ART That Lasts: Persistent Multiversion Adaptive Radix Trees with Fast Atomic Range Queries

ARASH KHALAJI, David R. Cheriton School of Computer Science, University of Waterloo, Canada
TREVOR BROWN, David R. Cheriton School of Computer Science, University of Waterloo, Canada
KHUZAIMA DAUDJEE, David R. Cheriton School of Computer Science, University of Waterloo, Canada

Indexes are essential for efficient query processing in database systems, with ordered indexes particularly suited for supporting range queries. Non-volatile memory (NVM) technology offers fast, byte-addressable access and persistence across system failures, enabling rapid recovery without costly index reconstruction. While recent efforts have focused on building durable NVM-based indexes for point queries and updates, they often lack efficient support for concurrent range queries with correctness.

We present `PermART`, a persistent, versioned adaptive radix tree that surpasses current NVM-based indexes in point query and update performance and supports efficient, consistent, range queries under concurrency. By integrating multiversioning into ART, we enable efficient, linearizable range queries while improving point operation performance through in-node logging. Our evaluation shows that our persistent ART matches or exceeds the performance of current NVM-based indexes for point queries and updates, while offering correct, linearizable range queries, which existing indexes do not provide.

CCS Concepts: • **Information systems** → **Unidimensional range search**; **Data scans**.

Additional Key Words and Phrases: Persistent Memory, Non-Volatile Memory, Persistent Indexes

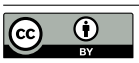
ACM Reference Format:
Arash Khalaji, Trevor Brown, and Khuzaima Daudjee. 2026. ART That Lasts: Persistent Multiversion Adaptive Radix Trees with Fast Atomic Range Queries. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 138 (June 2026), 26 pages. https://doi.org/10.1145/3802015

1 Introduction

Main-memory systems have sparked interest in high-performance indexing techniques. Indexes are a crucial building block for in-memory databases as they eliminate disk I/O bottlenecks and shift the performance focus to CPU-level optimizations. For in-memory indexes, maximizing CPU cache efficiency and exploiting multi-core parallelism are critical to achieving low-latency data access and high throughput, making these factors crucial in modern index design.

The introduction of Intel 3DXPoint DCPMM non-volatile memory (NVM) has led to a flurry of interest in developing efficient, highly concurrent, *persistent* indexes as they achieve the best of both worlds: persistence with no I/O bottlenecks [1, 4, 16, 17, 22, 24, 27, 29, 30, 33, 35, 37, 40, 45]. However, existing persistent indexes typically support only very simple operations: point query, key/value insertion, and key deletion. This is generally because richer features are very hard to develop. For example, range queries are a highly desirable feature, but the existing persistent indexes that offer range queries guarantee only the weak read-committed correctness criterion [17, 22, 27, 29, 30,

Authors' Contact Information: Arash Khalaji, arash.khalaji@uwaterloo.ca, David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada; Trevor Brown, trevor.brown@uwaterloo.ca, David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada; Khuzaima Daudjee, khuzaima.daudjee@uwaterloo.ca, David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada.



ACM 2836-6573/2026/6-ART138
https://doi.org/10.1145/3802015

33, 37, 40]. Weak correctness conditions, such as snapshot isolation and read-committed, can be difficult to reason about, and it is not clear how one could use range queries satisfying these weaker correctness conditions to implement, say, serializable database joins in the presence of concurrent updates.

Linearizability is the gold standard for correctness in concurrent data structures. However, for range queries, linearizability is substantially more difficult and computationally expensive to guarantee. To provide linearizability for range queries, there are three common options in the literature on volatile (DRAM) indexes: (1) potentially conflicting updates are blocked for the duration of the range query, (2) concurrent updates are logged and then used to patch inconsistencies in a non-atomic range query [15, 42], or (3) multiversioning is used, which typically entails creating copies of data structure nodes on each modification. To our knowledge, *no persistent index implements any* of these techniques.

We consider the pros and cons of each approach. Whereas (1) is simple and effective for very small range queries, it is impractical for large ranges, both due to the cost of acquiring and releasing fine-grained locks over the entire range, and due to the loss of concurrency as a large number of updates may be blocked. Known techniques for (2) offer strong progress guarantees, but are either very complex, encoding the log using sophisticated auxiliary data structures [15, 42], such as lock-free skip-lists, or require specific garbage collection algorithms to be used [3]. As for (3), in traditional multiversion concurrency control (MVCC) [9], the overheads involved in copying nodes on each modification, garbage collecting old node versions, and creating an extra level of indirection (version lists for each versioned address) that must be navigated to access data in the correct version, can be quite substantial.

Recent advancements, however, have suddenly made (3) a compelling option. Blelloch and Wei introduced the VERLIB library [11], which proposed a crucial optimization called *indirection on demand*. This optimization can frequently eliminate, on an address by address basis, the extra level of indirection that is typically required for MVCC. It also integrates highly efficient automatic garbage collection for old versions. The end result of VERLIB's optimizations is a library that can be used to produce MVCC indexes that are extremely close in performance to their unversioned counterparts.

We leverage these advancements to implement two different variants of persistent multiversion adaptive radix tries (`PermART` and `PermART (2-Log)`) with linearizable range queries. Our indexes are inspired by a multiversion, but *not persistent*, radix index that was introduced by the authors of VERLIB. Our new indexes not only (1) substantially outperform the state-of-the-art in a wide variety of workloads, but (2) offer durable linearizability [25] on systems with Optane 3DXPoint memory, meaning their contents are not lost in the event of a power failure, so it is not necessary to rebuild them after recovering from a crash, and (3) introduce a powerful new optimization (in `PermART (2-Log)`) that *further* **reduces the cost of VERLIB's Copy-on-Write (CoW) by up to 70%**. Our new indexes have highly efficient point operations and, crucially, they remain efficient even when there are many in-flight range queries.

The key idea behind our new optimization for reducing CoW overheads is to integrate a small (constant sized) log into each leaf node of our index. By performing the writes for a constant number of versions to this log, we are able to delay CoW for a constant number of updates without losing any version information. The cache locality and compactness of these logs makes them efficient to maintain, and by reserving only two log entries in each leaf node, we reduce the number of CoW operations by two thirds. To the best of our knowledge, we are the first to reduce CoW overheads with in-node logging, an idea that we believe can be applied to paging, buffer management and other data subsystems where CoW is utilized as a standard technique.

We evaluate our new indexes, PermART and PermART (2-Log), against the state-of-the-art in persistent indexes. Our results show that PermART outperforms competitors in point operations (lookups and updates) in dense and sparse key universes by up to 6× and 7×, respectively, and achieves a geometric mean of 35% speedup across all workloads and thread counts. Our copy-on-write optimization using in-node logging improves the throughput of point operations by up to 70% in workloads that have high skew, and thus contention.

Contributions.

(1) We propose PermART and PermART (2-Log), two novel persistent variants of a multiversion ART index.
(2) To the best of our knowledge, our designs are the first persistent indexes to support range queries under correctness guarantees stronger than read committed.
(3) We introduce a new optimization for avoiding CoW overheads in MVCC, called in-node logging, outperforming the state-of-the-art in MVCC by up to 70%.
(4) We systematically stress test existing persistent indexes and identify issues in a considerable number of them.
(5) For indexes that pass our stress tests, we achieve up to 2× performance improvement by redesigning memory management with a state-of-the-art NUMA-friendly NVM allocator.
(6) PermART and PermART (2-Log) outperform not only the existing state-of-the-art, but also these newly improved competing indexes. In particular, our linearizable range queries are up to three orders of magnitude faster than a straightforward linearizable variant of our fastest competitor.

The rest of the paper is organized as follows. Background on NVM, ART, VERLIB, and durable linearizability appears in Section 2. In Section 3, we explain how we make a durably linearizable versioned ART using NVM, and detail our in-node logging CoW optimization. Our experimental evaluation appears in Section 4, followed by related work in Section 5 and concluding remarks in Section 6.

2 Background

This section provides conceptual background on four key areas related to our proposed PermART index. Section 2.1 provides a brief introduction to adaptive radix trees. Section 2.2 introduces VERLIB, and Section 2.3 explains how VERLIB is utilized to make multiversion adaptive radix trees. Section 2.4 overviews non-volatile memory (NVM). Finally, Section 2.5 summarizes durable linearizability as the standard correctness condition for concurrent persistent data structures on NVM.

2.1 Adaptive Radix Trees (ART)

Radix trees are a search tree variant in which different levels of the tree correspond to different chunks in the bitwise representation of the keys. The keys are stored in the leaves, with the inner nodes storing routing information. The Adaptive Radix Tree (ART) is a space-efficient implementation of a radix tree with a radix of 256 [31, 32]. ART, and radix trees in general, are ordered data structures supporting range queries, making them a suitable alternative for B^+-Trees as indexes in databases such as HyPer [26] and DuckDB [43].

ART makes several optimizations to reduce the space consumption and improve the performance of a typical radix tree. First, ART utilizes path compression and lazy expansion to reduce the height of the tree. In path compression, a sequence of nodes with a single child is compressed into a single node that contains the concatenation of the keys in the compressed nodes. Path compression also reduces the number of pointer chases required to traverse the tree, which improves the performance.

Second, ART utilizes four different node types to store the inner nodes, which are chosen based on the number of children. The four node types are: Node4, Node16, Node48, and Node256, which can store up to 4, 16, 48, and 256 children, respectively. The node types are dynamically switched based on the number of children, which helps reduce the space consumption. There are also two types of leaf nodes in ART: small and large. Small leaves store up to 2 key-value pairs, and large leaves store up to 14 key-value pairs. Small leaves are created when fewer keys share a common prefix, while large leaves are used when many keys cluster at the same prefix. In sparse keysets, keys are distributed widely across the key space, leading to more prefix clustering and thus more large leaves. In dense keysets, consecutive keys can be distinguished deeper in the tree structure, eliminating the need for large leaves.

In terms of concurrency, there are multiple approaches for implementing a concurrent ART. These approaches include using Hardware Transcational Memory (HTM), Lock Coupling, Optimistic Lock Coupling (OLC), and Read-Optimized Write EXclusion (ROWEX) [32]. In this paper, we use the ROWEX variant of ART, which has been shown to be more performant in face of high contention [32], and is more suitable for implementing a multiversion variant of ART, as we explain in Section 2.3. In ROWEX, reads never acquire locks, while writes do and perform Copy-on-Write (CoW) to create a new version of the modified node, which is then atomically installed in the tree using a pointer swing. This approach prevents readers from seeing "dirty" intermediate states of the tree, which is crucial for correctness.

As a result of ART's success with in-memory indexing and its support for range queries, there has been a lot of interest in making it persistent, or using it as part of a persistent index. The existing implementations include but are not limited to `ROART` [37], `WOART` [29], `P-ART` [30], and `PACTree` [27], all of which we discuss in Section 5.

2.2 Multiversion Data Structures (VERLIB)

In the context of linked data structures (e.g., linked lists or trees), multiversioning is typically implemented using version chains. In this approach, each node is replaced with a chain of versions, typically implemented with a linked list, where each version (link in the chain) contains versioning metadata (e.g. a timestamp) and a pointer to the next version in the chain. Recently, the trend has been to develop generic versioned data structure frameworks that can be used to convert any concurrent data structure to a multiversion one with minimal expert programmer involvement [11, 48].

Wei et al. [48] propose a VersionedCAS object that can be utilized in any concurrent data structure that accesses shared memory using load and CAS operations. However, due to inherent limitations of this approach, using it enforces an extra level of indirection to every pointer access, which can be detrimental to the performance of pointer-heavy data structures such as trees. Blelloch and Wei [11] propose a follow-up called VERLIB that avoids the extra level of indirection in most cases, and adds it only when it is necessary. In addition, VERLIB can get rid of the extra level of indirection when it is no longer needed, which is called shortcutting.

VERLIB requires every node type in the data structure to extend a `versioned` class, which augments the node with two fields: a timestamp and a pointer to the previous version of the node. Additionally, the pointers in the data structure are replaced with `versioned_ptr` pointers, which handle the versioning logic by adding special functionality to "normal" pointers. For the rare cases where the extra level of indirection is inevitable, VERLIB provides a `version_link` class that contains the pointer to the previous version and the timestamp of the new version, in addition to a pointer to the current version (which is the indirection-causing pointer).

VERLIB maintains a global timestamp (a logical clock starting at 0) that is used to support taking atomic snapshots of memory. Snapshots make it straightforward to compute range queries, since

taking a snapshot of a data structure allows it to be traversed without worrying about concurrent updates.

A range query can be performed by first invoking VERLIB's take_snapshot() function, which atomically increments the global timestamp from ts to $ts + 1$, and saves ts locally. Then it can begin traversing the data structure. For each node it encounters, it can traverse the node's version chain, which is ordered from newest version to oldest, until it finds a version $\leq ts$, which reflects the state of the data structure node when the snapshot was taken (i.e., when take_snapshot() incremented the global timestamp).

As for how versions are maintained in data structure nodes, a newly created node initially has a special timestamp value TBD that indicates an update is in progress. Once the node is attached to the data structure and made visible to other threads, its version is changed from TBD to the current value of the global timestamp. Since nodes receive timestamps when they become visible, any nodes added immediately after a snapshot has incremented the global clock to $ts + 1$ will have a timestamp $\geq ts + 1$.

Old versions can be garbage collected after they are no longer needed by any active range query. To this end, VERLIB maintains a global done_stamp, which tracks the timestamp of the oldest active range query. If done_stamp is ts, any node with a timestamp $< ts$ can be reclaimed. If there are no range queries running in the workload, the global timestamp is never incremented, and it incurs little overhead.

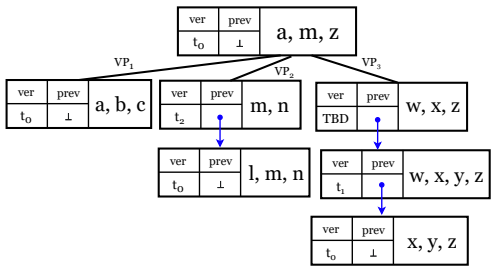


Fig. 1. Example of a trie-like data structure utilizing VERLIB

As an example, Figure 1 depicts a trie-like data structure that uses VERLIB to implement atomic range queries. Suppose updates in this trie use Copy-on-Write (CoW), and the keys are stored in leaves, while internal nodes simply route queries to the appropriate leaf. The pictured trie has only one internal node: the root. There are three versioned pointers, labeled VP_1, VP_2 and VP_3. Suppose three range queries are active, and they each stall right after taking their snapshots, obtaining time stamps t_0, t_1 and t_2, respectively, resulting in a global timestamp value of t_3. Suppose the range queries are each scanning the entire data structure. We trace the steps taken by the range query with timestamp t_1.

(1) Follow VP_1, see $t_1 \geq t_0$ so add $\{a, b, c\}$ to the result.
(2) Follow VP_2, see $t_1 \not\geq t_2$ so follow the prev pointer, see $t_1 \geq t_0$ and add $\{l, m, n\}$ to the result (effectively ignoring the update that erased l).
(3) Follow VP_3, see TBD. Perform a CAS(TBD, t_3), effectively helping[1] the in-flight update to set its timestamp. See $t_1 \not\geq t_3$ so follow prev, see $t_1 \geq t_1$ and add $\{w, x, y, z\}$ to the result.

[1]An alternative would be to block until the node's timestamp has been set, but helping results in faster progress.

2.3 Integrating VERLIB with ART

As explained in Section 2.1, the ART implementation we use utilizes ROWEX for synchronization. In ROWEX, every modification to a node requires creating a new node that reflects it. Once the new node is created, the appropriate pointer is atomically swung to the new node, which makes it visible to the readers. The atomicity of the swing operation is typically achieved by using `std::atomic` pointers. Since the nodes are protected by locks, there is only one writer at a time, which makes it easier to reason about the correctness of the data structure.

Although VERLIB (Section 2.2) is a general-purpose library that can be used to make any data structure multiversion, including lock-free ones, only a small subset of its features is needed to make a versioned ART with ROWEX. In this section, we describe those features that are the most relevant to our work.

There are only two types of writes to the data structure that need to be modified to support multiversioning with VERLIB:

(1) Writes that swing pointers to newly allocated nodes:
   - **Inserts:** adding a new key-value pair to a leaf node entails creating a new version of the leaf node that contains the new key-value pair, placed in the correct order with respect to the other key-value pairs.
   - **Deletes:** removing a key-value pair from a leaf node entails creating a new version of the leaf node that does not contain the key-value pair, while the other key-value pairs remain in the same order.
   - **SMOs:** structural modifications, such as promotions and splits, entail creating new node types that reflect the new structure of the tree.

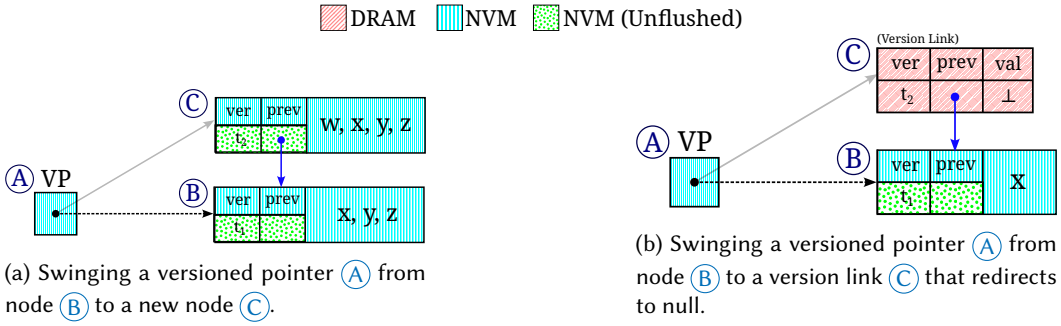
(2) Writes that swing pointers to null:
   - **Deletes:** if a key-value pair is the only one in a leaf node, removing it entails creating a new version link that redirects to null.

We explain these two types of writes below. For simplicity of explanation, we assume that the data structure contains only keys, and the values are omitted as they do not affect the logic.

2.3.1 Swinging Pointers to New Nodes. Figure 2a depicts the process of swinging a versioned pointer (A) from a versioned node (B) to a new one (C) which contains a new key w. The process starts by allocating a new versioned node, the content of which is copied from the old node, and the new key is inserted. The previous pointer of the new node is set to the old node, while the timestamp is still `TBD` (not shown in the figure). At this point, the node is not yet visible to the readers, and the writer can safely swing the pointer to the new node as it is holding a lock on the parent. According to VERLIB specifications, the timestamp on the new node is either set by the writer or by any other thread that sees `TBD` timestamp on the node (helping). The new timestamp t_2 is greater than or equal to t_1.

- $t_2 > t_1$ is true if at least one concurrent range query has incremented the global timestamp.
- $t_2 == t_1$ is true if no concurrent range queries have been initiated.

2.3.2 Swinging Pointers to Null. Figure 2b shows the process of swinging a versioned pointer (A) to null as a result of removing the key x. In this case, merely writing null to the pointer is incorrect, since any range query q that may have started before the deletion must be able to reach the previous version of the node, as x had been present in the data structure when q took its snapshot. Consequently, adding an extra level of indirection is necessary to ensure that the previous version is still accessible to in-flight range queries. To achieve this, the writer allocates a new version link (C), which contains the pointer to the previous version and the timestamp of the new version (which is calculated similarly to the previous case). In VERLIB, the extra level of indirection



(a) Swinging a versioned pointer (A) from node (B) to a new node (C).

(b) Swinging a versioned pointer (A) from node (B) to a version link (C) that redirects to null.

Fig. 2. Two pointer swing types in a VERLIB-based versioned ART

is removed when it is no longer needed, either immediately by the same writer, or by any other thread some time later. This process is called shortcutting, and is explained in great detail in the original VERLIB paper [11].

2.4 Non-Volatile Memory (NVM)

Persistent (or non-volatile) memory (PM or NVM) refers to a class of byte-addressable storage that retains its contents across power failures. Persistent memory sits between traditional volatile memory (DRAM) and block-based storage (SSDs/HDDs) in the memory hierarchy. Intel's Optane DC Persistent Memory (DCPMM) is the dominant commercially available non-volatile memory.

In terms of performance, persistent memory is 2-5x slower than DRAM, and its behaviour with respect to access patterns is not necessarily the same as DRAM. The read and write granularity of Intel's 3D XPoint technology is 256 bytes, so a single cache line (64 bytes) read or write results in a 256-byte read or write from or to the persistent memory. Moreover, the read/write performance asymmetry of NVM is more pronounced than DRAM, with writes being significantly slower than reads.

There are two main ways to utilize Intel Optane persistent memory: memory mode and app-direct mode. In memory mode, the persistent memory acts as a cache for DRAM. Although this mode is transparent to the application and easy to use, it does not provide persistence guarantees as the data in DRAM is lost in case of a power failure. In app-direct mode, on the other hand, persistent memory is directly accessed in a similar fashion to DRAM, and the application is responsible for managing the persistence of the data. In this paper, we focus on the app-direct mode of Intel Optane DCPMM. Our approach is targeted for Optane, and other persistent memory devices may have different characteristics.

To achieve correct persistence, NVM cannot be treated as a drop-in replacement for DRAM. This is because higher levels in the memory hierarchy (e.g., CPU caches and write buffers) are often volatile[2], and a power failure can lead to data loss and/or inconsistency if the data is not properly flushed to persistent memory. Thus, it is the responsibility of the programmer to ensure that the data is correctly and consistently flushed to persistent memory. This is typically done using cache line instructions (e.g., clwb, clflushopt, or clflush) to flush the modified cache lines to persistent memory, followed by a store fence instruction (e.g., sfence or a stronger primitive) to ensure that the flushes are completed before proceeding. Careful consideration must be given to the ordering of

[2]Some Intel processors offer extended Asynchronous DRAM Refresh (eADR), which effectively makes caches persistent. However, proper write ordering remains crucial, as the write ordering counterexamples in Section 3.1 apply equally to systems with eADR.

these instructions to ensure that the data is persisted in a consistent state. We describe in Section 3 how to implement our data structure in NVM.

2.5 Durable Linearizability

Introduced in 1990 by Herlihy and Wing [23], linearizability is the gold standard for the correctness of concurrent data structures. However, linearizability as we know it is not enough for data structures that reside in NVM since it does not take into account the durability of the data structure. To address this, Izraelevitz et al. [25] introduce the concepts of durable linearizability and buffered durable linearizability.

Durable linearizability defines the correctness of a durable data structure under a full system failure model. In this model, the data structure must be able to recover to a consistent state in which any operation completed before the failure is visible after the recovery. Moreover, the results of the read operations that completed before the crash should be completely reproducible after the recovery. Buffered durable linearizability is a weaker version of durable linearizability, where operations are not required to be persisted before completion, and only a consistent prefix of the operations before the failure is required to be visible after recovery. The definition of the consistent prefix is provided in the original paper in sufficient detail [25].

To understand durable linearizability, consider that it consists of two key components: *atomicity* (from traditional linearizability) and constraints on *what can be lost* during a crash. Specifically, durable linearizability guarantees any operations that finish before a crash will not be lost, and requires that operations lost during a crash form a *consistent cut* across the happens-before order—that is, if operation O_2 is preserved (persisted) and O_1 happens-before O_2, then O_1 must also be preserved. The following examples make these requirements concrete.

Example 1: Atomicity Without Durability. Consider a concurrent linked list with a size counter, where insertions are protected by a global lock to ensure atomicity. An insertion adds a new node n with key k and then increments the size counter within the same critical section. If these updates are persisted independently without ordering guarantees, a crash may occur after the node is persisted but before the counter update is durable. After recovery, the list contains node n with key k, while the size counter reflects the old value. This state violates durable linearizability: a post-recovery $\mathsf{find}(k)$ returns the inserted value, implying the insert completed, yet the size counter indicates that it did not happen.

Example 2: Non-Consistent Cut of Lost Operations. Consider two threads operating on a key-value store. Thread T_1 executes $\mathsf{put}(k, v_1)$. Thread T_2 then executes $\mathsf{get}(k)$, which returns v_1, followed by $\mathsf{put}(k, v_2)$. Because T_2 observes v_1, both of its operations happen after T_1's operation. Suppose a crash causes T_1's operation to be lost while T_2's operations persist. After recovery, the store contains $<k, v_2>$ with no record of v_1, even though T_2 read v_1 before the crash. This produces a history in which T_2 observed a value that does not exist in the recovered state. The lost operations do not form a consistent cut of the happens-before relation, which is disallowed under durable linearizability.

3 Design of PermART

In Section 2, we covered the background on ART and VERLIB. We also explained that durable linearizability, while challenging to provide efficiently, is highly desirable for indexes. In Section 2.3, we discussed how VERLIB enables multiversion ART [11]. In this section, we describe how the conceptual pieces of our design fit together to form a durably linearizable multiversion ART that resides in NVM.

In Section 3.1, we present the design of PermART by exploring four requirements and design decisions that are concerned with the conversion of a volatile multiversion ART index to a new,

durably linearizable one. PermART uses the same node types as ART, and the same rules for choosing which node type to use (as discussed in section 2.1). However, as we explain in Section 3.2, when keys are sparsely distributed (a common case), these rules tend to produce many big leaves, which are highly inefficient for CoW. We thus propose an *in-node logging* technique that significantly improves copy-on-write performance and memory consumption in such workloads.

3.1 Ensuring Durable Linearizability

So far, we have discussed the basic steps to make updates to a multiversion ART that resides in DRAM. However, there are significant additional considerations. Importantly, in persistent data structures, the order of writes often matters in a way that it does not in volatile data structures. In a volatile data structure, if a power failure occurs, all data is lost, so the order of a pair of writes performed immediately before the failure is irrelevant. However, in a persistent index, incautious write ordering can completely destroy the index.

First Requirement: Write Ordering. As a toy example, in a volatile linked-list, insertion of a new node b between existing consecutive nodes a and c can be done as follows: (1) lock node a, (2) create the new node node b, (3) determine which node c *currently* follows a, (4) change a to point to b, (5) change b to point to c, and finally (6) release the lock on a. However, if one were to implement a persistent linked-list in the same way, and a power failure occurred after step (4) reaches NVM, but before step (5), the entirety of the list after a is lost. How does ordering help here? If steps (4) and (5) are reversed, then this data loss does not occur.

This problem can easily occur in version lists in a multiversion ART index, as shown in Figure 2a. The writer thread needs to flush every cache line in Ⓒ before flushing the cache line containing Ⓐ so that the possibility of Ⓐ pointing to a corrupt node after recovery is eliminated. In general, *a node's contents must be persisted before a pointer to it can be persisted.*

Second Requirement: Persisting as Little as Possible. Since writing to persistent memory and flushing and fencing to maintain correctness can incur high overhead, a typical design goal in persistent data structures is to persist as little as possible. However, there is a balance between minimizing persistent data and the simplicity of recovery after crashes or power failures that should be considered. In our design, we opted for ease of recovery, which means most of the pointers and nodes are fully persistent. However, there are two notable exceptions:

(1) Inline VERLIB metadata in nodes: Nodes such as Ⓑ and Ⓒ in Figure 2a and Ⓑ in Figure 2b make up most of the nodes in our data structure. However, the VERLIB metadata (timestamps and previous version pointers) are required by only range queries. Consequently, although these fields are stored in NVM alongside other fields, there is no need for them to be correctly persisted, as they will be ignored upon recovery. For example, there is no need to flush the timestamp field (and perform the necessary fences) once the node has been timestamped.
(2) Indirect version links: As explained in Section 2.3, indirect version links appear when setting a pointer to null to ensure in-flight range queries can access previous values. Since this is the only scenario in which these version links appear, we can store them in DRAM to speed up their reading and writing. If a pointer to DRAM is encountered during recovery, we can consider its value as null. Moreover, if VERLIB "shortcuts" this pointer to null, there is no need to correctly persist it as both possible values (pointer to DRAM or null) simply become null on recovery.

Third Requirement: Preventing Non-linearizable Point Operations. Although ordering writes to NVM is crucial for ending up with a correct data structure after recovery, it is not the only requirement to achieve a durably linearizable data structure. We now explain the most sophisticated requirement.

```
■ 1  lock.acquire();
  2  vptr<Node> new_node = copy(old_node); // Copy
  3  new_node->insert(new_kv_pair);
  4  new_node->prev = old_node;
  5  new_node->timestamp = TBD;
  6  new_node->flush();
■ 7  vp = new_node | UNPERSISTENT_BIT; // Swing
  8  vp.flush();
■ 9  vp = new_node & ~UNPERSISTENT_BIT;
■ 10 lock.release();
```

Listing 1. A typical ART Copy-on-Write section

Using Figure 2a as an example, consider a reader thread p doing a point lookup for key w concurrently with the writer thread q. Once q swings the pointer from (B) to (C) but before it flushes the cache line in (A), p reads the pointer in (A) and sees the new node (C), which means the point lookup successfully returns the presence of w. However, if a power failure occurs at this point, and (C) has not yet been flushed to NVM, then after recovery, the node will contain only keys x, y, z. In this case, it is impossible to linearize a point lookup that has seen the key w before the power failure, since the key w is not in the data structure after recovery. Similar violations of linearizability can also occur for insert and delete operations.

Our solution to this problem is inspired by the *link-and-persist* technique of David et al. [18]. We reserve a bit in each pointer to represent an *unpersisted mark*, indicating that the node it points to has not *necessarily* been persisten yet. We then prevent readers from accessing any nodes pointed to by *unpersisted* pointers. In the example above, once p sees the *unpersisted* bit in the pointer in (A), it knows that the pointer is necessarily flushed to NVM, and it should either (1) avoid reading the new node (C), read the old node (B) instead (effectively linearizing the read before the write), or (2) wait until the bit is unset (linearizing after the write).

How we CoW in PermART. Listing 1 shows a code snippet describing how CoW is performed PermART. The code is simplified for clarity, and it does not show the details of memory allocation, copying, flushing, fencing, or inserting the new key-value pair. The line numbers that are preceded by a ■ symbol indicate that the operation in that line is performing a std::atomic store under the hood, which implies a full memory fence with C++'s default sequentially consistent memory order (memory_order_seq_cst). The vp variable corresponds to the versioned pointer (A) in Figure 2a, while old_node and new_node correspond to (B) and (C), respectively. Crucially, although new_node is visible after line 7, no operation can read its contents until the *unpersisted* bit is unset at line 9.

Fourth Requirement: Do We Persist Correctly? We briefly sketch correctness, taxonomizing the scenarios in which a crash can occur, and explaining how we ensure durable linearizability in each.

- **Crash before line 7:** Since the pointer is not swung yet, the new node is not reachable, and the data structure remains unchanged, and hence it remains correct (induction).
- **Crash between lines 7 and 9:** In this case, the pointer is swung to the new node, but it might not be flushed to NVM.
  - If the pointer is not yet flushed to NVM, the data structure remains unchanged, and hence remains correct.

  - If the pointer is flushed to NVM, the new node is reachable. If the *unpersistent* bit is set, the recovery procedure *completes* the operation by unsetting the bit, and the first operation to see this change must be reading after the recovery, and hence it is correct to see the new node.
- **Crash after line 9:** In this case, we do not flush the pointer after unsetting the *unpersistent* bit, so the *cleaned* pointer may or may not be flushed to NVM. In each case, recovery *completes* this operation O by unsetting the *unpersistent* bit. If the bit was unset before the crash, but not persisted, then some operations may have seen the new node before the crash, and therefore need to be linearized after O. This is fine, because outside of recovery, no thread can determine whether O was was completed before the crash or after recovery. Consequently, we linearize O before the crash and, in turn, before any operations that observe its effects before the crash.

3.2 Optimizing Copy-on-Write

Implementing and evaluating an initial design of PermART revealed a challenge in extreme high-contention, update-heavy workloads: Our fastest competitor, PACTree, achieved higher point-operation throughput at high thread counts. This is shown concretely for a 95%-update Zipfian workload in Figure 6 (Section 4.1).

The Diagnosis. Even though the VERLIB CoW mechanism is highly optimized, it still has a significant drawback: It requires allocating a new node for every modification. This can be quite expensive in terms of both time and space. This cost is especially high in NVM, where writes are more expensive than reads, and memory allocation can be slow.

To pinpoint the most impactful source of remaining CoW overhead in PermART, we heavily instrumented our code to record the frequency of CoW modifications to the various types of nodes defined for our radix trie structure. We found that performance was poor when there are many updates to *large leaves*, each of which contains up to 14 keys and values (256B), and thus can dramatically increase CoW overhead.

This occurs, for example, in workloads where keys are drawn from a large, sparse universe (where keys can be spread out quite far, such as 300M keys drawn uniformly from $[1, 2^{64})$). As it turns out, **sparsity** in the index (which is common in real world data, e.g., [2]) **produces large leaves**. To see why, let's understand why **density produces small leaves**. Suppose two adjacent keys are inserted. A radix trie insertion will typically continue descending the trie as long two keys fall into the current bucket, terminating when each key is placed in its own leaf. A very dense set of keys, such as [1, 100] will then end up creating many small leaves because adjacent keys only differ in their very last byte, which causes them to fall into separate, small, leaves. As a result, our logging optimization for large leaves would be ineffective for such distributions. On the other hand, *sparsity* causes keys whose difference does not lie in their very last byte to fall into the same large leaf, where our optimization would be highly effective. Table 1 quantifies the number of internal and leaf nodes of each type in an ART data structure filled with 100M keys in a 300M key universe.

A subtlety worth mentioning is that inserting keys uniformly from a *dense universe* produces a *sparse trie* until sufficiently many keys have been inserted. A surprising consequence of this is that logging can improve performance significantly while *constructing* a trie, even if the trie is being filled with most of a dense key universe.

The Prescription. Our in-node logging optimization specifically augments these *large leaf nodes* with *two log entries*[3]. The layout of a log entry is shown in Listing 2, and a sketch of our logging

[3]We also experimented with 1, 3 and other numbers of log entries and found that 2 balanced simplicity and performance.

Table 1. Node type quantity for dense and sparse keysets

Node Type	Dense	Sparse
Full (2112 B)	1.2M	65K
Indirect (640 B)	2.8K	1.8K
Sparse (192 B)	0	20K
Small Leaf (64 B)	100M	1.35M
Large Leaf (256 B)	0	15.7M

```
struct LogEntry {
   std::atomic<uint8_t> meta;
   TS timestamp; K key; V value;
};
```

Listing 2. The Structure of a Log Entry

```
lock.acquire();
if (log.writeable()) {
   log.key = new_key;
   log.value = new_value;
   log.timestamp = TBD;
   log.meta = VALID | IS_INSERT | UNPERSISTENT_BIT;
   log.flush();
   log.meta &= ~UNPERSISTENT_BIT;
}
else { /* Reconcile and CoW */ }
lock.release();
```

Listing 3. A Typical Logging Code Path

code appears in Listing 3. The small size of our log entries helps them to fit in one cache line, and reduces the number of flushes needed to persist new writes.

With in-node logging, instead of performing CoW on a large leaf node for every modification, we first fill up its log entries. If there are l log entries, the number of CoW operations is reduced by a factor of $l + 1$ (e.g., 1 log entry reduces CoW count by half). The log entries are written only once, and are never modified. In addition to shortening the critical section, logging reduces the absolute number of cache line flushes by roughly 40% across all workloads.

Once all of the log entries are used up, we fall back to VERLIB's CoW mechanism, while also ensuring that the log entries are reconciled with the key-value pairs in the leaf node. For example, if a log entry indicates an insertion of a key-value pair, but the key-value pair is not in the leaf node, we need to insert it during CoW.

Logs that Last. Log entries blur the lines between nodes and version lists, but since they appear in a node, and their information is not duplicated elsewhere, they must be persisted. One can think of log entries as a special type of CoW — analogous to our usual CoW in the following ways:

(1) Whereas our CoW requires a new node, using a log entry does not. (2) Rather than initializing a node and then swinging a pointer to it, we populate a log entry, then set the valid bit to indicate that it is well-formed. (3) As in CoW, where we unset the unpersistent bit on a pointer to allow other operations to follow it, we unset the log entry's unpersistent bit to allow other operations to read it, including the recovery procedure. At recovery, valid bits, which are flushed only after the corresponding log entry is populated, are used to determine which log entries were persisted correctly, and which should simply be ignored and reset.

4 Experimental Evaluation

In this section, we present our performance evaluation of PermART against state-of-the-art competitors for point and range queries.

Experimental Setup. We use the publicly available benchmark Setbench for concurrent data structures [13]. We run our experiments on a 2-socket machine with Intel Xeon Gold 5220R processors (24 cores per socket, 2.2 GHz) and 192 GB of main memory. The machine is equipped with 1.5 TB of Intel Optane DC Persistent Memory (DCPMM) in app-direct mode, installed so that each socket has local access to 6 DCPMM modules of 128 GB capacity each. The machine runs Ubuntu 20.04.4 LTS with Linux kernel version 5.15.0-124-generic. We use the GNU C++ compiler version 11.4.0 with optimization level -O3. In all experiments, we pin the threads to the second socket first to delay NUMA effects as much as possible. Consequently, experiments with 48 threads only run on the second socket (with hyper-threading). We report the average of 5 runs for each experiment, and the error bars show the minimum and maximum observed value.

Competing Data Structures. We compare PermART with state-of-the-art persistent data structures. We integrate the following data structures into Setbench and use them as competing data structures in our experiments:

- **P-Masstree [30]:** A conversion of the DRAM index Masstree [38] to a persistent data structure using the techniques proposed in RECIPE [30]. Masstree's structure consists of a trie-like concatenation of fixed-length B$^+$-Tree structures. P-Masstree enhances the original data structure to guarantee correct read-committed persistence.
- **ROART [37]:** A range-optimized persistent implementation of ART [31, 32].
- **PACTree [27]:** A two-layer persistent trie that uses a B$^+$-Tree-like leaf node structure that sits below an ART-like search layer.

These three data structures all support atomic point queries and updates, as well as **non-atomic range queries**. To ensure fairness in our experimental results, we use Ralloc [14] to allocate persistent memory for all data structures.

Implementations of data structures considered are provided[4]. Several competitors were excluded for various reasons: wB$^+$-Tree [15] and WORT [29] lack concurrent operation support [33, 37]; uTree [17] has documented correctness issues [22]; DPTree [51] and P-HOT [10, 30] do not support delete operations required by our benchmark; FPTree [40], LB$^+$-Tree [35], NV-Tree [50], and RNTree [36] rely on Intel HTM for synchronization, which is frequently disabled for security reasons; and P-BwTree [30, 34, 47] is omitted in favour of faster alternatives that subsume it [30, 45]. Additionally, BzTree [4] lacks an official implementation and its open-source version [1] crashes under high-skew workloads due to an unresolved concurrency issue, while FastFair has known bugs in its public implementation [30] and exhibits frequent non-terminating executions; both are included in the artifact.

[4]https://github.com/arashctl/sigmod26-PermART

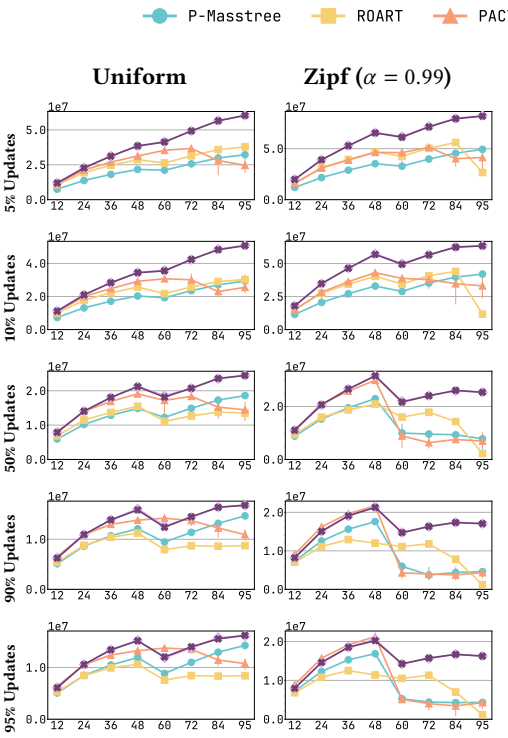


Fig. 3. Point operation performance in dense $[1, 100M]$ universe prefilled to 30M. X: # threads. Y: ops/sec.

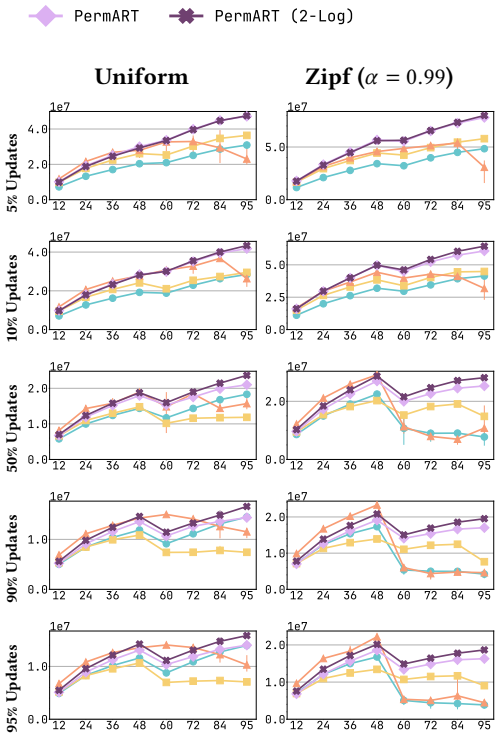


Fig. 4. Point operation performance in sparse $|100M|$ universe of 64-bit keys prefilled to 30M. X: # threads. Y: ops/sec.

4.1 Point Operation Performance

In this section, we compare the point operation (insert, delete, and lookup) throughput of our data structure with that of competing data structures. In Figures 3 and 4, we prefill the data structures with 30 million keys drawn from a universe of 100 million keys, while in Figures 5 and 6 we prefill with 100 million keys drawn from a 300 million key universe. The universe is either dense (keys from 1 to universe size) or sparse (64-bit random keys). Figures 3 and 5 depict the results for dense keysets, while Figures 4 and 6 demonstrate the results for sparse keysets in runs that take 10 seconds each. In a sense, sparse keysets resemble workloads with variable key length, but with a 64-bit cap on the length of keys.[5] The update percentages vary from 5% to 95%, and the key distributions are either uniform or Zipfian with an α value of 0.99, with 0 meaning no skew and 1 meaning extreme skew. Zipfian generated keys are scrambled to resemble real-world skewed workloads more closely.

Up to 100M Dense Keys. Figure 3 depicts the throughput for the dense keyspace experiments with a universe of 100 million keys prefilled with 30 million keys. There are a few observations made from these results:

[5]Note, however, that the lengths are not necessarily uniformly distributed. For example, for uniformly random keys, half of all keys are 64-bits long, half of the remaining keys (so 1/4 of all keys) are 63-bits long, 1/8th are 62-bits long, and so on. 99% of keys are 57-bits or longer. For Zipfian (skewed) distributed keys, arbitrary key lengths may be more or less popular, depending on which keys are hot.

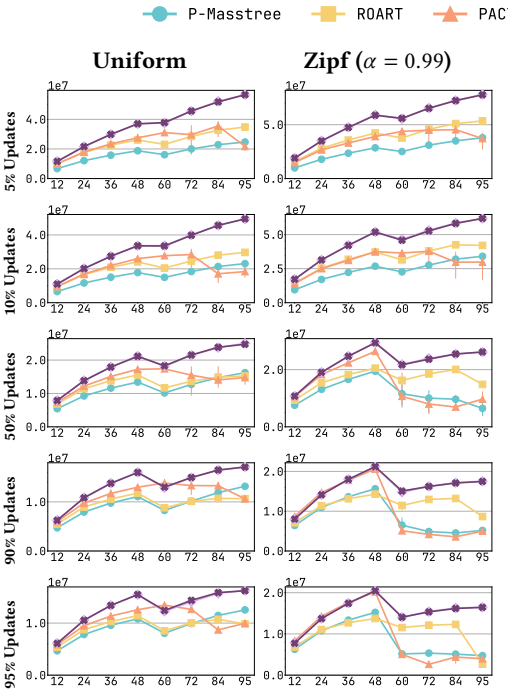


Fig. 5. Point operation performance in dense $[1, 300M]$ universe prefilled to 100M. X: # threads. Y: ops/sec.

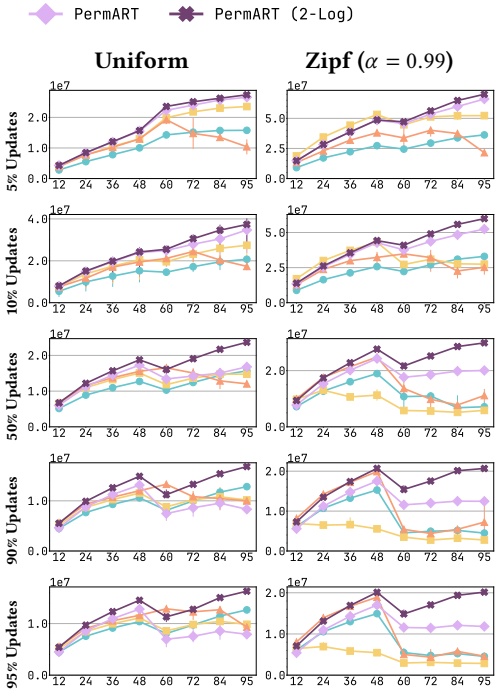


Fig. 6. Point operation performance in sparse $|300M|$ universe of 64-bit keys prefilled to 100M. X: # threads. Y: ops/sec.

- Throughout all workloads and distributions, our in-node logging approach (`PermART (2-Log)`) consistently matches the plain approach without logging (`PermART`), since the closeness of the keys in dense keyspaces prevents large leaves from forming, which leaves the logging code path inactive.
- Before NUMA effects kick in (up to 48 threads), `PACTree` exhibits good performance and scalability, closely matching `PermART` in write-heavy skewed workloads (bottom right plots).
- However, NUMA effects cause a drastic drop in throughput for `PACTree` in such workloads. This is because `PACTree` uses optimistic synchronization, which requires an operation to perform a consistent sequence of reads (uninterrupted by concurrent updates) in order to succeed. In highly skewed workloads, hot keys are highly contended, and it becomes highly likely that many operations will be interrupted by concurrent updates to those keys. This leads to a large drop in both read and write performance, as many operations must restart multiple times before they can obtain a consistent sequence of reads.

Performance drop in PACTree. In experiments that include *all* allocator options (data omitted to avoid cluttering the graphs), we observed that the drop in `PACTree` in update-heavy skewed workloads is less severe with its default memory allocator. This allocator consists of an object pooling mechanism built on top of PMDK, and it owes its improved performance entirely to the pooling mechanism that is an orthogonal optimization that could be performed to all of the data structures we study. Despite `PACTree`'s better performance with pooling in update-heavy skewed workloads, it is still significantly slower than `PermART` in the presence of NUMA effects (after the

drop). On the other hand, with PACTree's original allocator, it is slower than PermART in most other workloads. Therefore, we decided it was more charitable to show PACTree with Ralloc.

Up to 100M Sparse Keys. Figure 4 depicts the throughput for the sparse keyspace experiments with a universe of 100 million keys prefilled with 30 million keys. In these experiments, the keys are 64-bit random numbers, which means that the keys are not necessarily close to each other. There are a few takeaways from these results:

- Since sparse keyspaces lead to larger leaves, the in-node logging approach (PermART (2-Log)) is executed more often, leading to a throughput improvement in write-heavy workloads compared to the plain approach without logging (PermART). For example, in the uniform distribution with 50% updates performed by 95 threads, PermART (2-Log) achieves a throughput of 23.6 million operations per second, while PermART achieves a throughput of 20.9 million operations per second, which is a 12.6% improvement. This improvement increases to 14.5% in the same distribution with 90% and 95% updates.
- Similar to the dense keyspace experiments, PACTree exhibits good performance and scalability before NUMA effects kick in (up to 48 threads), and it even slightly outperforms our data structure in write-heavy workloads with high skew. For example, at 48 threads with 90% updates, PACTree achieves a throughput of 23.3 million operations per second, which is 11.8% higher than the throughput achieved by PermART (2-Log), 20.8 million operations per second.
- However, a PACTree performance drops sharply in write-heavy high-skew workloads when NUMA effects kick in (>48 threads).
- Although it is not specifically designed for NVM usage, P-Masstree exhibits good performance in write-heavy uniform workloads, and it even matches PermART in the 95% update workload with 95 threads. However, with our in-node logging approach, PermART (2-Log) outperforms P-Masstree by $\approx$ 14%.

Up to 300M Dense Keys. Figure 5 shows a workload similar to Figure 3, but with a universe that is three times larger.

- As explained earlier, due to the key distribution of this dense keyset, in-node logging does not offer any performance gains as large leaf nodes that enable logging are seldom created.
- The general trends and even the absolute performance numbers of all data structures are very similar to Figure 3.

Up to 300M Sparse Keys. Figure 6 shows a workload similar to Figure 4, but with a universe that is three times larger.

- In high-skew workloads, even read-heavy ones, ROART's performance exhibits a significant drop compared to the corresponding (smaller universe size) workloads in Figure 4.
- As expected, the performance gain of in-node logging is visible in this set of experiments. Unlike with Figure 4 where the performance improvement achieved by in-node logging was at most 14%, in this workload, **in-node logging achieves up to 70% improvement** over the plain implementation of multiversion ART, being the deciding factor in its superior performance compared to other data structures.

4.2 Range Query Performance

We compare the range query performance of our data structure with the competing data structures while observing the performance of concurrent updates. First, we describe the methodology we use to measure the range query performance, and then present the results. There are two main factors that we vary in our experiments:

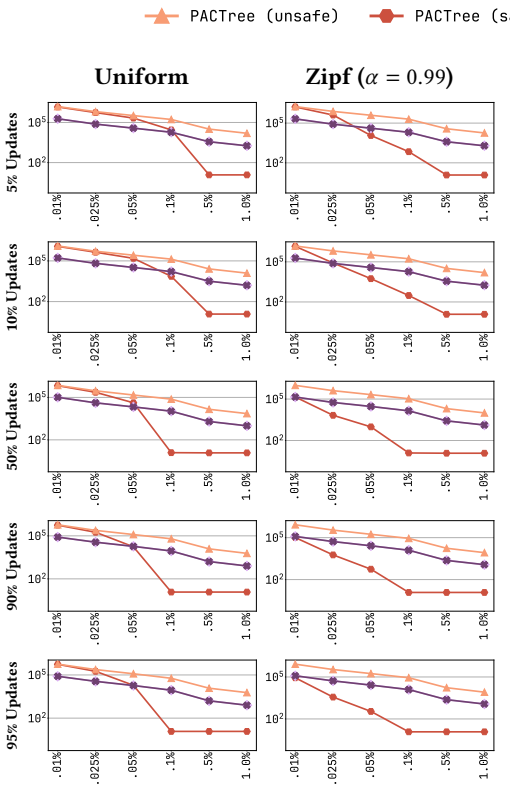


Fig. 7. Range query performance (dense data). X: range selectivity. *Y (log)*: total # of range queries.

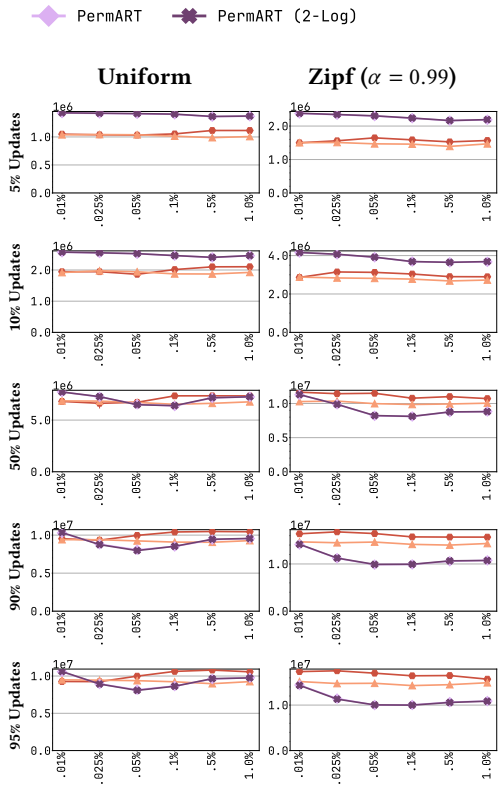


Fig. 8. Update performance in presence of range queries (dense data). X: range selectivity. Y: ops/sec.

(1) **Selectivity:** With respect to the universe size, the cardinality of the range being queried affects the performance of both range queries and other concurrent operations. We vary the selectivity from 0.01% to 1% of the universe size. Since we use a universe of 100 million keys, this means the range queries look for ranges as small as 10,000 keys and as large as 1 million keys.
(2) **Contention:** Concurrent updates can affect the performance of range queries, and the magnitude of this effect depends on the synchronization mechanism employed by the data structure. We fix the total number of threads to 48 to avoid NUMA effects, and use a fixed number of threads, 12, to perform range queries, while the remaining 36 threads perform point operations. Then, we vary the update percentage of the point operations from 5% to 95%, similar to experiments in Section 4.1.

Contention and selectivity also affect each other: higher selectivity means that range queries touch more keys, which increases the chance of contention with concurrent updates.

Competitors' Range Query Support. The competing data structures in Section 4.1 support range queries, albeit with different APIs and different synchronization mechanisms. However, the synchronization granularity for all of them is at the level of an individual leaf, meaning, if a range query needs to read multiple leaves, it synchronizes on each leaf separately. Consequently, these range queries are not linearizable (and in fact, neither atomic nor snapshot isolated) in these data

structures: Concurrent updates can affect the result of a range query after it has read a leaf and before it reads the next leaf. We call such range queries **unsafe**.

For the sake of comparison, we evaluate PACTree's **unsafe** range query implementation in this section, and we also implement a naive optimistic, linearizable range query for it by restarting the range query if any concurrent update is detected in any of the leaves being read. We call PACTree with unsafe range queries **PACTree (unsafe)** and **PACTree (safe)** is defined analogously.

4.2.1 Range Queries Completed in 30 sec. We increase the experiment duration to 30 sec since range queries are more time-consuming than point operations, and premature termination of long range queries (when an experiment ends) can lead to misleading results. Figure 7 shows the total number of range queries completed during the experiment. Some takeaways from these results:

- An overall reduction in the absolute number of completed range queries is expected (since larger queries take longer) and universally observed as the selectivity of the range queries increases.
- PACTree (unsafe) is consistently the fastest in all workloads, outperforming linearizable range queries significantly. This is expected, since it makes no attempt to coordinate with concurrent updates in the range, and returns inconsistent results.
- For very small range queries, PACTree (safe) performs similarly to PACTree (unsafe). However, as range queries become larger, and as updates become more frequent, performance drops dramatically. However, even for very low update rates, PACTree (safe) virtually grinds to a halt as the range queries become large.
- Concretely, with 5% uniform updates and 0.5% selectivity, **only 12 range queries** are completed in 30 seconds[6], whereas with 0.1% selectivity, around 27,000 range queries are completed in the same time. PermART, on the other hand, completes around 3,600 and 18,000 range queries in the same workloads, respectively—somewhat slower for small queries, but **PermART is orders of magnitude faster for large queries**.

4.2.2 Update Throughput in the Presence of Range Queries. Figure 8 shows the throughput of update operations performed by *the other* 36 threads in the exact same experimental runs as Figure 7.

In read-mostly workloads, PermART and PermART (2-Log) significantly outperform PACTree. However, as the selectivity of range queries (x-axis) increases, there is a drop in the performance of PermART and PermART (2-Log), and this drop is particularly significant when those 36 threads are executing write-heavy workloads. We can see this *performance drop is at most 31% in the worst scenario for our algorithm* (95% updates, Zipfian, 0.05% selectivity). On the other hand, **our algorithms are up to 60% faster** for, e.g., a 5% update, Zipfian workload with 0.01% selectivity.

The performance drops shown are not entirely unexpected, as PermART and PermART (2-Log) incur overhead to create and garbage collect multiple versions of leaf nodes, and similar effects were also observed in the corresponding DRAM data structure in VERLIB.

However, we were initially quite surprised to observe that performance often **initially decreases**, and then **eventually increases** with selectivity. This subtle effect was quite challenging to explain, as is often the case for results with two opposing factors.

4.2.3 Why does performance initially decrease? (Factor 1). Our observations show that the root of the problem lies in VERLIB's epoch-based reclamation (EBR) and the internals of Ralloc. Updates generate garbage via CoW, and this garbage is reclaimed only if EBR decides there are no range queries that might read the old node. As selectivity increases from 0.01% to 0.05%, range queries take longer to complete, reducing the frequency of epoch advancements and causing garbage to

[6]Since we have 12 range query threads, this means that each thread completes only one range query in 30 seconds, and 12 other range queries are terminated mid-flight.

accumulate in per-thread retirement lists. For example, in the workload shown in the bottom-right corner of Figures 7 and 8, the number of epoch incrementations reduce from 11400 to 2500 when selectivity is increased from 0.01% to 0.05%. Our measurements show that the average number of cycles spent to free a node peaks at 0.05% selectivity with 3500 cycles, while at 0.01% it takes 590 cycles, and at 0.5% it takes 1700 cycles. This is *not* the case with DRAM allocators (malloc) and non-persistent NVM allocators (libvmmalloc), where a peak is not observed and the number of cycles to free an average node monotonically increases with selectivity.

4.2.4 Why does performance eventually increase? (Factor 2). As selectivity increases and more keys must be located, the time window during which a range query can be interrupted grows. In **PermART and PermART (2-Log)**, this leads to longer version lists since more updates occur during each range query (and therefore within each global version timestamp). As version lists become longer, range queries spend an increasing proportion of their time accessing tails of long version lists (confirmed experimentally), and proportionately *less time accessing the heads* of version lists. The head is a version list that is special because unlike range queries, *updates access only the head*. Thus, range queries with long version lists spend less time accessing the same memory as updates, and thus create less memory contention with updates while slowing them less. Range queries spend more of their time accessing data that is irrelevant to updates, and updates spend more of their time uncontended, thereby allowing faster updates. This explains why our new algorithms' point operations regain some of their lost performance at high selectivities, but this explanation does not apply to PACTree.

The write performance of **PACTree (unsafe)**, however, is relatively unaffected by the selectivity of concurrent range queries. This is because its **unsafe** range queries make absolutely no attempt to synchronize with updates, so there is no reason for updates to be slowed significantly by range queries. This explains why we do not see such a substantial positive effect for PACTree (unsafe).

Interestingly, **PACTree (safe)** is a very different story. There's clearly a powerful positive effect as selectivity increases, but the mechanism is slightly different than in PermART. Range queries are optimistic, and as the range query traverses new keys, it continually revalidates already-read values to ensure they haven't changed. As a result, under heavy contention, range queries are almost always aborted very early on in their traversals, which causes them to restart. Consequently, many range query attempts end up *not touching* the vast majority of keys they are trying to access. This translates into greatly lowered contention experienced by update threads, which dramatically improves their performance.

On the whole, our algorithms appear to compare quite favourably to the state of the art PACTree, being able to *serve orders of magnitude* more **atomic** range queries per second, while remaining competitive for updates.

4.3 MVCC Cost Breakdown

We perform an empirical MVCC cost profile of the performance trends of earlier experiments. We first analyze the cycle-level overheads of insert operations to understand the impact of Copy-on-Write (CoW), timestamping and lock contention under high concurrency, and how reducing CoWs with PermART (2-Log) changes this cost profile. We then study the cost of incrementing the global version used for range-query snapshotting, isolating its effect on both range queries and updates in mixed workloads.

4.3.1 CoW and Timestamping Cost. Using the rdtscp instruction for read timestamp counter, we instrumented our code to measure the cycles spent in different parts of the insert operation. We chose a workload corresponding to the bottom-right corner of Figure 6, particularly the data points

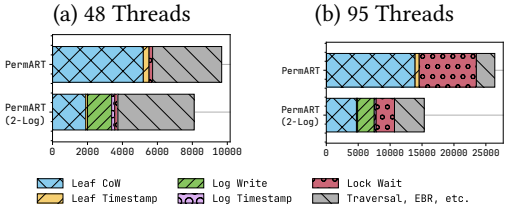


Fig. 9. Cycle breakdown of an average randomly sampled successful insert

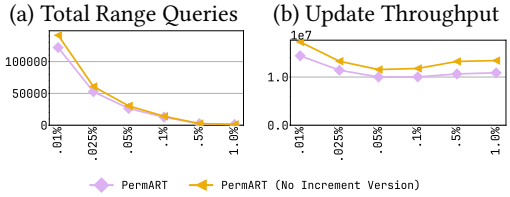


Fig. 10. Effect of disabling global version increments. X: Selectivity.

at scale having 48 and 95 threads, to construct Figure 9, which shows the cycle breakdown of an average randomly sampled successful insert.

Figure 9 shows that leaf CoW (including its associated overheads such as allocation and attachment) accounts for a considerable fraction of cycles in an average insert. By reducing the number of CoWs by one-third with `PermART (2-Log)`, we observe that the log write operations that replace CoWs consume fewer cycles overall. Moreover, `PermART` struggles at 95 threads due to the long critical section caused by CoW, which manifests as a large number of cycles spent waiting for locks to be released—a bottleneck that in-node logging significantly alleviates.

4.3.2 Performance Impact of Incrementing the Global Version. As explained in Section 2.2, range queries increment the global version upon invoking take_snapshot(). In workloads where range queries are absent, such increments never occur. However, in mixed workloads that include range queries, the performance impact becomes observable.

To isolate this cost, we disabled the increment of the global version at the expense of correctness for range queries. Doing so yields a consistent improvement of around 28% in the total number of completed range queries (Figure 10a), under a Zipfian workload with 95% updates (identical workload to the bottom-right corner of Figures 7 and 8). The improvement occurs because range queries are satisfied by the first node they visit in every version chain, rather than traversing further to find the appropriate snapshot. Figure 10b shows that keeping the global version unchanged also improves update throughput, as writers no longer incur cache misses on the cache line containing the global version (since it is not incremented).

4.4 In-Node Logging Benefits

This section examines the impact of in-node logging on updates and range queries. We first show that logging reduces insert latency under high update rates. We then analyze range queries in sparse key sets, demonstrating that logging shortens version chains and reduces version traversal cost, with benefits that grow as range-query selectivity increases.

4.4.1 Impact of In-Node Logging on Write Latency. To gain further insight into how logging affects write performance, we present latency percentiles of successful insert operations in Figure 11. The experimental setup matches the bottom-right corner of Figure 6: a 300M sparse universe with 100M prefilled keys under a Zipfian workload with 95% updates. Figures 11a and 11b show the results for 48 and 95 threads, respectively. In-node logging improves latency across all percentiles in both configurations, with the improvement being more significant with 95 threads.

4.4.2 Impact of In-Node Logging on Range Queries. Figures 7 and 8 show the performance of range queries in dense key sets. However, as discussed earlier, logging does not improve performance in dense key sets because large leaves, which benefit most from logging, are absent. Since logging shortens version chains, it allows range queries to reach the desired version of each node in fewer

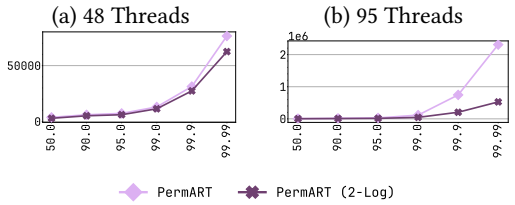


Fig. 11. Latency percentiles of successful insert operations. X: percentile. Y: Successful insert latency.

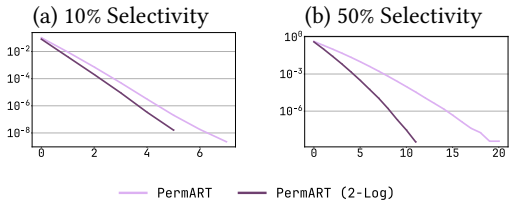


Fig. 12. CCDF of version chain traversal length per visited node. X: Versions traversed. *Y (log)*: Fraction of traversals with length > X.

hops. To demonstrate this, we repeated the experiments of Figures 7 and 8 with sparse key sets and recorded, for every node encountered by a range query, the number of pointer chases required to reach the desired version.

Figure 12 depicts the complementary CDF, i.e. (1−CDF), of the number of version chain traversals per node visited, for selectivities of 10% (Figure 12a) and 50% (Figure 12b). As selectivity increases, the gap between the two curves widens and the maximum number of versions traversed grows. As expected, range queries in `PermART (2-Log)` traverse fewer versions because in-node logging shortens version chains by consolidating multiple updates within each node.

4.5 Durability Cost Breakdown

We next examine read behavior, isolating the cost of durable linearizability by measuring the impact of the unpersisted mark on read latency under contention.

4.5.1 Cost of Durable Linearizability of Reads. As described in the third requirement of Section 3.1, readers must spin on an *unpersisted mark* to avoid observing data that has not yet been persisted. To study the performance impact of this mechanism, we measured the latency of find(key) operations under a heavily update-dominant (99%) Zipfian workload. We deliberately used a small, dense universe of 10 million keys to maximize contention, as the window during which the unpersisted mark is set is very brief. Figure 13 shows that the cost of durable linearizability manifests as increased tail latency for read operations as contention increases.

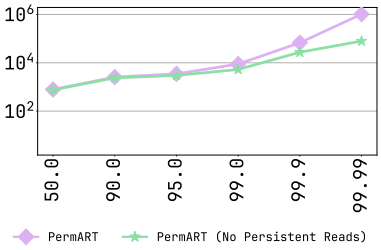


Fig. 13. Latency percentiles of find(key) operations. X: percentile. *Y (log)*: Latency.

5 Related Work

We covered ART and VERLIB in Section 2. We discuss the remaining related work on multiversioning and persistent indexes next.

5.1 Multiversioning

Maintaining multiple versions of data structures using version chains is not a new idea [44]. MVCC is used widely in database systems [6–9, 21, 39, 41], where it is generally implemented with paging; however, paging is less of an issue in in-memory data structures and database systems. Consequently, the literature on multiversioning in data structures takes slightly different approaches [3, 5, 11, 12, 15, 42, 48]. These approaches typically support high-performance range queries without hindering the performance of writes. Brown and Avni [12] implemented linearizable range queries

for k-ary search trees. Skip lists supporting range queries were introduced by Avni et al. [5]. Some prior work implement concurrent binary search trees that support range queries [20, 49].

General methods have been proposed that apply to certain *classes* of concurrent data structures to augment them with range query support. Petrank and Timnat [42] propose a general method for lock-free data structures to support linearizable, non-blocking, scan (full iterator) support. This approach was later improved to support arbitrarily sized range queries by Chatterjee [15]. Arbel-Raviv and Brown [3] propose using epoch-based memory reclamation to support range queries, which is also taken by VERLIB [11].

Even more general approaches augment *any* concurrent data structure with snapshotting and linearizable range query support [11, 48]. Since PermART uses VERLIB to achieve multiversioning, we present VERLIB earlier in Section 2.2.

5.2 Persistent Indexes

5.2.1 Hybrid NVM-DRAM Indexes. These indexes typically store the search layer in main memory, which means the recovery procedure needs to reconstruct the index.

Yang et al. [50] introduced NV-Tree that introduces *selective consistency*, which means leaf nodes are strictly consistent, but internal nodes' consistency is relaxed. Zhou et al. [51] proposed DPTree that batches updates to amortize the cost of persistence. DPTree combines up to two B$^+$-Tree instances, in addition to a hybrid trie. Oukid et al. [40] introduce FPTree that leverages fingerprinting to reduce the in-leaf probing overhead. FPTree uses unsorted leaves alongside a bitmap to reduce the number of expensive NVM writes, similar to slotted pages used in database systems. However, there is no mechanism for versioning the updates, which means there is no support for atomic range queries. LB$^+$-Tree is another hybrid persistent index that implements some NVM-specific optimizations such as entry moving and logless splits [35]. uTree is a persistent B$^+$-Tree variant with a focus on lowering tail latency [17]. uTree also designs a hybrid leaf level by introducing auxiliary structures that also live in DRAM.

FPTree, NV-Tree, and LB$^+$-Tree utilize hardware transactional memory (HTM) for better synchronization performance. However, HTM requires specialized hardware, and is usually disabled due to security reasons. Due to these key limiting factors, our PermART and PermART (2-Log) do not utilize HTM.

5.2.2 NVM-Only Indexes. Chen and Jin [16] proposed wB$^+$-Tree, one of the earliest persistent B$^+$-Tree variants. Their work draws key conclusions about the performance characteristics of NVM data structures, which are observed and reported in later works. Hwang et al. [24] introduced FastFair, a lock-based B$^+$-Tree that improves the crash consistency overhead by letting readers tolerate transient inconsistencies. Lee et al. [29] present WORT, a write-optimized variant of the adaptive radix tree for persistent memory. This work WORT maintains that the radix tree's inherent features make it more suitable for NVM. ROART is a persistent variant of the adaptive radix tree by Ma et al. [37] that implements certain optimizations such as leaf compaction and entry compression to reduce the persistence overhead and improve range query performance. Kim et al. [27] introduced PACTree that employs an ART index on top of a B$^+$-Tree-like leaf structure in its data layer. In their paper, PACTree authors describe a number of design goals that are valuable and applicable to any concurrent data structure on NVM. BzTree is a lock-free B$^+$-Tree variant introduced by Binna et al. [10] that utilizes Persistent Multi-word Compare-and-Swap (PMwCAS [46]) to achieve atomic updates on NVM [4]. BzTree's leaves can be partially unsorted to reduce the cost of NVM writes, with reconciliation occurring at particular free space thresholds when read performance is negatively affected. The unsorted portion of leaves in BzTree is designed to use PMwCAS without restructuring the leaves, and it is not concerned with version chain management and range queries.

Srivastava and Brown [45] present ABTree-OCC and ABTree-Elim, concurrent (a,b)-Tree variants that are designed primarily for main memory, but also support persistence.

5.2.3 General Approaches. Lee et al. [30] proposed RECIPE, a framework with which to construct crash-consistent persistent indexes beginning from their volatile counterparts. They utilize their framework to build persistent variants of B^+-Tree and trie-based indexes such as ART [31, 32], HOT [10], Bw-Tree [34, 47], and Masstree [38], as well as a persistent variant of the hash table CLHT [19]. RECIPE is mostly concerned with the correctness of consistency, and does not propose NVM-specific data structure optimizations.

Logging is a common technique in database systems. E.g., Lee and Moon use in-page logging on flash storage, appending per-page physiological log records to log sectors and later merging them to reduce write cost [28]. In contrast, PermART (2-Log) uses in-node logging as an NVM-centric optimization, embedding a fixed number of logical updates within leaf nodes to defer copy-on-write and shorten version chains targeted at versioning and range query performance.

6 Conclusion

This paper presents PermART, a persistent multiversion adaptive radix tree that achieves durable linearizability and supports efficient, atomic range queries, a capability lacking in existing persistent indexes. By integrating recent advances in multiversion concurrency control and proposing a novel in-node logging optimization, PermART significantly reduces copy-on-write overheads. Extensive evaluations demonstrate that PermART delivers state-of-the-art performance in point operations and delivers orders-of-magnitude improvements in range query throughput with concurrency. These results establish PermART as a robust and scalable solution for high-performance persistent indexing in NVM-based systems.

Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada under Discovery Grants 2019-04227 and 2024-04657. We thank the anonymous reviewers for their constructive feedback. We also thank University of Waterloo's CSCF for supporting the computing infrastructure used for this work.

References

[1] [n. d.]. GitHub - sfu-dis/bztree: An open-source BzTree implementation — github.com. https://github.com/sfu-dis/bztree/tree/master. [Accessed 22-09-2025].

[2] [n. d.]. Take advantage of sparse indexes - Amazon DynamoDB — docs.aws.amazon.com.

[3] Maya Arbel-Raviv and Trevor Brown. 2018. Harnessing epoch-based reclamation for efficient range queries. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*, Andreas Krall and Thomas R. Gross (Eds.). ACM, 14–27. doi:10.1145/3178487.3178489

[4] Joy Arulraj, Justin J. Levandoski, Umar Farooq Minhas, and Per-Åke Larson. 2018. BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory. *Proc. VLDB Endow.* 11, 5 (2018), 553–565. doi:10.1145/3187009.3164147

[5] Hillel Avni, Nir Shavit, and Adi Suissa. 2013. Leaplist: lessons learned in designing tm-supported range queries. In *ACM Symposium on Principles of Distributed Computing, PODC '13, Montreal, QC, Canada, July 22-24, 2013*, Panagiota Fatourou and Gadi Taubenfeld (Eds.). ACM, 299–308. doi:10.1145/2484239.2484254

[6] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, Michael J. Carey and Donovan A. Schneider (Eds.). ACM Press, 1–10. doi:10.1145/223784.223785

[7] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (1981), 185–221. doi:10.1145/356842.356846

[8] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control - Theory and Algorithms. *ACM Trans. Database Syst.* 8, 4 (1983), 465–483. doi:10.1145/319996.319998

[9] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley. http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx

[10] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 521–534. doi:10.1145/3183713.3196896

[11] Guy E. Blelloch and Yuanhao Wei. 2024. VERLIB: Concurrent Versioned Pointers. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2024, Edinburgh, United Kingdom, March 2-6, 2024*, Michel Steuwer, I-Ting Angelina Lee, and Milind Chabbi (Eds.). ACM, 200–214. doi:10.1145/3627535.3638501

[12] Trevor Brown and Hillel Avni. 2012. Range Queries in Non-blocking *k*-ary Search Trees. In *Principles of Distributed Systems, 16th International Conference, OPODIS 2012, Rome, Italy, December 18-20, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7702)*, Roberto Baldoni, Paola Flocchini, and Binoy Ravindran (Eds.). Springer, 31–45. doi:10.1007/978-3-642-35476-2_3

[13] Trevor Brown, Aleksandar Prokopec, and Dan Alistarh. 2020. Non-blocking interpolation search trees with doubly-logarithmic running time. In *PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020*, Rajiv Gupta and Xipeng Shen (Eds.). ACM, 276–291. doi:10.1145/3332466.3374542

[14] Wentao Cai, Haosen Wen, H. Alan Beadle, Chris Kjellqvist, Mohammad Hedayati, and Michael L. Scott. 2020. Understanding and optimizing persistent memory allocation. In *ISMM '20: 2020 ACM SIGPLAN International Symposium on Memory Management, ISMM 2020, virtual [London, UK], June 16, 2020*, Chen Ding and Martin Maas (Eds.). ACM, 60–73. doi:10.1145/3381898.3397212

[15] Bapi Chatterjee. 2017. Lock-free Linearizable 1-Dimensional Range Queries. In *Proceedings of the 18th International Conference on Distributed Computing and Networking, Hyderabad, India, January 5-7, 2017*. ACM, 9. http://dl.acm.org/citation.cfm?id=3007771

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

[17] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. 2020. ?Tree: a Persistent B+-Tree with Low Tail Latency. *Proc. VLDB Endow.* 13, 11 (2020), 2634–2648. http://www.vldb.org/pvldb/vol13/p2634-chen.pdf

[18] Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-free concurrent data structures. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 373–385.

[19] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*, Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas (Eds.). ACM, 631–644. doi:10.1145/2694344.2694359

[20] Panagiota Fatourou, Elias Papavasileiou, and Eric Ruppert. 2019. Persistent Non-Blocking Binary Search Trees Supporting Wait-Free Range Queries. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2019, Phoenix, AZ, USA, June 22-24, 2019*, Christian Scheideler and Petra Berenbrink (Eds.). ACM, 275–286. doi:10.1145/3323165.3323197

[21] Alan D. Fekete, Dimitrios Liarokapis, Elizabeth J. O'Neil, Patrick E. O'Neil, and Dennis E. Shasha. 2005. Making snapshot isolation serializable. *ACM Trans. Database Syst.* 30, 2 (2005), 492–528. doi:10.1145/1071610.1071615

[22] Yuliang He, Duo Lu, Kaisong Huang, and Tianzheng Wang. 2022. Evaluating Persistent Memory Range Indexes: Part Two. *Proc. VLDB Endow.* 15, 11 (2022), 2477–2490. doi:10.14778/3551793.3551808

[23] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. doi:10.1145/78969.78972

[24] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies, FAST 2018, Oakland, CA, USA, February 12-15, 2018*, Nitin Agrawal and Raju Rangaswami (Eds.). USENIX Association, 187–200. https://www.usenix.org/conference/fast18/presentation/hwang

[25] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9888)*, Cyril Gavoille and David Ilcinkas (Eds.). Springer, 313–327. doi:10.1007/978-3-662-53426-7_23

[26] Alfons Kemper, Thomas Neumann, Jan Finis, Florian Funke, Viktor Leis, Henrik Mühe, Tobias Mühlbauer, and Wolf Rödiger. 2013. Processing in the Hybrid OLTP & OLAP Main-Memory Database System HyPer. *IEEE Data Eng. Bull.* 36, 2 (2013), 41–47. http://sites.computer.org/debull/A13june/hyper1.pdf

[27] Wook-Hee Kim, Madhava Krishnan Ramanathan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 424–439. doi:10.1145/3477132.3483589

[28] Sang-Won Lee and Bongki Moon. 2007. Design of flash-based DBMS: an in-page logging approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou (Eds.). ACM, 55–66. doi:10.1145/1247480.1247488

[29] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. 2017. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 - March 2, 2017*, Geoff Kuenning and Carl A. Waldspurger (Eds.). USENIX Association, 257–270. https://www.usenix.org/conference/fast17/technical-sessions/presentation/lee-se-kwon

[30] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: converting concurrent DRAM indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 462–477. doi:10.1145/3341301.3359635

[31] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou (Eds.). IEEE Computer Society, 38–49. doi:10.1109/ICDE.2013.6544812

[32] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN 2016, San Francisco, CA, USA, June 27, 2016*. ACM, 3:1–3:8. doi:10.1145/2933349.2933352

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

[34] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou (Eds.). IEEE Computer Society, 302–313. doi:10.1109/ICDE.2013.6544834

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

[36] Mengxing Liu, Jiankai Xing, Kang Chen, and Yongwei Wu. 2019. Building Scalable NVM-based B+tree with HTM. In *Proceedings of the 48th International Conference on Parallel Processing* (Kyoto, Japan) *(ICPP '19)*. Association for Computing Machinery, New York, NY, USA, Article 101, 10 pages. doi:10.1145/3337821.3337827

[37] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. 2021. ROART: Range-query Optimized Persistent ART. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, Marcos K. Aguilera and Gala Yadgar (Eds.). USENIX Association, 1–16. https://www.usenix.org/conference/fast21/presentation/ma

[38] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, Pascal Felber, Frank Bellosa, and Herbert Bos (Eds.). ACM, 183–196. doi:10.1145/2168836.2168855

[39] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 677–689. doi:10.1145/2723372.2749436

[40] 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, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 371–386. doi:10.1145/2882903.2915251

[41] Christos H. Papadimitriou and Paris C. Kanellakis. 1984. On Concurrency Control by Multiple Versions. *ACM Trans. Database Syst.* 9, 1 (1984), 89–99. doi:10.1145/348.318588

[42] Erez Petrank and Shahar Timnat. 2013. Lock-Free Data-Structure Iterators. In *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8205)*, Yehuda Afek (Ed.). Springer, 224–238. doi:10.1007/978-3-642-41527-2_16

[43] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM,

1981–1984. doi:10.1145/3299869.3320212

[44] David P. Reed. 1978. *Naming and synchronization in a decentralized computer system.* Ph. D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA, USA. https://hdl.handle.net/1721.1/16279

[45] Anubhav Srivastava and Trevor Brown. 2022. Elimination (a,b)-trees with fast, durable updates. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 416–430. doi:10.1145/3503221.3508441

[46] Tianzheng Wang, Justin J. Levandoski, and Per-Åke Larson. 2018. Easy Lock-Free Indexing in Non-Volatile Memory. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 461–472. doi:10.1109/ICDE.2018.00049

[47] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 473–488. doi:10.1145/3183713.3196895

[48] Yuanhao Wei, Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. 2021. Constant-time snapshots with applications to concurrent data structures. In *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*, Jaejin Lee and Erez Petrank (Eds.). ACM, 31–46. doi:10.1145/3437801.3441602

[49] Kjell Winblad, Konstantinos Sagonas, and Bengt Jonsson. 2021. Lock-free Contention Adapting Search Trees. *ACM Trans. Parallel Comput.* 8, 2 (2021), 10:1–10:38. doi:10.1145/3460874

[50] 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, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*, Jiri Schindler and Erez Zadok (Eds.). USENIX Association, 167–181. https://www.usenix.org/conference/fast15/technical-sessions/presentation/yang

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

Received October 2025; revised January 2026; accepted February 2026