocks subsequent read transactions in the same thread. Therefore, the shared lock implementations do not scale with a larger number of concurrent threads. Although shared lock implementations show comparable performance when the workload is read-only or write-only transactions, the shared lock implementations suffer from lock contention when read and write transactions are mixed. The throughput of the shared lock implementation obtained with 100 threads (36 txn/sec) is similar to when we run only two threads for read-only transactions (32 txn/sec).

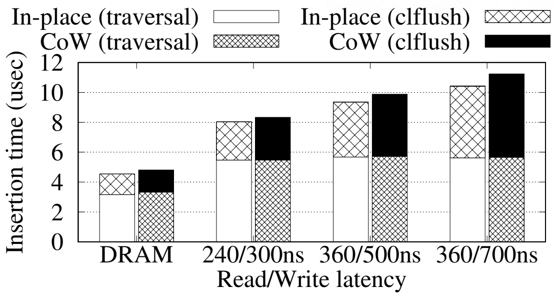
Note that these experiments not only evaluate the concurrency but also show the instant recoverability of FBR-tree. In the experiments, search queries concurrently access partially updated inconsistent tree nodes and return correct results. That is, even if a system crashes and partially updated inconsistent tree nodes are stored in persistent memory, the search algorithm of FBR-tree can return correct search results, i.e., it guarantees recoverability. We run a large number of search queries while write transactions, which make various tree nodes transiently inconsistent, are often suspended by the OS. If concurrent read transactions can construct a consistent view of index even if there exist some transient inconsistent tree nodes, recovery is not even necessary and the system can instantly recover. Using Optane DCPM, we also performed physical power-off tests and verified that partially updated FBR-tree nodes do not affect the invariants of index.

5.3 PM Latency Effect

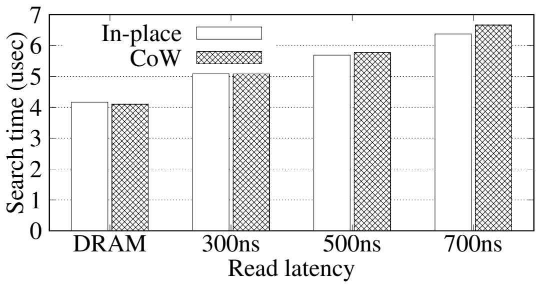
Although Intel's Optane DCPM is on the market, it is not the only emerging byte-addressable persistent memory technology, but other emerging byte-addressable PM technologies, such as STT-MRAM [3] and PCM technologies [2] are expected to offer a large performance spectrum. For example, it has been reported that STT-MRAM can be optimized to be even faster than DRAM [55]. Therefore, we use Quartz, a DRAM-based PM latency emulator [56, 57] to vary the PM latency when measuring the performance of FBR-tree. We note that Quartz has been used in numerous previous studies [15, 16, 21, 22, 24, 25, 48, 58, 58, 59]. Quartz models PM latency by inserting stall cycles at the boundaries of a small time interval called *epoch*. In our experiments, the minimum and maximum epochs are set to 5 and 10 nsec, respectively. We assume that PM bandwidth is the same as that of DRAM because Quartz does not allow us to emulate latency



(a) Write Latency Insert Time



(b) Read/Write Latency Insert Time



(c) Read Latency Search Time

Figure 11: Insertion Time with Varying PM Latency (AVG. of 5 Runs)

and bandwidth simultaneously.

In the experiments shown in Figure 11, we insert 80 million polylines in batches and breakdown the insertion time spent on each query as the read and write latencies of PM vary. `(traversal)` denotes the tree traversal time, which includes the MBR computations and LLC misses caused by node visits, and `clflush` denotes the time to update PM, i.e, the overhead of `store`, `mfence`, and `clflush` instructions. In Figure 11(a), we set the read latency of PM to that of DRAM but increase the write latency. Therefore, tree traversal times are unaffected by PM write latency; however, the cacheline flush overhead increases as PM write latency is increased. `In-place` calls fewer cacheline flushes than `CoW`; thus, the performance gap between `In-place` and `CoW` is widened up to 9% due to the difference in flush overhead.

In the experiments shown in Figure 11(b), we vary PM read latency using Quartz in addition to PM write latency. The read latency of the local node memory in our testbed machine is approximately 100 nsec. The average insertion time increases as we increase both PM latencies. Interestingly, insertion performance is more sensitive to PM write latency than read latency due to the CPU cache effects and because comparing MBRs in the multidimensional index requires a large number of CPU cycles.

In the experiments shown in Figure 11(c), we generate synthetic range queries in uniform distribution and submit 10,000 queries in a batch. The average selection ratio of the range queries is set to 1.9%. Note that we do not show the results of other selection ratios because no critical differences are observed. As we increase the read latency of PM, the query latency also increases; however, this does not result in a difference in the relative performance of the two split methods.

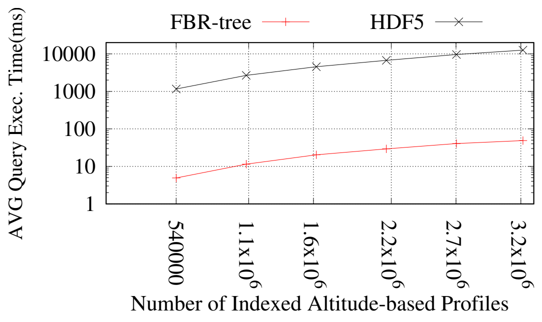
5.4 Case Study: Indexing HDF-EOS Datasets

To help in accessing large scientific datasets, various self-describing scientific data file formats, such as NetCDF [60] and HDF [61], have been developed. Self-describing scientific data formats allow applications to navigate through the file efficiently by providing semantic information about the dataset stored in the file and by allowing for direct access to particular data points using the semantic information. HDF-EOS is an extension of HDF library designed for NASA's Earth Observing System Data and Information System (EOSDIS). HDF-EOS allows for the construction of satellite specific data structures, called *grids*, *swaths*, and *points* [62].

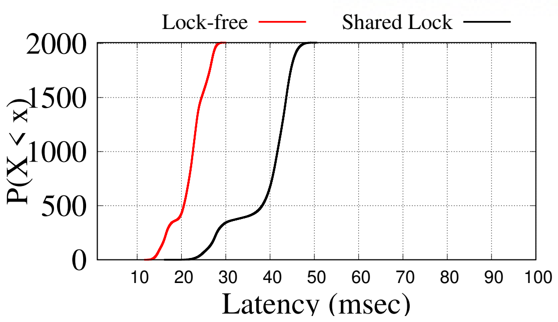
HDF-EOS provides the `defboxregion()` and `extractregion()` functions, which allows a user to specify a query range and to read data points from the query region of interest. For example, if a scientist looks for sensor data, for example, "find the tropopause pressure of the ozone layer where altitude $>$ 200 km and zenith temperature $>$ 30", `defboxregion()` scans semantic metadata in HDF files and `extractregion()` returns data points of the region. Note that HDF-EOS library does not use spatial indexing structures but performs linear scanning. To improve the performance of range query on HDF-EOS datasets, several studies [63–65] have developed spatial indexing techniques for HDF files. In particular, HDF5-FastQuery [65] was shown to outperform HDF-EOS range query by a factor of two [65].

In the experiments shown in Figure 12(a), we measure and compare the performance of FBR-tree against the standard HDF-EOS range query functions, i.e., linear scanning. For this experiment, we used HDF-EOS L1B solar event transmission datasets collected from SAGE (Stratospheric Aerosol and Gas Experiment) III/ISS instrument mounted on the International Space Station. SAGE III instrument measures the attenuation of solar radiation to understand the Earth's atmosphere and ozone depletion. The dataset consists of total 3,216,200 altitude-based profiles, and the total dataset size is approximately 6 GBytes. We varied the number of indexed altitude-based profiles in SAGE and measured the average range query execution time. Because we do not run insert query for this experiment, we do not show the performance of the shared-lock implementation. We see from Figure 12(a) that the average query execution time of FBR-tree is consistently three orders of magnitude faster than that of HDF-EOS range query function, i.e., 4.9$\sim$48.7 msec vs 1.3$\sim$12.2 sec).

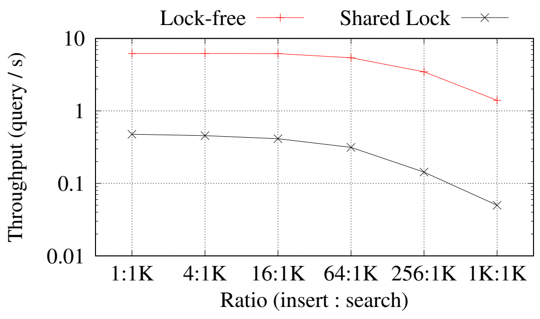
SAGE III/ISS creates one HDF file per a solar event, such as solar occultation. For each event, 200 altitude-based profiles are created. If we employ the shared-lock implementation of FBR-tree, concurrent search queries will be temporarily blocked to insert new profiles into the



(a) Time to read selected regions of dataset



(b) Tail latency of range queries when a solar event occurs



(c) Throughput for range query

Figure 12: Range Query Performance with SAGE III/ISS HDF-EOS Dataset

index whenever a new solar event occurs. For the experiments shown in Figure 12(b), we generated query inter-arrival patterns using Poisson distribution where the λ rate is set to the average query processing throughput of FBR-tree. We measured the latency of range queries while one million range queries arrive in Poisson distribution and one write thread simultaneously inserts a single solar event (200 profiles) into the FBR-tree, i.e., the insert/search ratio is 2:98.

We see from Figure 12(b) that approximately 30% of range queries suffer from the lock contention when the shared-lock implementation is used. In contrast, the lock-free implementation keeps the 99th percentile tail latency three orders of magnitude smaller than that of the shared-lock implementation. Note that we observe a small number of queries have very high latency even if the lock-free search algorithm is used. This is because the range queries happen to access tree nodes that are being updated; thus, they rollback the previous sub-tree traversal, reread the parent node, and visit the child nodes again, which increases the tree traversal time significantly.

In the experiments shown in Figure 12(c), we ran 64 concurrent threads with varying the ratios of search and insert transactions. When the insert query accounts for only 0.1% of total transactions (1:1K), the throughput of lock-free search is an order of magnitude higher than that of the shared-lock implementation. Note that the performance gap widens as the percentage of write transactions increases.

VI Conclusion

In this study, we have designed and implemented Failure-atomic Byte-addressable R-tree (FBR-tree) to obtain the most benefit from byte-addressability and the high-performance of PM. We carefully control the order of store and cacheline flush instructions and prevent single store instructions from making the FBR-tree inconsistent. Our performance study demonstrates that the FBR-tree reduces legacy logging overhead. In addition, the lock-free range query algorithm shows up to 9.4 times higher query processing throughput than the shared lock-based crabbing concurrency protocol. We also show that our FBR-tree on PM improves the performance of range query on HDF datasets by three orders of magnitude against the standard HDF-EOS range query functions.

References

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

[2] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, 2010.

[3] Y. Huai, "Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects," *AAPPS bulletin*, vol. 18, no. 6, pp. 33–40, 2008.

[4] R. Fang, H.-I. Hsiao, B. He, C. Mohan, and Y. Wang, "High performance database logging using storage class memory," in *Proceedings of the 27th International Conference on Data Engineering (ICDE)*, 2011, pp. 1221–1231.

[5] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via JUSTDO logging," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages (ASPLOS))*, 2016.

[6] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "High-performance transactions for persistent memories," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016, pp. 399–411.

[7] A. Rudoff, "Programming models for emerging non-volatile memory technologies," *;login*, vol. 38, no. 3, pp. 40–45, June 2013.

[8] P. Sehgal, S. Basu, K. Srinivasan, and K. Voruganti, "An empirical study of file systems on NVM," in *Proceedings of the 31st International Conference on Massive Stroage Systems (MSST)*, 2015.

[9] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger, "Metadata efficiency in versioning file systems," in *Proceedings of the 2nd USENIX conference on File and Storage Technologies (FAST)*, 2003, pp. 43–58.

[10] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *9th USENIX conference on File and Storage Technologies (FAST)*, 2011.

[11] J. Xu and S. Swanson, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[12] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 421–432.

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

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

[15] J. Huang, K. Schwan, and M. K. Qureshi, "Nvram-aware logging in transaction systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 4, 2014.

[16] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won, "NVWAL: Exploiting NVRAM in write-ahead logging," in *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[17] E. Lee, H. Bahn, and S. H. Noh, "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)*, 2013.

[18] G. Oh, S. Kim, S.-W. Lee, and B. Moon, "SQLite optimization with phase change memory for mobile applications," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 8, no. 12, pp. 1454–1465, 2015.

[19] J. Ou, J. Shu, and Y. Lu, "A high performance file system for non-volatile main memory," in *Proceedings of the 11th European Conference on Computer Systems (EuroSys 16)*, 2016.

[20] Y. Zhang and S. Swanson, "A study of application performance with non-volatile main memory," in *Proceedings of the 31st International Conference on Massive Stroage Systems (MSST)*, 2015.

[21] J. Seo, W.-H. Kim, W. Baek, B. Nam, and S. H. Noh, "Failure-atomic slotted paging for persistent memory," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[22] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, “WORT: Write optimal radix tree for persistent memory storage systems,” in *Proceedings of the 15th USENIX conference on File and Storage Technologies (FAST)*, 2017.

[23] J. Yang, Q. Wei, C. Chen, C. Wang, and K. L. Yong, “NV-Tree: reducing consistency const for NVM-based single level systems,” in *Proceedings of the 13th USENIX conference on File and Storage Technologies (FAST)*, 2015.

[24] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, “Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Trees,” in *Proceedings of the 11th USENIX Conference on File and Storage (FAST)*, 2018.

[25] M. Nam, H. Cha, Y. ri Choi, S. H. Noh, and B. Nam, “Write-optimized dynamic hashing for persistent memory,” in *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, 2019, pp. 31–44. [Online]. Available: https://www.usenix.org/conference/fast19/presentation/nam

[26] P. Zuo and Y. Hua, “A write-friendly hashing scheme for non-volatile memory systems,” in *Proceedings of the 33st International Conference on Massive Storage Systems and Technology (MSST)*, 2017.

[27] P. Zuo, Y. Hua, and J. Wu, “Write-optimized and high-performance hashing index scheme for persistent memory,” in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA, 2018.

[28] M. Gowanlock and H. Casanova, “Indexing of spatiotemporal trajectories for efficient distance threshold similarity searches on the GPU,” in *28th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2015.

[29] B. Nam and A. Sussman, “A comparative study of spatial indexing techniques for multidimensional scientific datasets,” in *Proceedings of 16th International Conference on Scientific and Statistical Database Management (SSDBM)*, Jun. 2004.

[30] M. R. Vieira, P. Bakalov, and V. J. Tsotras, “On-line discovery of flock patterns in spatio-temporal data,” in *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, ser. GIS ’09. New York, NY, USA: ACM, 2009, pp. 286–295.

[31] F. Giannotti, M. Nanni, F. Pinelli, and D. Pedreschi, “Trajectory pattern mining,” in *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD ’07. New York, NY, USA: ACM, 2007, pp. 330–339.

[32] H. Jeung, M. L. Yiu, X. Zhou, C. S. Jensen, and H. T. Shen, “Discovery of convoys in trajectory databases,” *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 1068–1080, Aug. 2008.

[33] Z. Li, J. Han, M. Ji, L.-A. Tang, Y. Yu, B. Ding, J.-G. Lee, and R. Kays, "MoveMine: Mining moving object data for discovery of animal movement patterns," *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 4, pp. 37:1–37:32, Jul. 2011.

[34] P. Cudre-Mauroux, E. Wu, and S. Madden, "TrajStore: An adaptive storage system for very large trajectory data sets," in *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, March 2010, pp. 109–120.

[35] B. Nam, H. Andrade, and A. Sussman, "Multiple range query optimization with distributed cache indexing," in *Proceedings of the ACM/IEEE SC2006 Conference*, 2006.

[36] H. Andrade, T. Kurc, A. Sussman, and J. Saltz, "Efficient execution of multiple query workloads in data analysis applications," in *Proceedings of the ACM/IEEE SC2001 Conference*, Nov. 2001.

[37] T. Kurc, C. Chang, R. Ferreira, A. Sussman, and J. Saltz, "Querying very large multi-dimensional datasets in ADR," in *Proceedings of the ACM/IEEE SC1999 Conference*, 1999.

[38] S. Goil and A. N. Choudhary, "High performance multidimensional analysis and data mining," in *ACM/IEEE Conference on Supercomputing (SC)*, 1998, p. 21.

[39] F. Tauheed, L. Biveinis, T. Heinis, F. Schürmann, H. Markram, and A. Ailamaki, "Accelerating range queries for brain simulations," in *28th International Conference on Data Engineering (ICDE)*, 2012.

[40] X. Chen, Y. Wang, E. Schoenfeld, M. M. Saltz, J. H. Saltz, and F. Wang, "Spatio-temporal analysis for New York state SPARCS data," in *Summit on Clinical Research Informatics, CRI*, 2017.

[41] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1984.

[42] Y. Kwon, D. Nunley, J. P. Gardner, M. Balazinska, B. Howe, and S. Loebman, "Scalable clustering algorithm for N-body simulations in a shared-nothing cluster," in *Scientific and Statistical Database Management, 22nd International Conference, SSDBM 2010, Heidelberg, Germany, June 30 - July 2, 2010. Proceedings*, 2010, pp. 132–150.

[43] D. Morozov and T. Peterka, "Efficient delaunay tessellation through K-D tree decomposition," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2016, pp. 728–738.

[44] M. Goldfarb, Y. Jo, and M. Kulkarni, "General transformations for GPU execution of tree traversals," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013, pp. 10:1–10:12.

[45] V. Pascucci and R. J. Frank, "Global static indexing for real-time exploration of very large regular grids," in *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (SC)*, 2001.

[46] M. S. Warren, "2HOT: an improved parallel hashed Oct-tree N-body algorithm for cosmological simulation," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[47] T. Kurc, U. Catalyurek, X. Zhang, J. Saltz, M. Peszynska, R. Martino, M. Wheeler, A. Sussman, C. Hansen, M. Sen, R. Seifoullaev, P. Stoffa, C. Torres-Verdin, and M. Parashar, "A simulation and data analysis system for large scale, data-driven oil reservoir simulation studies," *Concurrency and Computation: Practice and Experience*, 2005.

[48] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory," in *Proceedings of 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2016.

[49] T. Wang, J. Levandoski, and P.-A. Larson, "Easy lock-free indexing in non-volatile memory," in *IEEE 34th International Conference on Data Engineering (ICDE)*, 2018.

[50] Intel, "PMDK: Persistent memory development kit," 2018, *https://github.com/pmem/pmdk.*

[51] H. E. Lab, "Memory Driven Computing." 2018, *https://www.labs.hpe.com/next-next/mdc.*

[52] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, Jan. 1991. [Online]. Available: http://doi.acm.org/10.1145/114005.102808

[53] A. Kogan and E. Petrank, "A method for creating fast wait-free data structures," in *17th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP)*, 2012, pp. 141–150.

[54] A. Silberschatz, H. Korth, and S. Sudarshan, *Database Systems Concepts.* McGraw-Hill, 2005.

[55] Y. Jin, M. Shihab, and M. Jung, "Area, power, and latency considerations of stt-mram to substitute for main memory," in *Proceedings of The Memory Forum*, 2014.

[56] H. E. Lab, "Quartz," 2018, *https://github.com/HewlettPackard/quartz.*

[57] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li, "Quartz: A lightweight performance emulator for persistent memory software," in *15th Annual Middleware Conference (Middleware '15)*, 2015.

[58] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[59] J. Arulraj, A. Pavlo, and S. R. Dulloor, "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, 2015, pp. 707–722.

[60] R. Rew, G. Davis, and S. Emmerson, "NetCDF User's Guide for C," 1997, *http://www.unidata.ucar.edu/packages /netcdf/cguide.pdf.*

[61] M. Folk, "A white paper: HDF as an archive format: Issues and recommendations," January 1998, *http://hdf.ncsa. uiuc.edu/archive/hdfasarchivefmt.htm.*

[62] Larry Klein, "An HDF-EOS and data formatting primer," March 2001, *http://edhs1.gsfc.nasa.gov/waisdata/sdp/pdf/wp1750102.pdf.*

[63] B. Nam and A. Sussman, "Improving access to multi-dimensional self-describing scientific datasets," in *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, May 2003.

[64] J. Chou, K. Wu, O. Rubel, M. Howison, J. Q. Prabhat, B. Austin, E. W. Bethel, R. D. Ryne, and A. Shoshani, "Parallel index and query for large scale data analysis," in *Proceedings of the ACM/IEEE SC2011 Conference*, 2011.

[65] L. Gosink, J. Shalf, K. Stockinger, K. Wu, and W. Bethel, "HDF5-FastQuery," in *Proceedings of 18th International Conference on Scientific and Statistical Database Management (SSDBM)*, 2006.