Persistent Memory Range Indexes and applications

Huayi Wang
Georgia Institute of Technology
huayiwang@gatech.edu

Kuilin Li
Georgia Institute of Technology
kuilin@gatech.edu

Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/likuilin/pmri.

1 INTRODUCTION

As the data size increases, range indexes become larger and larger. For a 1 billion size dataset, range indexes can be more than 200GB to support update, search and delete [1]. This huge size makes it impossible to fit the range index structure into memory. However, if we put this data structure on disk, this can negatively affect the performance of the search algorithm, because it is assuming that its working scratch memory is fast, not burdened by low I/O speeds on HDD.

To solve the scaling issue of range indexes, Non-Volatile Memory (NVM) technology has been widely studied for more than a decade as a useful solution to address the scaling problem of DRAM. For example, a dual-socket machine can be equipped with up to 6TB of NVM memory at a modest cost [12]. However, novel data structures algorithms are required to maximize performance on this new hardware model, because of its asymmetric read/write speeds. So, disk-resident data structures like B+-trees need to be re-implemented with NVM.

Recently, many researchers have proposed optimized implementations of B+-trees in NVM, such as FP-tree, BZ-tree and LB+-tree [4, 12, 14]. However, only one recent paper has comprehensively studied the B+-tree implementations on real, next-generation persistent memory [9], and it does not evaluate recent papers like [12] which is range index in persistent memory.

Furthermore, the range indexes in persistent memory have also been introduced to other database areas such as the similarity search algorithm. To solve the scaling problem of similarity search algorithm on large datasets, some researchers have also proposed to put the index of dataset in persistent memory for fast search on billion level datasets [16]. However, only few similarity search algorithms have tried to build their index structures on persistent memory such as [16, 22]. Many popular and widely used similarity search algorithms like multi-probe LSH [13] and min-hash [5] have not been proposed to extend on persistent memory.

Based on the lacking of enough evaluation of range indexes in persistent memory and their applications in other database areas, we plan to study these implementations and evaluate these range index indexes in this project. We will mainly focus on studying how these PM range indexes deal with byte-addressability and read-write asymmetry. Also, we plan to improve on previous range indexes based on our study of the implementations and evaluation results. Furthermore, we will study the possibility of extend the persistent memory range indexes to similarity search algorithms. We will focus on extending a widely-used similarity algorithm multi-probe LSH to persistent memory and optimize its performance in persistent memory.

2 BACKGROUND AND RELATED WORK

2.1 Persistent Memory

Intel Optane DC Persistent Memory is the first NVM main memory solution available for mainstream computer systems [2]. The product is packaged in the NVDIMM format and plugs into standard DDR4 DIMM sockets. It uses the proprietary DDR-T protocol, which revises the DDR4 protocol to support asynchronous operations. The capacity of an NVDIMM is typically 128GB — 512GB. A dual-socket machine can be equipped with up to 12 NVDIMMs or up to 6TB of NVM memory.

There are two main persistent memory configuration modes, as shown in Figure 1. In the memory mode, DRAM is managed as a cache for persistent memory by the memory controller. The main memory capacity is equal to the total size of the persistent memory. This mode enables applications to exploit the large capacity provided by persistent memory. However, there are two drawbacks for the memory mode. First, the memory mode does not support persistent data structures. Therefore, it can only be used as large volatile main memory. Second, it takes longer time to load data from persistent memory because a load consists of a DRAM cache visit followed by a persistent memory visit. Hence, it may be sub-optimal to run data-intensive applications with working sets larger than the DRAM cache capacity in the memory mode. For this reason, most range indexes does not use this memory mode.

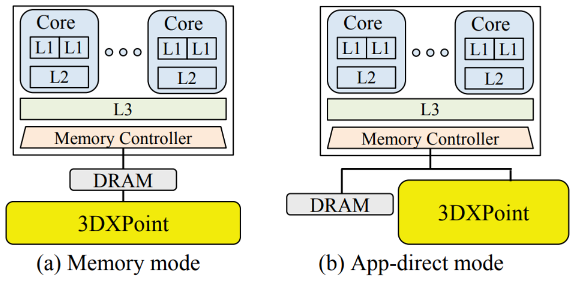


Figure 1: Persistent Memory modes

In the app-direct mode, persistent memory and DRAM are both directly accessed by the CPU. Persistent memory modules are recognized as special devices by the system. We can install file systems on persistent memory modules and use PMDK (Persistent Memory Development Kit) [3] to map a file from persistent memory into the virtual memory space of an application. The system runs DAX device drivers for persistent memory so that accesses to persistent memory get around the system page cache. In this way, applications can directly access persistent memory using load and store instructions, and implement data structures in persistent memory. Many B+-tree structures in persistent memory are using this app-direct mode, our evaluation will also focus on the range indexes using app-direct mode. So PMDK will be the main developing tools in this project. Furthermore, since app-direct mode can access CPU directly, our design of similarity algorithms in persistent memory will also focus app-direct mode.

2.2 Range Indexes in Persistent Memory

Since persistent memory devices have limited write performance and endurance, range indexes in persistent memory focus on reducing the number of stores to persistent memory. wB+-Trees [6] proposes to use an indirection slot array and/or a bitmap so that most insertions and deletions do not require the movement of index entries and adopts a redo-only logging algorithm for ensuring durability. In contrast, NV-Tree [21] employs an append-only update strategy and re-constructs internal nodes during recovery. FP-Tree [14] is a hybrid DRAM-PM index that keeps the internal nodes in volatile memory and stores the leaf nodes on PM. Bz-Tree [4] is a latch-free B+-Tree for PM that uses persistent multi-word compare-and-swap (PMwCAS) [19] to achieve concurrency and make the implementation easy. The recent LB+-Trees [12] nodes are designed to be 256B or a multiple of 256B, since 256B is the internal data access size in Intel Optane persistent memory. Also, it introduces two techniques: entry moving to reduce the number of writes for insertions when creating empty slots and logless node split which use the NVM atomic write to reduce logging overhead.

2.3 Similarity Search Algorithms

Similarity search is a fundamental algorithmic problem in database, with numerous applications in many areas of computer science, including informational retrieval [11], recommendations [15], near-duplication detections [18], etc. The problem is defined as follow: given a query object q, we search in a massive dataset $\mathcal{D}$, for one or more objects in $\mathcal{D}$ that are *among the closest* to q according to some similarity metric.

One of the most popular similarity search algorithms is multi-probe LSH [13], which solves the large index problem arised in LSH [7]. Due to multi-probe LSH's spectacular efficacy, it has been widely deployed in various systems including smartphone applications [17], audio content retrieval [23], automatic product suggestions [8], etc. However, as the size of dataset increases, the size of a dataset with 1 billion points can be as large as several hundreds gigabytes [16]. Such huge raw data size makes multi-probe LSH hard to fit in DRAM as it needs to store the raw data in memory for efficient search, even though the index sizes (excluding raw data size) of multi-probe LSH for a dataset of one billion points is approximately 24GB. We believe that the fast reading speed of persistent memory can help addressing the scaling problem of raw data size in multi-probe LSH.

2.4 PMwCAS

PMwCAS [20] is a library that allows allows for multi-word compare and swaps. Usually, with one CAS (compare-and-swap), you can atomically compare a memory address with what you expect the contents to be, and, if they are equal, swap it, if they are not, then nothing happens. This happens transactionally, so the entire contents either change or nothing is changed. What PMwCAS adds is persistent memory multi-word CAS. Once a "descriptor pool" is allocated, we can call `desc->AllocateDescriptor()` to create a descriptor, and then add any number (up to a compile time constant) of "entries" to the descriptor. Each entry consists of an address, expected value, and new value, and means one CAS for a uint64_t. This is implemented by `desc->AddEntry(uint64_t *addr, uint64_t expected, uint64_t newvalue)`. Then, when `desc->MwCAS()` is called, all of these compare and exchanges happen at the same time, transactionally - if any of the addresses do not contain their expected values, no memory is modified and the entire thing fails gracefully. Otherwise, all of the addresses are updated to their new values. BzTree uses this as a base structure to handle all of its thread-safety requirements. This is lock-free because PMwCAS uses persistent memory operations, instead of locks, itself.

3 ISSUES IN IMPLEMENTATION

In this section, we briefly discuss the issues we find when implementing Bz-Tree in this project. And we will also discuss about the method we use to solve these problems.

3.1 The PMwCAS Issue

The first problem we encountered is that when we attempted to compile and run the Microsoft implementation, we ran into problems related to a mysterious segfault inside the library. So, instead, we decided to begin implementing our own PMwCAS library. Unfortunately, we figured that it would be too much of a workload to implement this ourselves in addition to the BzTree part of the project. Additionally, since BzTree only uses PMwCAS for its guarantees, and nothing else specific to NVM, if we implemented both ourselves, we would not be exactly be evaluating the efficacy of any BzTree improvements we make in a way that is easily distinguishable from the performance of our PMwCAS implementation, which may be worse than the Microsoft implementation. Therefore, we decided to revisit using the library.

After building the library ourselves and then tracing the error (specifically, a segfault on the address `0xffffffffffffffff` inside the library), we found that it was because of a failed mmap syscall in the allocation of the emulated persistent memory, and mmap returns -1 as the address if the allocation fails. Specifically, it calls the following code with $size\_ = 0x4000360$:

mmap(base_address, size_, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_LOCKED, map_fd_, (off_t)0)

This took a long time to figure out because one of the first things we did to diagnose this issue is strace the executable (back when we were still using this as a compiled library, instead of looking at the source) to see the mmap call, and then we tried to mmap with MAP_SHARED|MAP_LOCKED flags in our own binary with a size of $0x4000000$, and it worked, so we did not think the issue was the mmap. However, the mmap syscall the library actually calls is $0x4000360$, because it needs some extra persistent memory metadata in addition to the data it reserves for the descriptor pool. And, unfortunately, any mmap with MAP_LOCKED on our local system fails at any size above $0x4000000$, which was just unlucky.

Now that we have isolated the issue, we can actually fix it without modifying the library code at all. The emulation layer is executed as a separate binary as a "shim server" to persist memory using

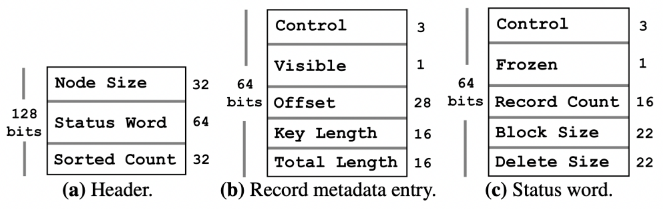


Figure 2: Node layout and details for the BzTree

Tree Operation	PMwCAS Size
NODE OPERATIONS	
Insert [Allocation, Completion]	2, 2
Delete	2
Update [Record Pointer, Inlined Payload]	3, 2
Node Consolidation	2
SMOs	
Node Split [Preparation, Installation]	1, 3
Node Merge [Preparation, Installation]	2, 3

Figure 3: The size of the PMwCAS operations associated with modification operations.

ordinary means even after the executable using the memory crashes. This shim server accepts command-line agruments, including one to change the size of the descriptor pool, which is default 262144. Changing this to a slightly smaller 262000 makes the mmap syscall succeed and everything work.

While we were diagnosing this issue, we found a small bug in the PMwCAS library and submitted a pull request as shown in this link https://github.com/microsoft/pmwcas/pull/11.

3.2 The height issue

After we made the PMwCAS library works, we started to begin our implementation. While we were implementing BzTree, we found a concurrency issue in the original paper's description of the root data structure for the tree: the height issue. The BzTree paper doesn't discuss about how the height of the tree is stored. There isn't any field in the node layout that tracks whether the node is an inner node or child node, as shown in Figure 2 copied from the paper, so we must rely on this height field to determine whether a node is an inner node or a leaf. Since we do need to know if a node is an inner node or a child node during tree traversal, we consider to track the height of Bztree by having a variable for the tree about the height. Then each time the tree is traversed, we can just count the dereferences to children. However, BzTree paper [4] doesn't say this. They mention that the global index epoch and root node pointer are the only things that need to be saved.

We note that we cannot naively save and update the height, since nonzero time must pass between when we do that and when we actually update the root pointer, which means another thread can read an incorrect height that is off by one depending on if we're merging or splitting. In this project, we consider three ways to solve this height issue:

(1) We can incur an extra PMwCAS on the height and the root pointer to update the height.
(2) We can change the node data structure to add an additional bit "`is _child` ".
(3) We can add a new data structure to solve this issue.

In our implementation, we find that the first and second solutions does not work very well because these solutions will incur an extra PMwCAS to update the height. This extra PMwCAS might have large impacts on performance since the size of PMwCAS operations are very small in the BzTree modification operations as shown in Figure 3.

So the final solution we use in our implemenation is we add a new data structure called *root metadata* called `BzPMDKMetadata` data structure in our implementation. This data structure only contains a pointer to the root node and the height of the tree. When the root needs to be split or merged, instead of directly writing to the root persistent memory object, we create and write a new root metadata structure with the new root pointer and the new height, and then swap the pointer into the root object, which is the root persistent memory object. Since this is only an one-pointer (and thus one-word) assignment, it is atomic.

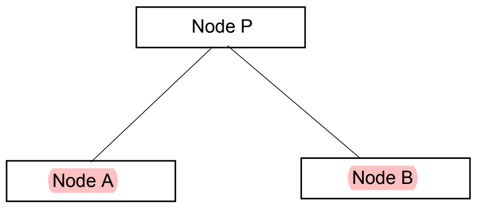
3.3 Node merge issue

The third issue we have found in the implementation is that the original BzTree paper says that during a node merge, the two children are the only things that are frozen. But we find that it is best to freeze the parent of as well. We use the following figure Figure 4 to describe this issue. Let node A and node B be the two children nodes of node P. Original BzTree paper describes that in node merge, only node A and node B are frozen. But if a concurrent split of the node P makes the node A and node B have two different parent node P and node P' cannot be checked since we do not have a reference to the other parent. For example, node B only has the reference to node P but no reference to node P'. We can check if a child belongs to a particular parent by re-scanning all the keys but this operation cannot be done atomically. Since node merging is more costly than other forms of node manipulations, such as splitting (very unlikely to happen to a node undergoing a merge) and compaction, and since once it begins we already know it must either modify the parent to succeed or yield, it makes sense for node merging to have the highest "priority" among operations that can freeze inner nodes, and therefore freeze the parent when they begin.

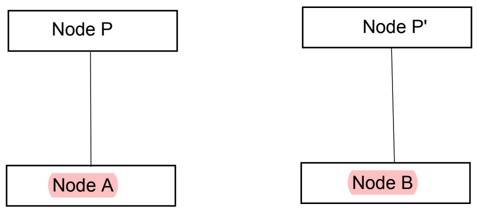
In our implementation, we choose to freeze the parent nodes of two children similar to the node split in preparation step as described in [4]. We hope this improvement can make our BzTree implementation more robust to node operations.

4 DESCRIPTION OF IMPLEMENTATION

In this section, we briefly describe how we implemented the BzTree based on our understanding of the original BzTree paper [4] and our solutions to the problems we found in the original BzTree as described in the preceding section. More details of our implementation can be found in the Github repository https://github.com/likuilin/pmri.



(a) Node merge



(b) An concurrent node split

Figure 4: Node Merge with a concurrent node split.

4.1 Interface

Our BzTree interface is compiled to a shared object `libbztree` which other applications, including our testbench, can link to. The application interface is entirely contained within one public class, the `BzTree` class. This allows the implementation details to be hidden from any consumers of this library, and streamlines comparing this library against others which implement a similar interface.

At compile time, there are several constants that can be defined that affect the operation of our BzTree:

- `BZTREE_NODE_SIZE` is the size in bytes of each node in the BzTree. Each node requires a 16 byte header, 16 bytes of metadata per key-value pair, and a variable amount of space for the actual key-value data. This is implemented in accordance with the original BzTree paper.
- `BZTREE_MIN_FREE_SPACE` is the heuristic for how much free space a node must have at minimum. It should be about 10-20% of the node size. When a node's free space (specifically, the area between the fixed-size headers list and the variable-size data heap) decreases to below this threshold, the node will be a candidate for the split SMO, described in the following subsection. Any non-read traversal of the heap will then perform this SMO first before proceeding in the traversal, in accordance with the original BzTree paper. Additionally, this is also the threshold that determines how often the merge SMO occurs. If two sibling nodes can be merged such that the resulting node has less space than the minimum, then a merge occurs. This is less-than-optimal because it results in more splits and merges, so as a further optimization, separating this into two variables may greatly increase performance.
- `BZTREE_MAX_DELETED_SPACE` is the heuristic for how much deleted space a node can have at maximum. Deleted space consists both of variably-sized holes in the variable-size data heap at the end of the node (which are marked by not having a pointer pointing at them) and by deleted entries in the node metadata header array at the beginning of the node (which are marked by having their visible bit set to zero). Whenever data is deleted from the node through a key delete or update, the node's record for deleted space is updated, and when that reaches this defined threshold, the node is a candidate for the compaction SMO, described in the following subsection.
- `DEBUG_PRINT_ACTIONS` is a debug flag which prints every action that the BzTree takes on the internal state
- `DEBUG_PRINT_SMOS` is a debug flag which prints every SMO that the BzTree tries to begin during the traversal step of non-read operations.

When a BzTree is instantiated, it uses the global PMwCAS allocator to allocate space for it into the root node of the current persistent memory data structure. So, the constructor does not require any arguments. After it is instantiated, there are five class methods that can be called:

- `bool insert(string key, string value)` inserts a key-value pair into the BzTree. It returns false upon failure, if there is already a value with that key.
- `bool update(string key, string value)` atomically updates a key-value pair's value. It returns false upon failure, if there is no such key.
- `optional<string> lookup(string key)` looks up a particular key and returns `std::nullopt` if it doesn't exist, or the value if it does. This API call is guaranteed to return in O(log(n)) time, since it does not invoke any SMOs, so it only requires one dereference per tree level to find a result.
- `bool erase(string key)` erases a key from the BzTree. It returns false upon failure, if there is no such key.
- `void destroy()` destroys the BzTree, un-allocating from physical memory all the resources it allocated. This is required, and explicitly not the destructor, because when the application exits or crashes or loses power, the data structure is still guaranteed to be persistent, since it is on persistent memory. So, there needs to be an out-of-band way to explicitly de-allocate the BzTree.

4.2 Garbage Collection

Our BzTree borrows the epoch-based garbage collection method implemented by PMwCAS. At any time, a thread can choose to enter an epoch using the Protect public call. Then, until it calls Unprotect, it is guaranteed to be able to dereference any pointers it finds while dereferencing anything in its view of the data structure.

If any data needs to be unallocated, unless no pointer to the data has ever touched any space traversable from the root node (PMwCAS can transactionally guarantee this upon failure), it must instead be passed to the garbage collector's Add method. This will add the pointer to the list of items un-allocated during the current epoch.

Every time Protect is called, a new epoch is entered, where the items unallocated during the epoch are guaranteed to persist until the epoch ends. When Unprotect is called, the garbage collector traverses its internal data structures to unallocate any pointers for which all of their entered epochs have exited.

4.3 Data Structures

Here, we go more in detail about our particular implementation of BzTree, particularly with regards to how it differs from the reference BzTree paper's implementation.

The root PMDK object in the persistent memory pool is `BzPMDKRootObj` which only contains a persistent memory pointer (which we will call TOID, see the next section Type Safety for why) to a global metadata object `BzPMDKMetadata` and the descriptor pool, a handle to the global PMwCAS object it uses for metadata. We need this layer of indirection at the root because of the height problem, described in a previous section. In summary, multiple views of the BzTree can have different heights, so we need to atomically update a pointer to a data structure containing (height of tree, pointer to root node) to be completely safe. This data structure is the `BzPMDKMetadata` struct.

The `BzPMDKMetadata` struct contains a TOID to the root node (of struct `Node`) of the tree, and the height and global epoch counter. The height of the BzTree is defined as the number of levels the tree has, so height minus one dereferences are required to get to any leaf node. The global epoch counter is described in the original BzTree paper - it is used to resolve edge concurrency cases relating to a crash of the system during an in-process update. It is only incremented by one every time the database restarts from a crash.

The `Node` struct is a struct that implements the same node layout as the original BzTree paper. The size of this struct is defined using the `BZTREE_NODE_SIZE` define, and inside that size, there is a fixed size node header and a body. Assuming that there are n key-value pairs stored in the node (including deleted pairs), the first n blocks of 16 bytes in the body consist of fixed-size `NodeMetadata` objects in a list, and the last n blocks of variably-sized bytes in the body consist of dynamically allocated variable-length key and value strings. These keys and values are fixed width, with offset and length parameters identified in the `NodeMetadata` struct that includes them.

When a key-value pair is deleted from a `Node`, the only thing that occurs is that the visible bit in its metadata struct is set to zero, and then the deleted space counter is updated to add the size of the released metadata struct and key-value memory. The actual data remains in the heap, and is only unallocated when a node compaction SMO occurs, which is triggered by the deleted space counter reaching a certain threshold defined by the `BZTREE_MIN_FREE_SPACE` heuristic.

4.4 Concurrency

Concurrency-stability is guaranteed for node operations in the same way the BzTree describes. Each node is either frozen or unfrozen, and, once a node is frozen, it is guaranteed to not be modified again until it is removed from the tree. If a node is reachable from the root node, then all threads modifying the node must PMwCAS the frozen bit in the status word, either to set it, or to ensure that it was not set. To encapsulate this requirement, in our implementation, the only operation that modifies a non-frozen node is the `swap_node` operation, described below. If a thread intends to deallocate a node, it should set the frozen bit first so other threads do not try to write new data into the node. If a PMwCAS fails due to writing to a frozen node or a failed contest for the frozen bit, the node is re-traversed.

4.5 Structural Modifications

Structural modifications, or SMOs, are internal operations that the BzTree implements to keep the tree healthy. They do not change the key-value pairs stored in the BzTree from the perspective of the outer interface, and so theoretically can happen at any time. These SMOs are described in the original BzTree paper, but here we will go into detail on how we implemented these on our trees.

The original paper makes a heuristic claim that insertion will be fast, because SMOs will be performed at an adequate frequency. However, for ease of implementation, we chose to make an explicit guarantee that, if all keys and values are sized smaller than the minimum free space heuristic, insertion will always succeed. This simplifies our API such that the only reason insertion could fail is if the node already exists.

This guarantee is implemented by the fact that, while traversing the tree looking for the child in which to insert our key value pair, we first perform any SMOs that traversed nodes require. So, upon dereferencing a new (grandparent, parent, child) pair, we do not need to be concerned with updating or even maintaining a reference to any nodes above the grandparent in the ancestor chain, because the parent is guaranteed to have enough space to store an adequately small key if the child splits, the parent inner node can accommodate the new key value pair.

The three structural modifications we implemented are as follows:

- Node compaction: When a node's deleted space reaches a certain threshold, the node is selected to be compacted. The visible keys and values of the node are iterated and copied to a new node, and then the node's parent is atomically updated with a reference to this new node. Since nodes do not look at non-visible key value pairs in their body, this exchange does not affect any other node.
- Node splitting: When a node's free space reaches a certain threshold, the node is selected to be split. Before a node is split, it is first checked for compaction, and compaction takes priority. This is because if a node that does not need compaction or splitting suddenly needs both compaction and splitting, generally, compaction will solve the problem of having too little free space, so a split is not necessary most of the time. Since compaction is a much less expensive operation than splitting, compaction is performed first and then the node is re-evaluated.
  The node split proceeds in the way the original BzTree paper describes. After freezing both the parent node and the child to be frozen, first, a suitable key is chosen as the split key (based on the sizes of the keys and values, we choose a key that is geometrically halfway into the key-value heap so the sizes are as even as we can get them). Next, three nodes are created, a new parent which contains the two children where there was one, and the two new children. Finally, a pointer swap in the grandparent is done to install the new parent.

- Node merging: When a node's free or deleted space is so few that it can merge with a sibling and still have more free space than the node splitting threshold, it is merged. First, both the parent of the siblings and the siblings are merged. Then, a new single child is constructed with all the keys and values, and then a new parent is constructed with one less child. Finally, a pointer to the new parent is atomically exchanged into the grandparent's data structure.

All three operations end with atomically exchanging a pointer into the parent or grandparent's data structure in order to make the changes visible. This common swap is abstracted in the internal helper function `swap_node` because it has some nuance to it. Specifically, this is the only operation that we do on a non-frozen node, in order to ease safety implementation.

This swap function performs a PMwCAS on both the frozen bit of the node and the new pointer, setting the frozen bit of the node to the same unfrozen state, and the new pointer in the child field of whatever value needs to be edited. The frozen bit must be included because otherwise, it is possible that we enter the `swap_node` function, and then another thread freezes the node and begins traversing the children. If a node split happens to the children of a child, it is possible that the traversal sees the right siblings of the result of the split twice, because it is expecting it to be frozen. Therefore, we must ensure that the node was not frozen before and after the swap.

5 EVALUATION

As we described in the proposal, we evaluate our BzTree under different workloads similar as [10]. In details, we evaluate our BzTree performance under three rounds of lookup, insert and delete operations. In each round, we have ten thousand number of insert, lookup and erase operations, respectively. Due to the time limitation, we only benchmarked the BzTree performance for uniformly distributed data with single threaded settings.

During the first round of insert, lookup and delete operations, only node splits occur. Because node merges only happens upon traversal of a node that needs merging, and erasing doesn't automatically trigger node merge. For the second and third rounds of operations, we find that the insert operations cause lots of node merging because the down traversals along the BzTree can find many under-filled nodes. We also find few compactions in our benchmark results which are prevalent when along key or value is updated, which is very rare in our benchmark. As for the insert rate, we find that our BzTree implementations need approximately 40 seconds for ten thousand keys insertion. We also find that the speed of insertion can be increased by using larger node size. For example, we find that using larger node size results in 10 seconds for ten thousand keys insertion. More details about the benchmark can be found in the benchmark branch in the github repository https://github.com/likuilin/pmri/blob/benchmark/benchmark_results.txt.

6 PREVIOUS GOALS ACHIEVEMENT

In this section, we briefly describe how much we have achieved in this project based on the previous goals and what we have learned from this project. Recall our previous goals for this project in the progress report are:

- 75% goal will be learning the PMwCAS library and re-implementing the BzTree data structures [4].
- 100% goal will be the evaluation of BzTree under different workloads by using the benchmark framework PiBench. We will develop figures of performance under different workloads and analyze the reasons for such difference in these workloads, such as why they chose to insert or update nodes in a particular way. Then based on our analysis, we hope that we can propose some improvements over current implementation of BzTree from our analysis and implement these improvements to see if they work well in real scenarios. We hope that we can compare our improved BzTree with previous BzTree ideas and check if this can boost the performance.
- The 125% goal will be the implementation on the persistent memory-based multi-probe LSH and evaluating its performance on datasets with one billion points.

For the first goal, we spent a huge amount of time to learn how PMwCAS library works and successfully re-implemented BzTree based on what we have learned from PMwCAS. We believe what we have learned from this project can help us programming in persistent memory in our future research.

As for the second goal, although we do not have thorough benchmark as we proposed, we do evaluate the performance of BzTree under different workloads. More importantly, we discover some potential issues of BzTree and we propose some solutions for these issues as described in §3. We hope to compare our new implemented BzTree against previous BzTree in real scenarios in the future work. We hope what we discovered in this project can shed light on the implementation of range indexes in persistent memory.

As for the 125% goal, we cannot implement the persistent memory-based LSH solution in this project due to the time limitation. But this project does help us understand more about persistent memory and how to write programs in this new hard-ware. Hopefully, we will improve the previous LSH solution by using persistent memory in near future based on what we have learned in this project.

REFERENCES

[1] [n.d.]. Datasets for ANN neighbor search. http://corpus-texmex.irisa.fr/.
[2] [n.d.]. Intel optane dc persistent memory architecture overview. https://techfieldday.com/video/intel-optane-dc-persistentmemory-architecture-overview/..
[3] [n.d.]. Intel. Persistent Memory Development Kit. http://pmem.io/pmdk..
[4] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. Bztree: A High-Performance Latch-Free Range Index for Non-Volatile Memory. *Proc. VLDB Endow.* 11, 5 (Jan. 2018), 553–565. https://doi.org/10.1145/3164135.3164147
[5] A.Z. Broder. 1997. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171).* 21–29. https://doi.org/10.1109/SEQUEN.1997.666900
[6] Shimin Chen and Qin Jin. 2015. Persistent B<sup>+</sup>-Trees in Non-Volatile Main Memory. *Proc. VLDB Endow.* 8, 7 (Feb. 2015), 786–797. https://doi.org/10.14778/2752939.2752947
[7] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *STOC*. ACM, 604–613.
[8] Yannis Kalantidis, Lyndon Kennedy, and Li-Jia Li. 2013. Getting the look: clothing recognition and segmentation for automatic product suggestions in everyday photos. In *ICMR*. 105–112.
[9] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating Persistent Memory Range Indexes. *Proc. VLDB Endow.* 13, 4 (Dec. 2019), 574–587. https://doi.org/10.14778/3372716.3372728

[10] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating Persistent Memory Range Indexes. *Proc. VLDB Endow.* 13, 4 (Dec. 2019), 574–587. https://doi.org/10.14778/3372716.3372728

[11] K. Lin, H. Yang, J. Hsiao, and C. Chen. 2015. Deep learning of binary hash codes for fast image retrieval. In *2015 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 27–35. https://doi.org/10.1109/CVPRW.2015.7301269

[12] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *Proc. VLDB Endow.* 13, 7 (March 2020), 1078–1090. https://doi.org/10.14778/3384345.3384355

[13] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *PVLDB*. 950–961.

[14] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 371–386. https://doi.org/10.1145/2882903.2915251

[15] L. Qi, X. Zhang, W. Dou, and Q. Ni. 2017. A Distributed Locality-Sensitive Hashing-Based Approach for Cloud Service Recommendation From Multi-Source Data. *IEEE Journal on Selected Areas in Communications* 35, 11 (Nov 2017), 2616–2624. https://doi.org/10.1109/JSAC.2017.2760458

[16] Jie Ren, Minjia Zhang, and Dong Li. 2020. HM-ANN: Efficient Billion-Point Nearest Neighbor Search on Heterogeneous Memory. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 10672–10684. https://proceedings.neurips.cc/paper/2020/file/788d986905533aba051261497ecffcbb-Paper.pdf

[17] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. 2011. ORB: An efficient alternative to SIFT or SURF. In *ICCV*. IEEE, 2564–2571.

[18] Sadhan Sood and Dmitri Loguinov. 2011. Probabilistic Near-Duplicate Detection Using Simhash. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management* (Glasgow, Scotland, UK) *(CIKM '11)*. Association for Computing Machinery, New York, NY, USA, 1117–1126. https://doi.org/10.1145/2063576.2063737

[19] 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

[20] 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

[21] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-Based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (Santa Clara, CA) *(FAST'15)*. USENIX Association, USA, 167–181.

[22] Zhili Yao, Jiaqiao Zhang, and Jianlin Feng. 2021. NV-QALSH: An NVM-Optimized Implementation of Query-Aware Locality-Sensitive Hashing. In *Database and Expert Systems Applications*, Christine Strauss, Gabriele Kotsis, A. Min Tjoa, and Ismail Khalil (Eds.). Springer International Publishing, Cham, 58–69.

[23] Yi Yu, Michel Crucianu, Vincent Oria, and Ernesto Damiani. 2010. Combining multi-probe histogram and order-statistics based LSH for scalable audio content retrieval. In *ACM MM*. 381–390. https://doi.org/10.1145/1873951.1874004