Building a Bw-Tree Takes More Than Just Buzz Words

Ziqi Wang
Carnegie Mellon University
ziqiw@cs.cmu.edu

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

Hyeontaek Lim
Carnegie Mellon University
hl@cs.cmu.edu

Viktor Leis
TU München
leis@in.tum.de

Huanchen Zhang
Carnegie Mellon University
huanche1@cs.cmu.edu

Michael Kaminsky
Intel Labs
michael.e.kaminsky@intel.com

David G. Andersen
Carnegie Mellon University
dga@cs.cmu.edu

ABSTRACT

In 2013, Microsoft Research proposed the Bw-Tree (humorously termed the "Buzz Word Tree"), a lock-free index that provides high throughput for transactional database workloads in SQL Server's Hekaton engine. The Bw-Tree avoids locks by appending delta record to tree nodes and using an indirection layer that allows it to atomically update physical pointers using compare-and-swap (CaS). Correctly implementing this techniques requires careful attention to detail. Unfortunately, the Bw-Tree papers from Microsoft are missing important details and the source code has not been released.

This paper has two contributions: First, it is the missing guide for how to build a lock-free Bw-Tree. We clarify missing points in Microsoft's original design documents and then present techniques to improve the index's performance. Although our focus here is on the Bw-Tree, many of our methods apply more broadly to designing and implementing future lock-free in-memory data structures. Our experimental evaluation shows that our optimized variant achieves 1.1–2.5× better performance than the original Microsoft proposal for highly concurrent workloads. Second, our evaluation shows that despite our improvements, the Bw-Tree still does *not* perform as well as other concurrent data structures that use locks.

ACM Reference Format:
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 2018 International Conference on Management of Data (SIGMOD'18).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3183713.3196895

1 INTRODUCTION

Lock-free data structures are touted as being ideal for today's multi-core CPUs. They are, however, notoriously difficult to implement for several reasons [10]. First, writing efficient and robust lock-free[1] code requires the developer to figure out all possible race conditions, the interactions between which can be complex. Furthermore, The point that concurrent threads synchronize with each other are usually not explicitly stated in the serial version of the algorithm. Programmers often implement lock-free algorithms incorrectly and end up with busy-waiting loops. Another challenge is that lock-free data structures require safe memory reclamation that is delayed until all readers are finished with the data. Finally, atomic primitives can be a performance bottleneck themselves if they are used carelessly.

One example of a lock-free data structure is the Bw-Tree from Microsoft Research [29]. The high-level idea of the Bw-Tree is that it avoids locks by using an indirection layer that maps logical identifiers to physical pointers for the tree's internal components. Threads then apply concurrent updates to a tree node by appending delta records to that node's modification log. Subsequent operations on that node must replay these deltas to obtain its current state.

The indirection layer and delta records provide two benefits. First, it avoids coherence traffic of locks by decomposing every global state change into atomic steps. Second, it incurs fewer cache invalidations on a multi-core CPU because threads append delta records to make changes to the index instead of overwriting existing nodes. The original Bw-Tree paper [29] claims that this lower synchronization and cache coherence overhead provides better scalability than lock-based indexes.

To the best of our knowledge, however, there is no comprehensive evaluation of the Bw-Tree. The original paper lacks detailed descriptions of critical components and runtime operations. For example, they do not provide a scalable solution for safe memory reclamation or efficient iteration. Microsoft's Bw-Tree may support these features, but the implementation details are unknown. This paper aims to be a more thorough investigation of the Bw-Tree: to supply the missing details, propose improvements, and to provide a more comprehensive evaluation of the index.

Our first contribution is a complete design for how to build an in-memory Bw-Tree. We present the missing features required for a correct implementation, including important corner cases missing from the original description of the data structure. We then present several additional enhancements and optimizations that improve the index's performance. The culmination of this effort is our open-source version called the **OpenBw-Tree**. Our experiments show that the OpenBw-Tree outperforms what we understand to be the original Bw-Tree design by 1.1–2.5× for insert-heavy workloads and by 1.1–1.4× for read-heavy workloads.

Our second contribution is to compare the OpenBw-Tree against four other state-of-the-art in-memory data structures: (1) SkipList [8], (2) Masstree [31], (3) a B+Tree with optimistic lock coupling [22] and (4) ART [20] with optimistic lock coupling [22]. Our results

[1] In this the paper, we always use the term "lock" when referring to "latch".

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3196895

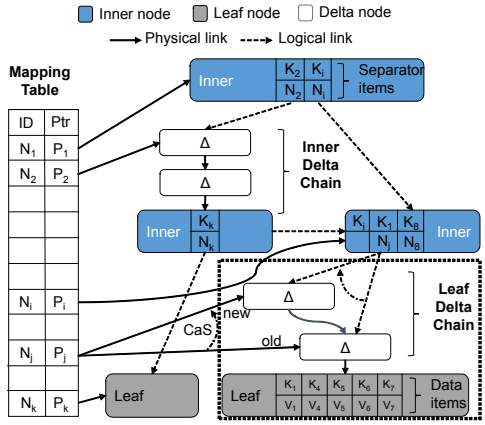


Figure 1: Architecture Overview – An instance of a Bw-Tree with its internal logical links, Mapping Table links, and an ongoing CaS operation on the leaf Delta Chain.

show that despite previous claims of lock-free indexes being superior to lock-based indexes on multi-core CPUs, the overhead of the Bw-Tree's indirection layer and delta records causes it to under-perform the lock-based indexes by 1.5–4.5×.

2 BW-TREE ESSENTIALS

Although the Bw-Tree has been covered in previous papers [25, 28–30], additional details are needed to understand our discussion in Sections 3 and 4. We assume that the Bw-Tree is deployed inside of a database management system (DBMS) with a thread pool, and has worker threads accessing the index to process queries. If non-cooperative garbage collection is used, the DBMS also launches one or more background threads periodically to perform garbage collection on the index.

The most prominent difference between the Bw-Tree and other B+Tree-based indexes is that the Bw-Tree avoids directly editing tree nodes because it causes cache line invalidation. Instead, it stores modifications to a node in a delta record (e.g., insert, update, delete), and maintains a chain of such records for every node in the tree. This per-node structure, called a Delta Chain, allows the Bw-Tree to perform atomic updates via CaS. The Mapping Table serves as an indirection layer that maps logical node IDs to physical pointers, making atomic updates of several references to a tree node possible.

The mapping table works as follows: As shown in Fig. 1, every node in the Bw-Tree has a unique logical node ID (64-bit). Instead of using pointers, nodes refer to other nodes using these IDs (*logical links*). When a thread needs the physical location of a node, it consults the Mapping Table to translate a node ID to its memory address. Thus, the Mapping Table allows threads to atomically change the physical location of a node without having to acquire locks: a single atomic compare-and-swap (CaS) instruction changes multiple logical links to a node throughout the index.

2.1 Base Nodes and Delta Chains

A *logical node* (called "virtual node" in [29]) in the Bw-Tree has two components: a base node and a Delta Chain. There are two types of base nodes: an *inner base node* that holds a sorted (key, node ID) array, and a *leaf base node* that holds a sorted (key, value) array. Initially, the Bw-Tree consists of two nodes, an empty leaf base node, and an inner base node that contains one separator item referring to the empty leaf node. Base nodes are immutable.

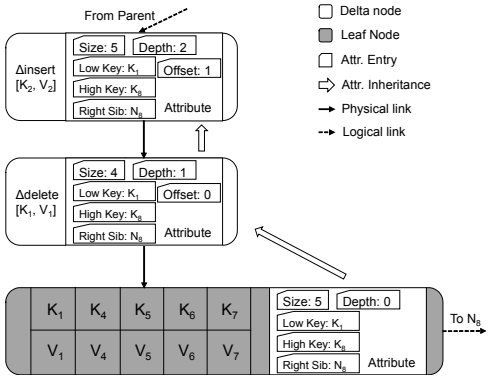


Figure 2: Delta Records Overview – A more detailed illustration of a logical leaf node from Fig. 1 with its base node and two delta nodes.

Attribute	Description
low-key	The smallest key stored at the logical node. In a node split, the low-key of the right sibling is set to the split key. Otherwise, it is inherited from the element's predecessor.
high-key	The smallest key of a logical node's right sibling. Δ**split** records use the split key for this attribute. Δ**merge** records use the high-key of the right branch. Otherwise, it is inherited from the element's predecessor.
right-sibling	The ID of the logical node's right sibling.
size	The number of items in the logical node. It is incremented for Δ**insert** records and decremented for Δ**delete** records.
depth	The number of records in the logical node's Delta Chain.
offset	The location of the inserted or deleted item in the base node if they were applied to the base node. Only valid for Δ**insert** and Δ**delete** records.

Table 1: Node Attributes – The list of the attributes that are stored in the logical node's elements (i.e., base node or delta records).

As shown in Fig. 2, a Delta Chain is a singly linked list that contains a chronologically-ordered history of the modifications made to the base node. The entries in the Delta Chain are connected using physical pointers, with the tail pointing to the base node. Both the base node and its delta records contain additional meta-data that represent the state of the logical node at that point in time (see Table 1). That is, when a worker thread updates a logical node, they compute the latest attributes of the logical node and store them in the delta record. The threads then use this information when navigating the tree or when performing structural modifications to avoid having to traverse (and replay) a node's Delta Chain.

As we next describe, a node's logical link in the Mapping Table points to either a Delta Chain entry or the base node. Threads append new entries to the head of a node's Delta Chain and then update the physical address in the Mapping Table.

2.2 Mapping Table

The Bw-Tree borrows the B^{link}-Tree design where a node can have two inbound pointers, one from the parent and one from the left sibling [19]. Updating these pointers atomically requires either hardware support (e.g., transactional memory [30]) or complex software primitives (e.g., multi-word CaS [2, 12, 14]).

The Bw-Tree's centralized Mapping Table avoids these problems and allows a thread to update all references to a node in a single CaS

instruction that is available on all modern CPUs. If a thread's CaS fails then it aborts its operation and restarts. This restart is transparent to the higher-level DBMS components. Threads always restart an operation by traversing again from the tree's root. Although more sophisticated restart protocols are possible (e.g., restarting from the previous level in the tree), we contend that restarting from the root simplifies the implementation. The nodes that a thread will revisit after a restart will likely be in the CPU cache anyway.

Although this paper focuses only on in-memory behavior of the Bw-Tree, it is worth emphasizing that the mapping table also serves the purpose of supporting log-structured updates when deployed with SSD. Updates to tree nodes will otherwise propagate to all levels without the extra indirection provided by the Mapping Table.

2.3 Consolidation and Garbage Collection

As we described above, worker threads update the Bw-Tree by appending new delta records to the logical nodes' Delta Chains. This means that these Delta Chains are continually growing, which in turn increases the time that it takes for threads to traverse the tree because they must replay the delta records to get the current state of a logical node. To prevent excessively long Delta Chains, worker threads will periodically consolidate a logical node's delta records into a new base node. Consolidation is triggered when Delta Chain's length exceeds some threshold. Microsoft reported that a length of eight was a good setting [29]. Our results in Section 5.3 show that using different thresholds for inner nodes versus leaf nodes yields the best performance.

At the beginning of consolidation, the thread copies the logical node's base node contents to its private memory and then applies the Delta Chain. It then updates the node's logical link in the Mapping Table with the new base node. After consolidation, the index reclaims the old base node and Delta Chain memory after all other threads in the system are finished accessing them. The original Bw-Tree uses a centralized epoch-based garbage collection scheme to determine when it is safe to reclaim memory [25].

2.4 Structural Modification

As with a B+Tree, a Bw-Tree's logical node is subject to overflow or underflow. These cases require **splitting** a logical node with too many items into two separate nodes or **merging** together underfull nodes into a new node. We briefly describe the Bw-Tree's *structural modification* (SMO) protocols for handling node splits and merges without using locks. The main idea is to use special delta records to represent internal structural modifications.

The SMO operation is divided into two phases: a *logical* phase which appends special deltas to notify other threads of an ongoing SMO, and a *physical* phase which actually performs the SMO. In the logical phase, some thread t appends a Δ**insert**, Δ**merge** or Δ**remove** to the virtual node, updating its attributes such as the high key and next node ID. The updated attributes guarantee that other worker threads always observe consistent virtual node states during their traversal. In the following physical phase, t' (not necessarily the same thread as t) will split or merge the node, and then replace the old virtual node with the new node via CaS.

Although the Bw-Tree is lock-free, threads are not guaranteed to make progress because of failed CaS. One way to alleviate this starvation is for threads to cooperatively complete a multi-stage SMO, which in known as the *help-along protocol* [29]. Threads must help complete an unfinished SMO before the corresponding virtual node can be traversed.

3 MISSING COMPONENTS

This section introduces our design and implementation of four components that are either missing or lacking details from the Bw-Tree papers. Since we assume that the Bw-Tree will be used in a DBMS, in Section 3.1 we also describe how to support non-unique keys, followed by iterators in Section 3.2. Finally, we discuss how to enable dynamic Mapping Table expansion in Section 3.3.

3.1 Non-unique Key Support

During traversal, Bw-Tree stops at the first leaf delta record that matches the search key, without continuing down the chain. This behavior, however, is incompatible with non-unique key support.

We handle non-unique keys in the OpenBw-Tree by having threads compute the visibility of delta records on-the-fly using two disjoint value sets for a search key, as shown in Fig. 3. The first set ($S_{present}$) contains the values that are already known to be present. The second set ($S_{deleted}$) contains the values that are known to be deleted. While traversing the leaf Delta Chain, if a worker thread finds an insert delta record with key K and value V where $V \notin S_{deleted}$, it adds V to $S_{present}$. If the thread finds a delete delta record with key K and value V where $K \notin S_{present}$, it adds V to $S_{deleted}$. Update deltas are processed as a delete delta of the old value followed by an insert delta of the updated value. Although we largely ignore update deltas in the discussion, they are posted when a worker thread modifies a key-value pair. After reaching the base node with values for K, denoted S_{base}, the thread returns $S_{present} \cup (S_{base} - S_{deleted})$ as the values of K. This result returns all values whose existence has not been overridden by any earlier delta records in the Delta Chain.

Computing $S_{present}$ and $S_{deleted}$ is inexpensive, and is performed only when traversing a leaf node's Delta Chain. A delta record may insert only one value to either set. Because the max chain length is typically small and fixed (e.g., 24 [29]), OpenBw-Tree stack allocates $S_{present}$ and $S_{deleted}$ as two variable-length arrays.

3.2 Iteration

To support range scans on an index, the DBMS's execution engine uses an iterator. This interface is complex to implement in the Bw-Tree since it must support concurrent operations without locks. Tracking a thread's current location in an iterator is difficult when other threads are simultaneously inserting and deleting items. Furthermore, concurrent SMOs (Section 2.4) make it more challenging when the iterator moves from one logical leaf node to its neighbor.

To overcome these problems, OpenBw-Tree's iterator does not operate directly on tree nodes. Instead, each iterator maintains a private, read-only copy of a logical node that enables consistent and fast random access. The iterator also maintains an offset of the current item in this node, as well as information for determining the next item. As the iterator moves forward or backward, if the current private copy has been exhausted, the worker thread starts

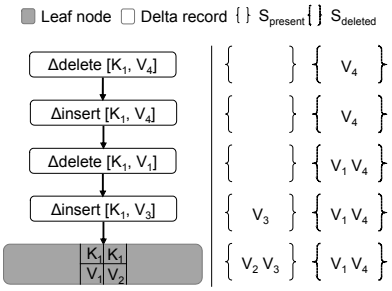


Figure 3: Non-unique Key Support – The two sets ($S_{present}$, $S_{deleted}$) track the visibility of Δ**insert** and Δ**delete** records in the Delta Chain.

a new traversal from the root, using the current low key or high key to reach the previous or the next sibling node.

3.3 Mapping Table Expansion

Since every thread accesses the Bw-Tree's Mapping Table multiple times during traversal, it is important that it is not a bottleneck. Storing the Mapping Table as an array of physical pointers indexed by the node ID is the fastest data structure. But using a fixed-size array makes it difficult to dynamically resize the Mapping Table as the number of items in the tree grows and shrinks. This last point is the problem that we address here.

The OpenBw-Tree pre-allocates large virtual address space for the Mapping Table without requesting backing physical pages. This allows it to leverage the OS to lazily allocate physical memory without using locks; this technique was previously used in the KISS-Tree [18]. As the index grows, a thread may attempt to access one of the Mapping Table's pages that have not been mapped to the physical memory, incurring a page fault. The OS then allocates a new empty physical page for the virtual page. In practice, the amount of virtual address space we reserve is estimated using the total amount of physical memory and the lower bound of virtual node size.

Although this approach makes it easy to increase the number of entries in the Mapping Table as the index grows, it does not solve the problem of shrinking the size of the Mapping Table. To the best of our knowledge, there is no lock-free way of doing this. The only way to shrink the Mapping Table is to block all worker threads and rebuild the index.

4 COMPONENT OPTIMIZATION

A good-faith implementation of the data structure described in original Bw-Tree paper design can further be improved. We present our optimizations for the OpenBw-Tree's key components to improve its performance and scalability. As we show in Section 5, these optimizations increase the index's throughput by 1.1–2.5× for multi-threaded environments.

4.1 Delta Record Pre-allocation

As described in Section 2.1, the Delta Chain in Bw-Tree is a linked list of delta records that is allocated on the heap. Traversing this linked list is slow because a thread can incur a cache miss for each pointer dereference. Additionally, excessive allocations of small objects create contention in the allocator, which becomes a scalability bottleneck as the number of cores increases.

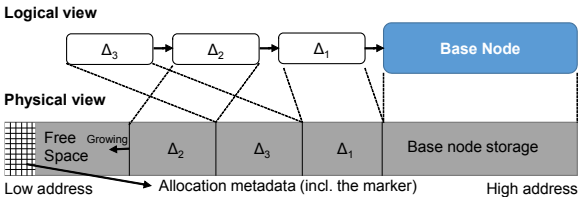


Figure 4: Pre-allocated Chunk – This diagram depicts the logical view and physical view of a OpenBw-Tree node. Slots are acquired by threads using a CaS on the marker, which is part of the allocation metadata on lower-address of the chunk.

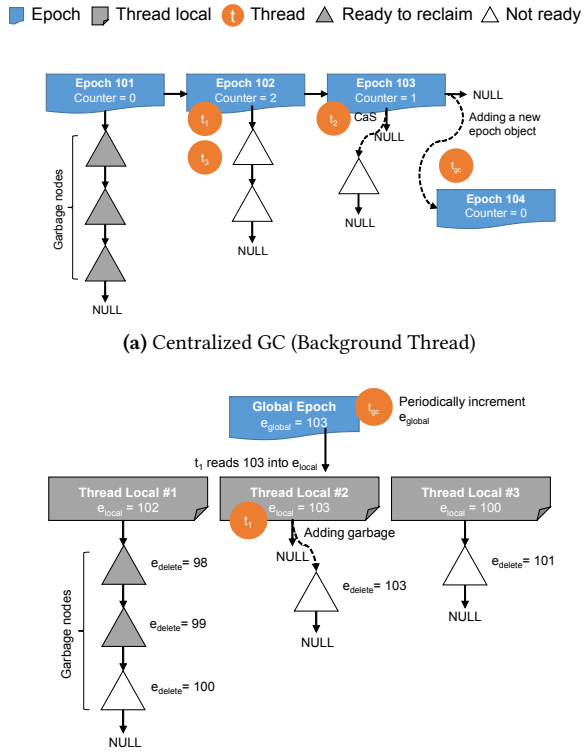
To avoid these problems, the OpenBw-Tree pre-allocates the delta records inside of each base node. As shown in Fig. 4, it stores the base node in the high-address end of the pre-allocated chunk and stores the delta records from high to low addresses (right-to-left in the figure). Each chain also maintains an *allocation marker* that points to the last delta record or the base node. When a worker thread claims a slot, it decrements this marker by the number of bytes for the new delta record using an atomic subtraction. If the pre-allocated area is full, then this triggers a node consolidation.

This reverse-growth design is optimized for efficient Delta Chain traversals. Reading delta records in the new-to-old order is likely to (but not always) access memory linearly from low to high addresses, which is ideal for modern CPUs with hardware memory prefetching. But threads must traverse a node's Delta Chain by following each delta record's pointer to find the next entry, rather than just scanning from low to high addresses. This is because the logical order of delta records may not match their physical locations in memory. Slot allocations and Delta Chain appendings are not atomic, permitting multiple threads to interleave them. For example, Fig. 4 shows that delta record Δ_3 was logically added to the node before delta record Δ_2, but Δ_3 appears after Δ_2 physically in memory.

4.2 Garbage Collection

The OpenBw-Tree adopts a garbage collection (GC) scheme that is similar to the one used in Silo [34] and Deuteronomy [24]. The epoch-based GC scheme of the original Bw-Tree [25] provides safe memory reclamation that prevents the index from reusing memory when there may exist a thread that is accessing it. With this approach, the index maintains a list of global epoch objects, and appends new epoch objects to the end of this list at fixed intervals (e.g., every 40 ms). Every thread must enter the epoch by enrolling itself in the current epoch object before it accesses the index's internal data structures (e.g., performing a key lookup). When the thread completes its operation, it removes itself from the epoch it has entered. Any objects that are marked for deletion by a thread are added into the garbage list of the current epoch. Once all threads exit an epoch, the index's GC component can then reclaim the objects in that epoch that are marked for deletion.

Fig. 5a illustrates the centralized GC scheme with three active epochs, three worker threads (t_1, t_2, t_3), and a background GC thread (t_{gc}). In this diagram, t_2 adds a new node to the garbage list of epoch *103*. At the same time, the GC thread t_{gc} installs a new epoch object to the epoch list. Since the counter inside epoch *101* has reached zero, t_{gc} will reclaim all entries in its garbage list.



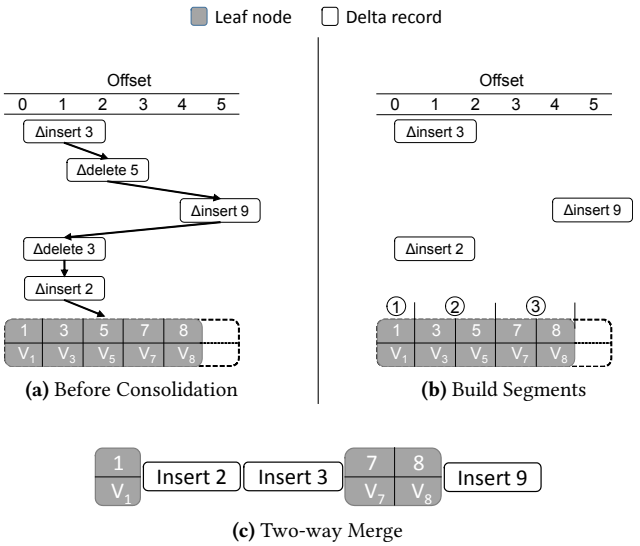
(a) Centralized GC (Background Thread)

(b) Decentralized GC (Cooperative)

Figure 5: Garbage Collection – Illustrations of the centralized GC scheme using a background thread and a cooperative decentralized GC scheme.

Such a centralized GC implementation has poor scalability because the worker threads enroll in an epoch by incrementing a counter that represents the number of active threads in the current epoch [33]. This becomes bottleneck when there are many threads because of cache coherence traffic [34].

The OpenBw-Tree adopts a decentralized GC scheme where threads avoid writing to the global memory [24, 34]. The index maintains a global epoch (e_{global}). Each worker thread also maintains a private epoch (e_{local}) and linked list of pointers to objects that the thread marked for deletion (l_{local}). Using the same example as above, Fig. 5b depicts this decentralized GC scheme. t_1 is the only active worker thread in the diagram, and it is adding a new garbage node, tagged with the current global epoch, into its l_{local}. At the beginning of a new index operation, a thread copies the current e_{global} to its e_{local}. Whenever that thread creates garbage (e.g., removing a logical node), it appends the pointer to that garbage object tagged with the latest value of e_{global} to its l_{local}. When the thread finishes its operation, it copies the latest e_{global} to e_{local} again and then initiates GC. The thread retrieves the e_{local} from all the other threads and then searches its l_{local} for the objects that are tagged with an epoch that is less than the minimum e_{local}. It then removes any objects that satisfy this condition from its l_{local} and reclaims their memory. This is safe to do because the epochs act as a watermark that guarantee that no other thread is accessing them. To make progress, the e_{global} is periodically incremented by one by the DBMS in a background thread.



(a) Before Consolidation

(b) Build Segments

(c) Two-way Merge

Figure 6: Fast Consolidation – This diagram depicts the fast consolidation algorithm. The base node is first divided into segments using the offset attribute in the delta records. Then a two-way merge applies all valid insert deltas onto the new base node after copying live elements from the old base node.

4.3 Fast Consolidation

The node consolidation scheme described in Section 2.3 can be expensive for write-heavy workloads. For these applications, threads create many delta records and thus must frequently consolidate long Delta Chains. On consolidation, a thread has to first replay the Delta Chain to collect all (*key*, *value*) or (*key*, *node ID*) items in the logical node and then sort them. We present a faster consolidation algorithm that reduces the sorting overhead.

As depicted in Fig. 6, the consolidation algorithm has two steps. The first step (Fig. 6b) reuses the Delta Chain replay technique of non-unique key support (see Section 3.1) to gather effective changes to the base node. As a thread traverses a Delta Chain, it adds the delta records' keys to $S_{present}$ and $S_{present}$ to find which insertions and deletions are not overridden by newer delta records. At the end of this step, the old base node is divided into segments. Then, as shown in Fig. 6c, a two-way merge combines **Δinsert** records and segments from the old base node into the new base node.

To divide the old base node's item array into segments, the OpenBw-Tree uses the delta record's offset attribute (Section 2.1). This attribute is valid only for **Δinsert** and **Δdelete** records, and it stores the location of K in the current base node (not in the logical node that delta records have modified). For an insertion, the offset stores where K would appear if this insertion is done in the base node (ignoring any other delta records). For a deletion, the offset is the position of the existing K in the base node. If a worker thread performs a base node search to create a delta record, the thread stores the key search within the base node as offset; otherwise, it copies the previous delta record's offset to the new delta record's offset. For non-unique key indexes, however, this is more complicated. One simplification is to use the smallest offset as the value of offset among all items having keys that is equal to K.

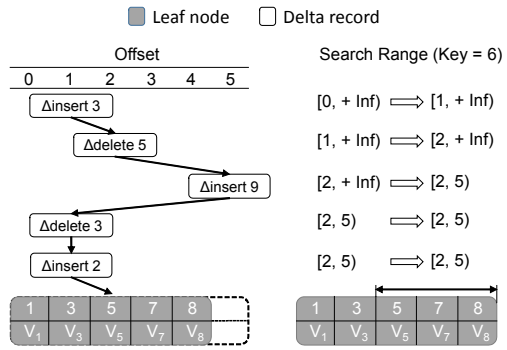


Figure 7: Node Search Shortcuts – This diagram illustrates how thread makes use of the offset attribute. On the base level the thread searches only three elements instead of five.

The OpenBw-Tree then divides the old base node's item array into segments using the offset field in delta records. A *segment* is a contiguous slice [*start*, *end*) of the base node items that does not contain any insertion and deletion between two items in it. The thread first sorts all items in $S_{present}$ and $S_{deleted}$ by their offsets and keys; sorting both sets is fast because they have a small number of elements. The thread starts with a single segment [0, *n*) (i.e., the entire array with *n* items) and applies the following rules:

- **Rule #1:** For an inserted item, an existing segment [*s*,*e*) is broken into the segments [*s*,offset) and [offset,*e*).
- **Rule #2:** For a deleted item, an existing segment [*s*, *e*) is broken into the segments [*s*, offset) and [offset +1, *e*).
- **Rule #3:** However, if a delete delta removes an item that does not exist in the base node (i.e., deleting an item that is newly inserted by an earlier delta), the thread ignores this deletion.

Node consolidation finishes with a two-way merge between the items of $S_{present}$ and the segments.

4.4 Node Search Shortcuts

As with any tree-based index, the Bw-Tree favors large nodes because it reduces the height of the tree and the number of virtual nodes. This in turn reduces the number of mapping table lookups and delta chain traversals. The downside, however, is that large base nodes reduces in-node search performance. Threads that reach the base node during a Delta Chain traversal conduct a binary search to find the target item. If the base node spans multiple cache lines, the first few probes of a binary search will likely incur cache misses.

We can optimize this last step by using the offset attribute to narrow down the range a thread must search in the base node, as shown in Fig. 7. The technique we use is known as micro-indexing [27]. When a worker thread traverses a Delta Chain, it initializes the binary search range [*min*, *max*] it accesses for search key K to [0, +inf). During the traversal, whenever the thread sees an **Δinsert** or **Δdelete** record with key K' and offset, it compares K with K'. If $K{=}K'$ then the range immediately converges to [offset, offset], avoiding the binary search. If offset > *min* and $K{>}K'$, then set *min* to offset. Otherwise, if offset < *max* and $K{<}K'$ then set *max* to offset. For non-unique keys, however, it is unclear how to interpret the offset attribute because it may point to the middle of a span consisting of items with key K. In this case, the index ignores the offset attribute.

5 EXPERIMENTAL EVALUATION

We evaluate the OpenBw-Tree as an in-memory OLTP index structure. We focus on understanding how the optimizations presented in Section 4 improve performance over a good-faith implementation of Microsoft's Bw-Tree. We provide a comparison with other state-of-the-art indexes in Section 6.

We created a testing framework that supports a variety of workloads and hardware configurations. All of our experiments were conducted on a system with two Intel Xeon E5-2680 v2 CPUs (10 threads with 2× HT) with 128 GB RAM. We compiled our framework using g++ (v5.4) with `tcmalloc`. Unless stated otherwise, multi-threaded experiments run on a single CPU socket by pinning threads and restricting memory allocation to that same NUMA node.

5.1 Workloads

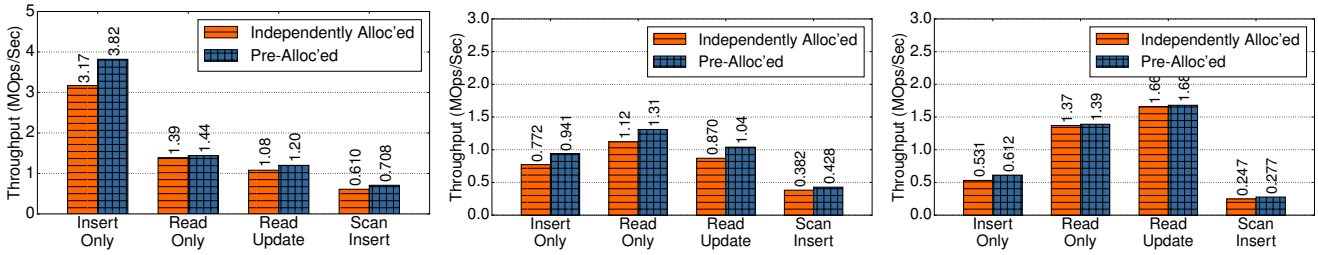
We used a set of Yahoo! Cloud Serving Benchmark (YCSB) microbenchmarks to mimic OLTP index workloads [7]. We used its default workloads **A** (*Read/Update*, 50/50), **C** (*Read-only*), and **E** (*Scan/Insert*, 95/5) with Zipfian distributions, which have skewed access patterns common to OLTP workloads. The average length of YCSB-E scan is 48, with standard deviation 30.13. We measure the initialization phase in each workload and report it as the *Insert-only* workload. For each workload, we tested three key types: 64-bit random integers (*Rand-Int*), 64-bit monotonically increasing integers (*Mono-Int*), and email addresses (*Email*). For integers, we populate the indexes with approximately 52M keys using either *Rand-Int* or *Mono-Int* trace first, and then the same YCSB-A/C/E workloads are run. For email, we insert the keys in the order they are stored in the trace file. The emails are derived from a real-world database; we store them as fixed-length 32-byte strings. There are around 27M entries in the Email workload. All values are 64-bit integers to represent tuple pointers. We run each trial five times and report the median. If the variation is high, error bars are also drawn.

Unless otherwise stated, we configured the Bw-Tree and OpenBw-Tree index to enforce unique keys. OpenBw-Tree uses a max inner node size of 64 entries and max leaf node size of 128 entries. The max Delta Chain length is 2 for inner nodes and 24 for leaf nodes. We set the index's GC threshold at 1024 entries in a thread's local garbage list. The OpenBw-Tree uses the cooperative GC scheme (see Section 4.2), while the Bw-Tree uses the centralized scheme with a background thread. The GC interval is set to 40 ms. We picked these numbers empirically such that performance variances of both Bw-Tree and OpenBw-Tree are small.

5.2 Delta Record Pre-allocation

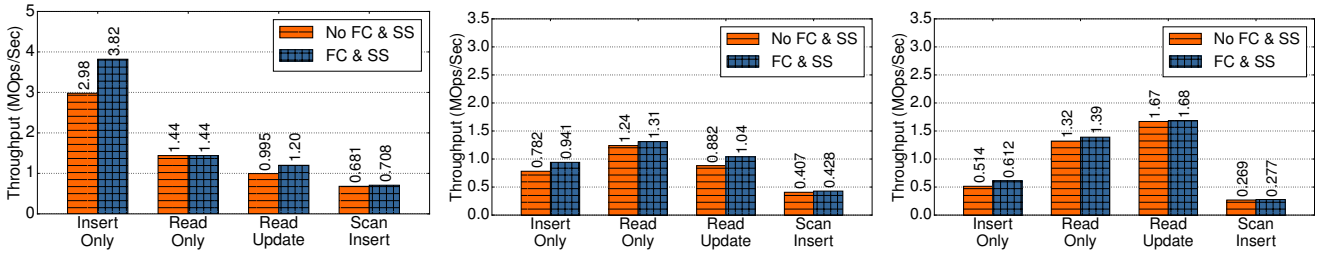
In this first experiment, we compare the pre-allocation optimization against the baseline approach where threads independently allocate delta records. The pre-allocated space on each node is set to the max Delta Chain length times the largest delta record size (176 bytes and 1728 bytes for inner and leaf base nodes, respectively).

Fig. 8 shows single-threaded performance numbers for the different workload configurations. Pre-allocation improves performance for all workloads by up to 22%. One reason for this improvement



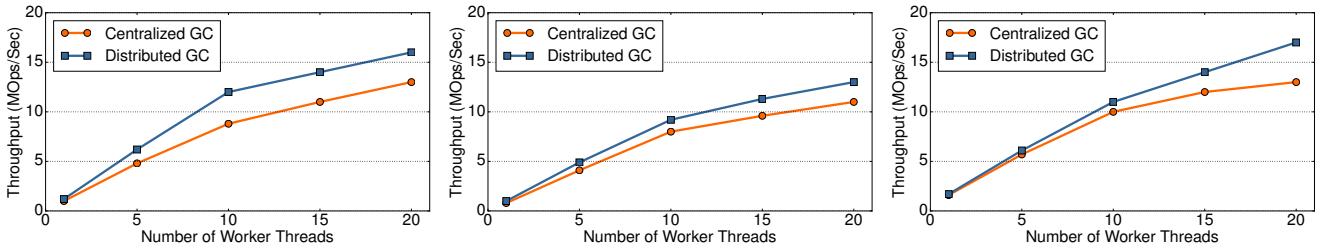
(a) Mono-Int Keys **(b)** Rand-Int Keys **(c)** Email Keys

Figure 8: Delta Record Pre-allocation (Single-Threaded) – Throughput with and without the pre-allocation optimization (Section 4.1).



(a) Mono-Int Keys **(b)** Rand-Int Keys **(c)** Email Keys

Figure 9: Fast Consolidation & Search Shortcuts (Single-Threaded) – Throughput with and without the fast consolidation (Section 4.3) and node search shortcut (Section 4.4) optimizations.



(a) Mono-Int Keys – Read/Update **(b)** Rand-Int Keys – Read/Update **(c)** Email Keys – Read/Update

Figure 10: Garbage Collection Scalability (Multi-Threaded) – All threads are pinned to NUMA node 0.

is better locality during Delta Chain traversal. To verify this, we ran these trials again with `perf` to collect hardware performance counters. We observe 24% fewer L1 misses and 22% fewer L3 misses when pre-allocation is enabled. Second, pre-allocation reduces the number of (small) memory allocations, which relieves contention in the allocator and memory fragmentation for insert-heavy workloads. The Mono-Int Read-only workload, on the other hand, shows little improvement. This is because after the Delta Chain is short, and hence there is almost no chain traversal needed on any level.

The downside of pre-allocation is wasted memory in the pre-allocated area because some slots are not used due. As demonstrated in Table 2, after the Rand-Int Insert-only workload, only 63% and 81% of pre-allocated storage is utilized for leaf and inner nodes, respectively. In contrast, for Mono-Int Insert-only, utilization is almost 100% because of the regular insert pattern. The space under-utilization is not significantly affected by changes in the workload size; we observed only small variance (< 1%).

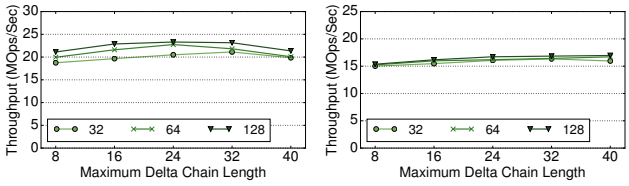
5.3 Fast Consolidation & Search Shortcuts

We next evaluate the fast consolidation (Section 4.3) and the node search shortcut optimization (Section 4.4). We compare these two optimizations against the baseline approach where threads sort the base node after replaying the delta chain. Threads also always search the entire leaf data item array using binary search. As in Section 5.2, we execute the trials with a single thread to avoid concurrency overhead.

The results in Fig. 9 show that our improved consolidation increases the performance of insert- and update-heavy workloads. For Insert-only workloads, performance increases between 19% (Email) and 28% (Rand-Int). For Read/Update and Scan/Insert workloads, the increase is 18–20%. Overall, fast consolidation is effective at reducing consolidation time, which increases the performance of insert-heavy workloads. We also see that the search shortcut optimization improves the performance of random lookups. For example, for random integers, we observed a Read-only performance

Name	Mono-Int	Rand-Int	Mono-HC
Avg. IDCL	0	0.46	0
Avg. LDCL	0	11.38	0.34
Avg. INS	32.04	44.18	32.00
Avg. LNS	72.00	99.84	72.34
Abort Rate	1.05%	1.44%	1078.63%
Avg. IPU	99.96%	81.29%	99.99%
Avg. LPU	100%	63.09%	97.56%

Table 2: OpenBw-Tree Statistics – Statistics from running Insert-only workload with 20 threads (IDCL: Inner Delta Chain Length; LDCL: Leaf Delta Chain Length; INS: Inner Node Size (number of key-ID items); LNS: Leaf Node Size (number of key-value items); IPU: Inner Pre-allocation Utilization (fraction of bytes pre-allocated); LPU: Leaf Pre-allocation Utilization (fraction of bytes pre-allocated); HC: Mono-HC workload type)



(a) Mono-Int Keys – Insert-only **(b)** Mono-Int Keys – Read/Update

Figure 11: Delta Chain Length & Node Size – OpenBw-Tree with varying node sizes and Delta Chain length configurations (20 worker threads).

increase of 6%. For Mono-Int keys, the Read-only workload's performance does not increase because, for monotonic insertions, the Delta Chain is usually short (Table 2). Also, the delta records always contain the last few keys in the leaf node. These optimizations incur a small memory overhead (< 5%) because adding an extra offset attribute increases nodes and delta record sizes slightly.

5.4 Garbage Collection Scalability

We next compare the original GC scheme from the Bw-Tree paper with the decentralized scheme described in Section 4.2. In this experiment, we use the Read/Update workload for all key types. We vary the number of threads in each trial, and measure the throughput. All threads are pinned to NUMA node 0.

Fig. 10 shows that under high levels of concurrency, our modified GC scheme is superior due to the elimination of the centralized epoch counter. The effect is more pronounced for Mono-Int keys (1.3× improvement with 20 threads) than for Rand-Int, because Mono-Int workloads run faster, and therefore, cache line invalidation happens more frequently, as worker threads increment and decrement the centralized epoch counter on entry and on exit of the index's routine.

5.5 Delta Chain Length & Node Size

We evaluate how the OpenBw-Tree's Delta Chain length and node size settings affect its performance. The former determines the number of records that can exist in the Delta Chain before it is consolidated, while the latter determines the number of key-value pairs that the base node's item array can store. Since these two parameters are highly dependent, we vary them together while executing Insert-only and Read/Update workloads using the Mono-Int keys. All experiments are run using 20 worker threads.

We see in Fig. 11b that the index's performance increases slightly for the Read/Update workload as node size and Delta Chain length threshold increases. If the Delta Chain length threshold is too large, however, the performance drops. where the node size is 32 and Delta Chain length threshold is 40, the performance will drop.

For the Insert-only workload, larger tree nodes results in faster inserts. The reason why the OpenBw-Tree favors larger nodes is because each Mapping Table lookup may incur a cache miss, implying that more shallow tree could have fewer cache misses than a deeper tree with smaller nodes. Furthermore, since the OpenBw-Tree does not move node elements for every insert, larger nodes do not necessarily imply frequent cache line invalidations during Insert-only workloads. In Table 3, it is shown that the OpenBw-Tree demonstrates better cache locality than a B+Tree for insertion.

From Fig. 11, we can conclude that the optimal Delta Chain length threshold for OpenBw-Tree is between 32–40, which is higher than Microsoft's recommended value of 8 [29].

We repeated the same experiment on the Bw-Tree. It is suggested by the results that the optimal Delta Chain length threshold is between 16 and 24 for both Insert-only and Read/Update workloads. We believe the difference on optimal Delta Chain lengths is related to the pre-allocation optimization, because it reduces the overhead of delta chain traversal. Furthermore, the type of keys also has an effect on the optimal threshold. For Email Insert-only workloads on OpenBw-Tree, the optimal Delta Chain length threshold is between 16–24, while for Read/Update workloads it is 32–40.

5.6 Discussion

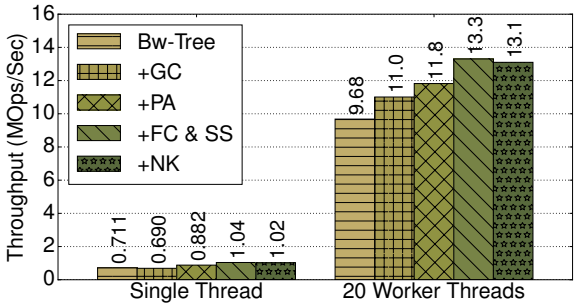
To better understand the benefits of each of these optimizations, we summarize the results from the previous experiments in Fig. 12. The results in Fig. 12a are the improvements from applying the optimizations to the OpenBw-Tree one-at-a-time. We use the Rand-Int keys for the Read/Update workload executing with both a single and 20 worker threads. When there is only a single thread, pre-allocation is the most beneficial optimization as it increases throughput by 28%. Fast consolidation and search shortcut is the second most effective optimization, increasing the speed by 18%. Enabling non-unique key support shows almost no impact if there is no duplicated key. Similar conclusions are also found for multi-threaded configuration.

Fig. 12b presents a comparison between the baseline Bw-Tree and OpenBw-Tree for all workload types, using Mono-Int keys. Numbers are measures using 20 worker threads. As shown in the diagram, the optimizations we apply to the Bw-Tree are equally effective for all workload types, speeding up operations by 27–35%.

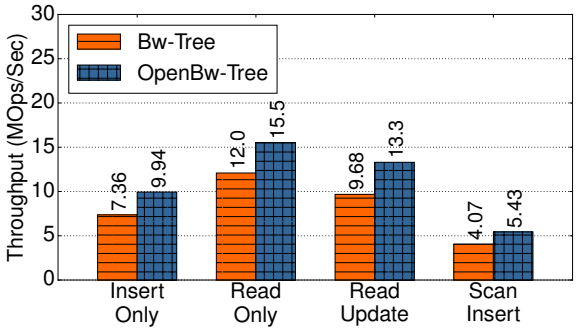
6 IN-MEMORY INDEX COMPARISON

The results from the previous section show that the Bw-Tree performs better with our optimizations from Section 4. The next step is to understand how it compares with other data structures. To the best of our knowledge, there are no previous results comparing the Bw-Tree with state-of-the-art in-memory indexes (the original paper [25] compared only against SkipList and BerkeleyDB). We therefore ported three additional indexes to our testing framework:

SkipList: SkipList [32] is an alternative to the B+Tree for in-memory databases. In our experimentation, we adopted the lock-free *No Hot Spot Non-blocking SkipList* (we simply use "SkipList"



(a) Rand-Int Keys – Read/Update



(b) Mono-Int Keys (20 Worker Threads)

Figure 12: Optimization Summary – The left diagram shows how performance goes up as we apply each optimization to the Bw-Tree. The right diagram shows a comparison between the baseline Bw-Tree and OpenBw-Tree for all four workloads. (GC: Garbage Collection; PA: Pre-allocation; FC: Fast Consolidation; SS: Search Shortcut; NK: Non-unique Key Support)

	Bw-Tree	OpenBw-Tree	SkipList	Masstree	B+Tree	ART
L1 Miss	5.0	3.2	14	2.2	4.3	0.78
L3 Miss	2.0	1.4	6.3	0.68	1.7	0.24
IPC	0.17	0.26	0.04	0.36	0.11	0.45
Branch	16	16	11	11	7.0	6.1
Br MisPred	1.0	1.2	0.74	0.75	0.95	0.14
Inst	77	73	50	51	43	30
Cycle	443	278	1150	179	380	72.9

Table 3: Microbenchmarks for Rand-Int Insert-only Workload – 20 worker threads. All worker threads are pinned to NUMA node 0. (L1/L3 Miss: L1/L3 data cache misses, 10^9; IPC: Instruction per cycle; Branch: Number of branch instructions, 10^9; Br MisPred: Number of branch mispredictions, 10^9; Inst: Number of instructions, 10^9; Cycle: Number of clock cycles, 10^9) All numbers are measured system-wide for all cores and all NUMA nodes.

below) that is optimized for multi-core CPUs [8]. This design uses a background thread to establish new pointers on each level for maintaining its expected performance. Worker threads do not need frequent synchronization because the background thread exclusively writes to upper levels of the linked-node structure.

Masstree: Masstree [31] is a hybrid trie/B+tree data structure that is used as the index structure of Silo [34] and its derivatives [16]. Its trie-based design allows Masstree to achieve high performance and low space consumption for keys with shared prefixes. It uses a lock-based synchronization protocol with epoch-based GC.

B+Tree: Although originally designed for disk-oriented DBMSs [6], B+Trees are widely used in main-memory database systems [35]. Instead of using traditional latching [3], our B+Tree implementation uses the *optimistic lock coupling* (OLC) [22] method. In OLC, each node has a lock, but instead of acquiring locks eagerly, read operations validate version counters (and restart if the counter changes). Read validations across multiple nodes can be interleaved, which allows implementing the traditional lock coupling technique for synchronizing tree-based indexes. Our B+Tree has a similar node organization as the OpenBw-Tree (sorted keys). We configure the B+Tree to use 4KB node size.

ART: The Adaptive Radix Tree (ART) [20] is designed for in-memory DBMSs and is the default index structure of HyPer [15]. Each node represents one byte of the key, which results in a max node fanout of 256. In contrast to most tries, which implement nodes as a fixed-size (256 for ART) array of child pointers, ART uses four different node layouts depending on the number of non-null child pointers. This saves space and improves cache efficiency. Our ART implementation also uses OLC [22] for synchronization.

Masstree and ART are tries and therefore require binary keys, unlike non-trie indexes that can handle any partially ordered keys. To support key types that are not necessarily ordered in their binary representation (e.g., integers on little-endian architectures), keys must be preprocessed to have a totally ordered binary form [20].

All of our experiments in this comparison use the same hardware as in the previous section. We begin with an evaluation that uses the same workload and key configurations from Section 5. We then present a worst-case scenario workload with high contention.

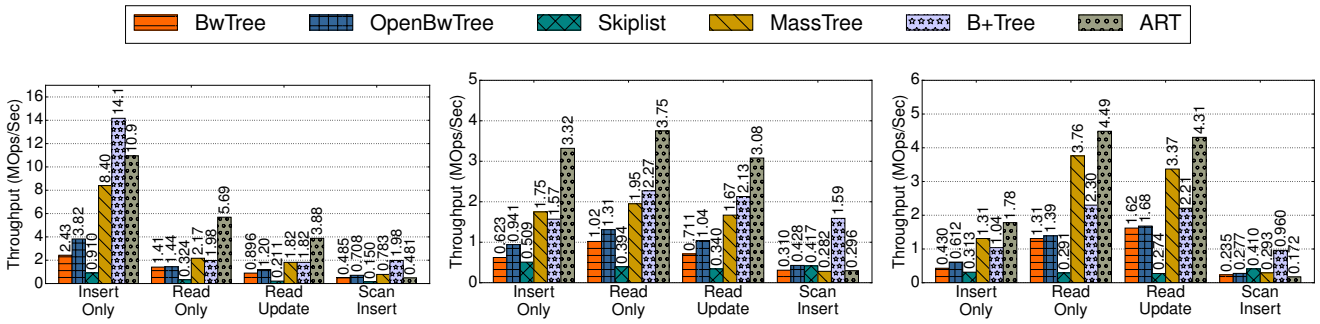
6.1 YCSB Workload

We compare the indexes using the YCSB-based workload in Section 5.1. We first run all four workloads with the three key configurations using a single worker thread. We then execute the trials again using 20 threads that are all pinned to a single CPU socket. The peak amount of memory consumed by the index during operations for the Read/Update workload are also measured. Finally, we measured the performance counters for the 20-thread Read/Update workload using *perf* and Intel's *Performance Counter Monitor*.

The results for the single-threaded and multi-threaded experiments are shown in Fig. 13 and Fig. 14 respectively. Memory numbers are in Fig. 15. Performance counter readings are in Table 3.

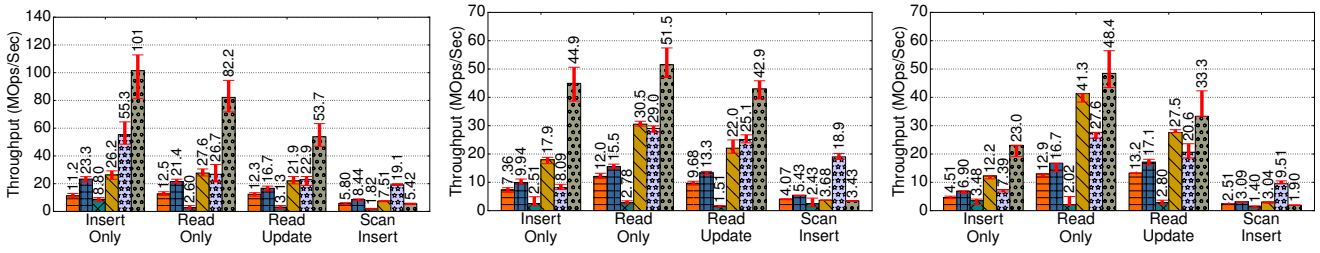
Although our optimizations made the OpenBw-Tree faster than the default Bw-Tree, it is still slower than its competitors except the SkipList. For example, the ART is more than 4× faster than the OpenBw-Tree for point lookups (though the ART is slower on the Scan/Insert workload). The OpenBw-Tree is also slower than the Masstree and the B+Tree, often by a factor of ~2×. Microbenchmark numbers show that the OpenBw-Tree in general has a higher instruction count and cache misses per operation (and hence lower IPC). Higher instruction count is a consequence of having complicated delta chain traversal routines. Higher cache misses are caused by features such as the Mapping Table.

The SkipList shows high variation and low performance for most multi-threaded experiments. This is because its threads do not create towers as they insert elements. Instead, the SkipList uses a



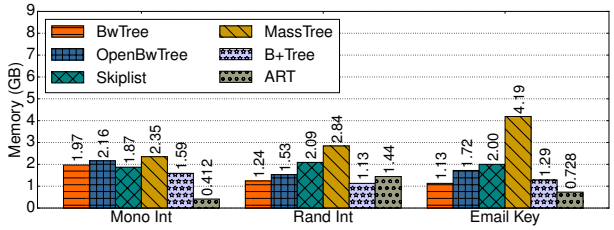
(a) Mono-Int Keys **(b)** Rand-Int Keys **(c)** Email Keys

Figure 13: In-Memory Index Comparison (Single-Threaded) – The worker thread is pinned to NUMA node 0.

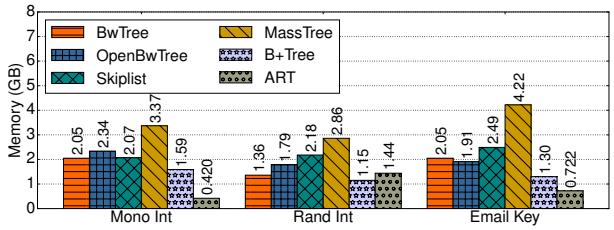


(a) Mono-Int Keys **(b)** Rand-Int Keys **(c)** Email Keys

Figure 14: In-Memory Index Comparison (Multi-Threaded) – 20 worker threads. All worker threads are pinned to NUMA node 0.



(a) Single-Threaded – Read/Update



(b) Multi-Threaded – Read/Update

Figure 15: Memory Usage – Single-threaded and multi-threaded

background thread that periodically scans the entire list and adjusts the height of towers. As a consequence, the background thread may not process recent inserts fast enough, and worker threads iterate through the SkipList's lowest level to locate a key, causing high cache misses and cycle counts.

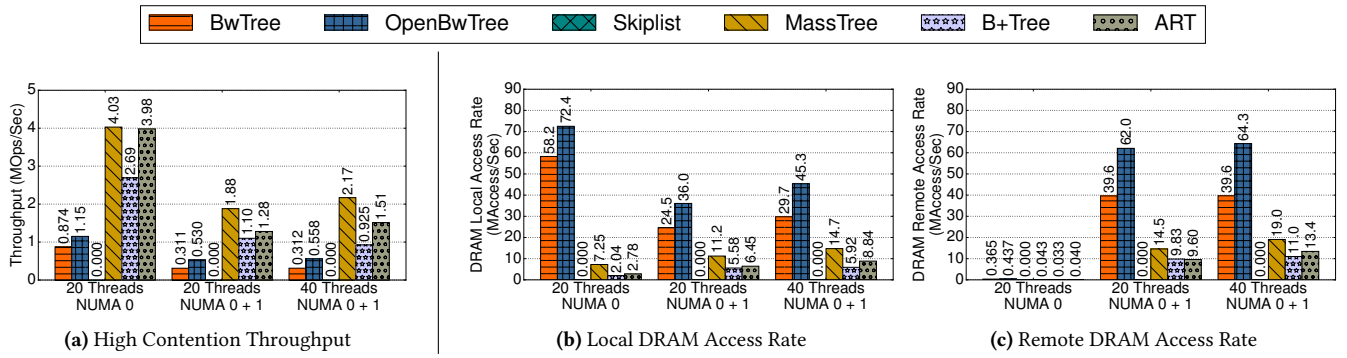
The Masstree has high single-threaded Mono-Int Insert-only throughput, but scales only by 3× using 20 threads. This is because Masstree avoids splitting an overflowed leaf node when items are inserted sequentially: it creates a new empty leaf node instead of copying half of the items from the previous leaf. This optimization, however, is less effective in the multi-threaded experiments where the threads' insert operations are interleaved. In general, the Masstree is comparable to the B+Tree for integer workloads (except Insert-only). And for Email, its performance is even comparable to the trie-based ART index, as its high-level structure is also a trie.

For integer keys, the B+Tree's Read-only and Read/Update performance is comparable to the Masstree, and much faster than the OpenBw-Tree. For the Mono-Int Insert-only workload, the B+Tree without any optimizations even outperforms the Masstree and ART, and is 3.7× faster than the OpenBw-Tree. The B+Tree also achieves high throughput for Scan/Insert workloads, and is usually 3–5× faster than all other indexes. But it has relatively poor performance for Email workloads. The microbenchmark indicates high cache misses and low IPC during Rand-Int and Email (not shown) insertion, which explains why the B+Tree is slower in these workloads.

ART outperforms the other indexes for all workloads and key types except Scan/Insert, where its iteration requires more memory access than the OpenBw-Tree.

As shown in Fig. 15a, both the Bw-Tree and the OpenBw-Tree use moderate amount of memory. The OpenBw-Tree consumes more memory than the Bw-Tree (10–31%) in all experiments due to pre-allocation and metadata. For the Mono-Int workload, as the pre-allocated space utilization is lower compared with the Rand-Int workload (Table 2). Correspondingly, the OpenBw-Tree uses more memory in the Rand-Int workload. For multi-threaded experiments, since worker threads keep garbage nodes in their thread-local chains, peak memory usage also increases slightly (8–17%).

Among all compared indexes, the ART has the lowest usage for Mono-Int and Email keys, while the B+Tree has the lowest



(a) High Contention Throughput **(b)** Local DRAM Access Rate **(c)** Remote DRAM Access Rate

Figure 16: High Contention Workload (Multi-Threaded) and DRAM Accesses Rate – Throughput using Mono-HC keys with the Insert-only workload on 20 threads. DRAM accesses are measured as operations per second for a high-contention workload under different NUMA configurations using 20 threads.

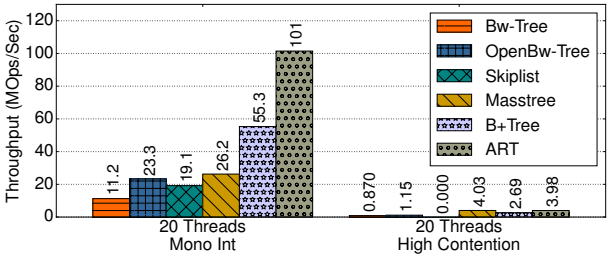


Figure 17: Comparison Between High Contention Workload and Normal Workload – 20 worker threads, with normal Insert-only and high contention Insert-only workload. All threads are pinned to NUMA node 0.

for the Rand-Int keys due to its compact internal structure and large node size (4 KB). The SkipList consumes more memory than the B+Tree/ART due to its customized memory allocator and pre-allocation; it has a memory usage comparable to the OpenBw-Tree. The Masstree always has highest memory usage, especially for the Email workload (2.0–5.7× higher). For integer workloads, although the Masstree still uses the most memory, the gap is smaller compared on the Email workload (only 1.3–2.5× higher, except for the ART).

The high throughput and low memory usage of the ART index under both single-threaded and multi-threaded environments should be attributed to its flexible way of structuring trie nodes of different sizes. Furthermore, only a single byte is compared on each level. Table 3 shows that both properties minimize CPU cycles and reduce cache misses, resulting in high IPC.

6.2 High Contention Workload

The salient aspect of the Bw-Tree's design is that it is lock-free, whereas most other data structures that we tested here use locks (although sparingly). Lock-free data structures are often favored in high contention environments because threads can make global progress [30], even though the progress may be small in practice. To better understand this issue, we created a specialized workload that with extreme contention. Each thread in the benchmark uses the `RDTSC` instruction with a unique thread ID suffix to generate monotonically increasing integers in real-time as keys, to mimic multiple threads appending new records to the end of a table. To further demonstrate how and in which way the NUMA configuration affects performance, we run the evaluation under three NUMA settings: 20 worker threads on a single NUMA node, 20 worker threads on two NUMA nodes, and 40 worker threads on two NUMA nodes. The last setting uses all available hardware threads on our testing system.

The results shown in Fig. 16a indicate that all five indexes degrade under high contention. Both Insert-only and Read/Update performance drops in both one- and two-node NUMA settings. The local and remote NUMA access rate, which is the number of DRAM accesses per second, is shown in Fig. 16b and Fig. 16c, respectively.

Under high contention, Masstree has the best result, followed by ART, and then B+Tree. OpenBw-Tree suffers from an extremely high abort rate as threads contend for the head of the Delta Chain. Table 2 shows that the abort rate is over 1000%, i.e., on average there are more than 10 aborts for every insert.

Overall, under high contention, none of these six data structures perform well. As shown in Fig. 17, compared with their multi-threaded performance numbers without high contention, all of them suffer from performance degradation. In particular, all lock-free indexes struggled more than any lock-based indexes; for example, the SkipList failed to make progress in this high-contention workload [2].

6.3 Bw-Tree Performance Decomposition

The Bw-Tree's lock-freedom comes at a cost: worker threads must traverse a Delta Chain on each level; the Mapping Table must be consulted to obtain the physical pointer of a virtual node; and instead of updating tree nodes in place, a delta records must be allocated and appended to the node being modified. To better understand these issues, we disabled the Bw-Tree's features one-by-one in a single-threaded environment. We ran our benchmark on each version of the modified Bw-Tree, and compare the result with the standard OpenBw-Tree and the B+Tree.

Disabling the Delta Chain: After populating the OpenBw-Tree using the Rand-Int Insert-only workload, we invoke a special routine that recursively visits every virtual node and consolidates its delta chain. After it returns, we run the Read-only workload under the same configuration as in Section 6. As Fig. 18 shows, by eliminating delta chains for the Read-only workload, the performance increases by 23%. If we backport this modification to the original Bw-Tree, the performance improvement will even be greater (45%),

[2]We reported a high contention insert problem to the authors of the SkipList implementation, but we unfortunately have received no response.

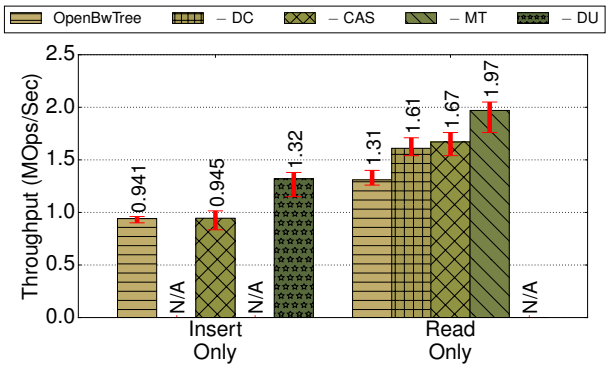


Figure 18: Performance Change as Features Are Decomposed – The leftmost number is from Fig. 13b. As we disable the OpenBw-Tree's features, performance also steps up. (DC: Delta Chain; CAS: compare-and-swap; MT: Mapping Table; DU: Delta Update) N/A indicates a crucial feature for inserts or an irrelevant feature for reads.

justifying our pre-allocation optimization in Section 4.1.

Disabling CaS: We instrumented the OpenBw-Tree source code with a "fake" atomic type that uses non-atomic primitives.

The third column of Fig. 18 shows that with CaS disabled, neither Insert-only nor Read-only operations become significantly faster. This seems to contradict the common belief that atomic operations like CaS usually takes more cycles on some ISAs. Our experiments, however, pins the worker thread on a single core, and therefore, the CPU can perform the CaS locally, requiring almost no cache coherence overhead [9]. While CaS would force memory barriers, the measured cost is dominated by main query processing.

Disabling the Mapping Table: We eliminated the translation of node IDs to physical pointers by invoking a post-insert process that replaces node IDs inside inner nodes with the corresponding physical pointers. Because inserts require Mapping Table to update multiple inbound pointers, we examine the Read-only workload only. The fourth column of Fig. 18 presents the effect of disabling the Mapping Table. Read performance increases by 18% due to fewer cache misses for load instructions (L1: 32% lower; L3: 52% lower).

Disabling Delta Updates: Both insert and delete operations append delta records to modify a virtual node. However, this is unnecessary when only a single thread modifies the tree at all times. To illustrate the trade-off between delta update and updating the node in place, we rewrote the leaf node update routine, and let the worker thread move elements in leaf nodes to insert instead of appending deltas. For simplicity, only leaf node modifications are done in place. We measured Insert-only numbers because the Read-only workload does not perform updates. The last column in Fig. 18 shows the trade-off of the two strategies. By replacing delta update with in-place update, the performance of Insert-only operations is 40% higher.

Overall, after disabling these lock-free features, the Bw-Tree is still 15%–19% slower than the B+Tree with OLC synchronization. We conjecture that the simplicity of OLC upper bounds the number of instructions for every operation, while for the Bw-Tree, even Read-only operations perform considerable bookkeeping to maintain the consistency of the tree, limiting its performance.

7 RELATED WORK

There have been many experimental studies on index performance. For instance, Gramoli examines the performance of concurrent data structure (list, queue, hash table, etc.) implementations [11] and Alvarez et al. compare radix trees with hash tables [1]. We note that prior work has limitations in either omitting concurrency or lacking experiments with important index designs such as the Bw-Tree [25] that are used in modern high-speed in-memory databases. In addition, previous studies often exclude realistic workloads such as string, Zipf-distributed, or monotonically increasing keys.

Traditional concurrent B-trees use lock coupling (also known as "hand-over-hand locking" or "crabbing") to provide fine-grained concurrent access by allowing a thread to hold no more than a certain number of locks [3]. This approach, however, scales poorly on modern multi-core CPUs because of the high overhead of frequent lock acquisition that occurs even under read-only workloads [5].

Optimistic locking is an approach to improve the scalability of B-trees on modern CPUs. OLFIT maintains a per-node version counter that is incremented on every node modification [5]. Readers traverse the tree by validating these version counters (and restart if necessary) instead of acquiring locks. Masstree is a hybrid trie/B-tree structure that uses separate version counters for node inserts and splits to reduce the chance of restarts [31]. Combining optimistic locks with lock coupling provides a general design for concurrent tree data structures [4, 22]. Optimistic variants of software transactional memory, on the other hand, is a more general design option for concurrent tree indexes [13, 26]. It achieves similar goals at the cost of more complicated design.

New hardware technologies create new opportunities and challenges for in-memory index designs. Master-tree is a B-tree index that can use large non-volatile memory [17]. It is built upon Masstree, but facilitates paging by maintaining only one inbound pointer per node. Hardware transactional memory (HTM) affords easy construction of concurrent data structures [21], but does not always outperform sophisticated non-HTM designs [22].

8 CONCLUSIONS

In this work, we introduced OpenBw-Tree, our clean slate Bw-Tree implementation. OpenBw-Tree incorporates a number of optimizations that were not described in the original Bw-Tree papers. Experimental results show that OpenBw-Tree outperforms the original Bw-Tree design. Nevertheless, even our optimized OpenBw-Tree, is still considerably slower than other state-of-the-art in-memory index structures like SkipList, Masstree and ART. OpenBw-Tree is also slower than a B+Tree that uses optimistic lock coupling, which indicates that lock-freedom does not always pay off in comparison with modern lock-based synchronization techniques.

The Bw-Tree has been proposed as an in-memory index and is used as such in Hekaton. Nevertheless, it is important to note that the Bw-Tree design supports eviction to SSD [23]. In the future, we will investigate larger-than-main-memory indexes and compare the log-structured nature of the Bw-Tree with the in-place approach of traditional B-trees.

REFERENCES

[1] Victor Alvarez, Stefan Richter, Xiao Chen, and Jens Dittrich. 2015. A comparison of adaptive radix trees and hash tables. In *ICDE*. 1227–1238.

[2] James H Anderson and Mark Moir. 1995. Universal constructions for multi-object operations. In *PODC*. 184–193.

[3] Rudolf Bayer and Mario Schkolnick. 1977. Concurrency of Operations on B-Trees. *Acta Informatica* 9 (1977), 1–21.

[4] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A Practical Concurrent Binary Search Tree. In *PPoPP*. 257–268.

[5] Sang Kyun Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. 2001. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *PVLDB*. 181–190.

[6] Douglas Comer. 1979. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)* 11, 2 (1979), 121–137.

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*.

[8] Tyler Crain, Vincent Gramoli, and Michel Raynal. 2013. No hot spot non-blocking skip list. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*. IEEE, 196–205.

[9] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2013. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask *(SOSP)*. 33–48.

[10] Keir Fraser. 2004. *Practical lock-freedom.* Ph.D. Dissertation. University of Cambridge.

[11] Vincent Gramoli. 2015. More Than You Ever Wanted to Know about Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms. In *PPoPP*.

[12] Timothy L Harris, Keir Fraser, and Ian A Pratt. 2002. A practical multi-word compare-and-swap operation. In *International Symposium on Distributed Computing*. 265–279.

[13] Nathaniel Herman, Jeevana Priya Inala, Yihe Huang, Lillian Tsai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2016. Type-aware transactions for faster concurrent code. In *Proceedings of the Eleventh European Conference on Computer Systems*. 31.

[14] Amos Israeli and Lihu Rappoport. 1994. Disjoint-access-parallel implementations of strong shared memory primitives. In *PODC*. 151–160.

[15] A. Kemper and T. Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*.

[16] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *SIGMOD*. 1675–1687.

[17] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *SIGMOD*.

[18] Thomas Kissinger, Benjamin Schlegel, Dirk Habich, and Wolfgang Lehner. 2012. KISS-Tree: Smart latch-free in-memory indexing on modern architectures. In *DaMoN*.

[19] Philip L. Lehman and s. Bing Yao. 1981. Efficient Locking for Concurrent Operations on B-trees. *ACM Trans. Database Syst.* 6, 4 (Dec. 1981), 650–670.

[20] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*. 38–49.

[21] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2016. Scaling HTM-supported database transactions to many cores. *IEEE Transactions on Knowledge and Data Engineering* 28, 2 (2016), 297–310.

[22] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *DaMoN*.

[23] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. LLAMA: A Cache/Storage Subsystem for Modern Hardware. *PVLDB* 6, 10 (2013), 877–888.

[24] Justin J. Levandoski, David B. Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. 2015. High Performance Transactions in Deuteronomy. In *CIDR*.

[25] Justin J Levandoski and Sudipta Sengupta. 2013. The BW-Tree: A Latch-Free B-Tree for Log-Structured Flash Storage. *IEEE Data Eng. Bull.* 36, 2 (2013), 56–62.

[26] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *SIGMOD*.

[27] David Lomet. 2001. The evolution of effective B-tree: page organization and techniques: a personal account. *SIGMOD Record* 30, 3 (2001), 64–69.

[28] David Lomet, Justin Levandoski, Sudipta Sengupta, Adrian Birka, and Cristian Diaconu. 2014. Indexing on Modern Hardware: Hekaton and Beyond. In *SIGMOD*.

[29] David B. Lomet, Sudipta Sengupta, and Justin J. Levandoski. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. In *ICDE*. 302–313.

[30] Darko Makreshanski, Justin Levandoski, and Ryan Stutsman. 2015. To lock, swap, or elide: On the interplay of hardware transactional memory and lock-free indexing. *PVLDB* 8, 11 (2015), 1298–1309.

[31] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *EuroSys*. 183–196.

[32] William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (1990), 668–676.

[33] Stephen Tu. 2013. Techniques for Implementing Concurrent Data Structures on Modern Multicore Machines. https://people.eecs.berkeley.edu/~stephentu/presentations/workshop.pdf. (2013).

[34] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *SOSP*. 18–32.

[35] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *SIGMOD*. 1567–1581.

A NODE SPLIT AND MERGE

This section details the process of node splitting and merging. As observed in Section 2.4, the Bw-Tree divides an SMO into two phases: the *logical* phase when a Δ**split**, Δ**merge** or Δ**remove** is appended to notify all worker threads of an ongoing SMO, and the *physical* phase that actually performs node split or merge by a successful CaS. In the following sections, we use *stage* to refer to an atomic step that changes the state of the Bw-Tree.

A.1 Node Split

Each logical node keeps track of the number of items it contains. When a worker thread discovers that the logical node's size is above a threshold, it initiates a node split. As shown in Fig. 19, the split operation occurs in three stages:

Stage I: To begin the split of node N_0, the thread creates a base node N_1 that is populated with the upper-half of N_0's items. The thread also copies some of N_0's attributes to N_1, such as its high-key and right-sibling, while the low-key is set to the split key (K_1). N_1's depth and size attributes are both set to zero. As shown in Fig. 19a, the thread then adds N_1 to the Mapping Table.

Stage II: The thread adds a Δ**split** record to the Delta Chain of N_0. As shown in Fig. 19b, this record changes N_0's high-key to K and right-sibling ID to N_1. At this point, the split process is incomplete because the split information has not been propagated to the parent node. This "half-split" state [29] is adopted from B^{link}-Tree. The Δ**split** serves as a flag to inform other worker threads of an unfinished split. We discuss the other delta records for node merges and removals in Appendix A.2. Threads check their search key against the high-key attribute of every logical node on their traversal path, even in the absence of a Δ**split** record. If a thread discovers a logical node whose high-key is less than or equal to the search key, it will move to its right-sibling node.

Stage III: Finally, in Fig. 19c the thread appends a Δ**separator** record to the parent logical node. This record not only stores (K_1, N_1), but also the separator item (K_2, N_2) that is the parent's next item in sorted key order. This way, when a thread sees the Δ**separator**, it can immediately compare the search key with K_1 and K_2. Thus, threads can avoid consulting the base node when the search key lies inside the interval $[K_1, K_2)$.

If multiple threads initiate the node split concurrently for the same logical node, only one of them will succeed in installing the Δ**split** record.

A.2 Node Merge

Merging two nodes together in the Bw-Tree follows roughly the same process as a split. One restriction is that a thread is only allowed to merge a node with its left sibling. We walk through the three steps for the merge process using the example shown in Fig. 20 where N_1 merges into N_0.

Stage I: The thread first appends a Δ**remove** record to N_1's Delta Chain. This record blocks access to the Delta Chain by forcing all threads to switch to N_1's left sibling N_0. In order for a thread to find N_0, it must keep a physical pointer to the parent node when traversing down to a child node of the parent.

Stage II: Any thread that reaches the left sibling N_0 without aborting appends a Δ**merge** record to N_0's Delta Chain. This record contains N_0's low-key, as well as N_1's high-key and right-sibling. A Δ**merge** differs from a Δ**split** because the former contains a physical pointer to N_1 instead of a logical node ID. Therefore, after a thread adds a Δ**merge** to N_0's chain, N_0 and N_1 are considered as part of the same logical node. The Δ**merge** contains a *merge key* (copied from N_1's low-key) that allows threads to choose between N_0 and N_1 during traversal.

Stage III: Any worker thread that encounters a Δ**merge** record will "help along" the merge process. As shown in Fig. 20c, a thread will attempt to install a Δ**separator** record in the parent node P. This record contains the node ID N_0 as well as two (key, node ID) pairs. The first is (K_0, N_0) and represents the separator item before the deleted item in P. The other pair (K_2, N_2) is the separator item that occurs after the deleted item. Any thread that observes a Δ**separator** will determine whether its search key lies in the interval $[K_0, K_2)$; if it does, that thread takes the fast path to N_0 and thus avoids searching the base node. After successfully appending a Δ**separator** record to P, the thread marks the Δ**remove** record from Stage I as deleted. Likewise, the thread also marks node ID N_1 as deleted, thereby completing the node merge process.

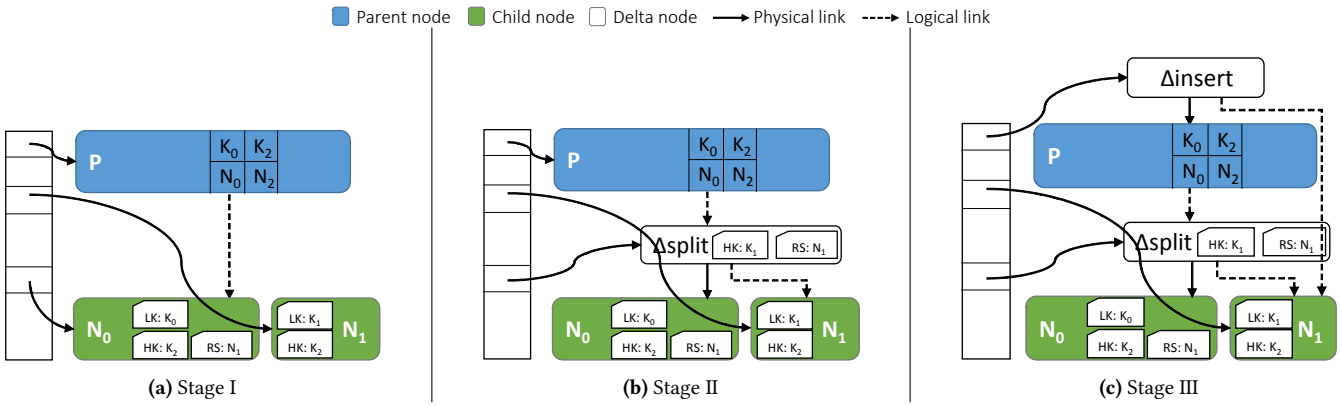
B CONCURRENT SPLIT AND MERGE

The SMO protocol in Section 2.4 guarantees that the index is in a valid state only if splits and merges do not interfere with each other. But a concurrent split operation on a parent node and a merge operation on its children will put it in an inconsistent state.

Fig. 21 illustrates this problem. Consider a parent inner node P, with two child nodes L and R, where R is L's right sibling. Assume that the tree uses high-keys as separators inside of its inner nodes. Let K_1 and K_2 be L and R's high-key, respectively. Suppose thread t_1 obtains the pointer to node P and traverses to P's child node R (Fig. 21a). Suppose another thread t_2 splits P into P_1 and P_2 (Fig. 21b), using K_2 as the split point; this process works if R does not merge into L. But if node R's size decreases below the threshold, then t_1 observes the underflow on node R and initiate the merge protocol on R by appending a Δ**remove** on R first (Fig. 21c) and then a Δ**merge** on L (Fig. 21d). t_1 now enters Stage III of the merge process and attempts to post a Δ**delete** to P. But since P has already split, the pointer is out-of-date and the CaS will fail.

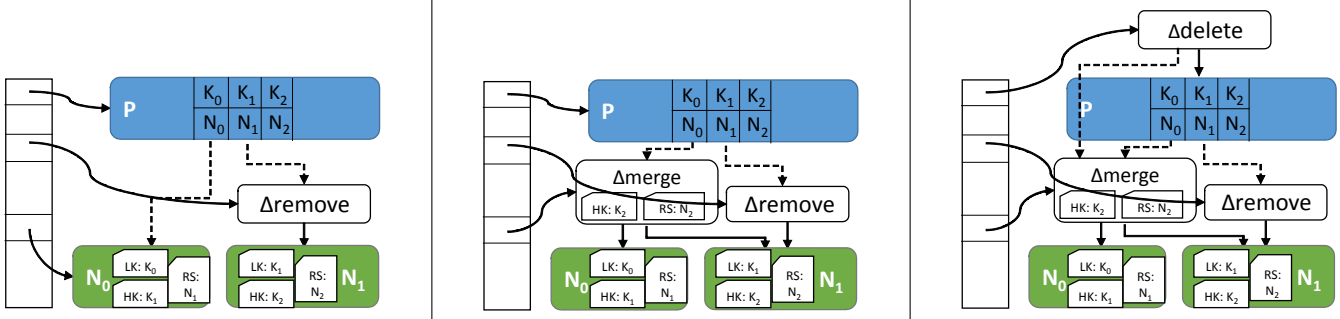
The tree is now inconsistent after the interleaved split and merge. Suppose a thread t_3 tries to complete the deletion after the above process. It visits logical node P_1 and then L, after t_2 appended the Δ**merge** record to L. According to the help-along protocol described in Section 2.4, t_3 searches P_1 for the separator item with key K_2 that is stored in the Δ**merge**. The search (correctly) does not find K_2 because K_2 is no longer within the range of P_1 after the split of node P. Therefore, t_3 assumes that the deletion process is finished even though the separator with key K_2 is left untouched in P_2. If the Delta Chain containing the Δ**merge** is consolidated afterwards, the link in the Δ**remove** becomes invalid.

Our solution in the OpenBw-Tree is to use a new kind of delta record called the Δ**abort** that will stop threads from appending to a Delta Chain. When a thread encounters one of these records during traversal, the Delta Chain is write-locked to that thread. Any thread



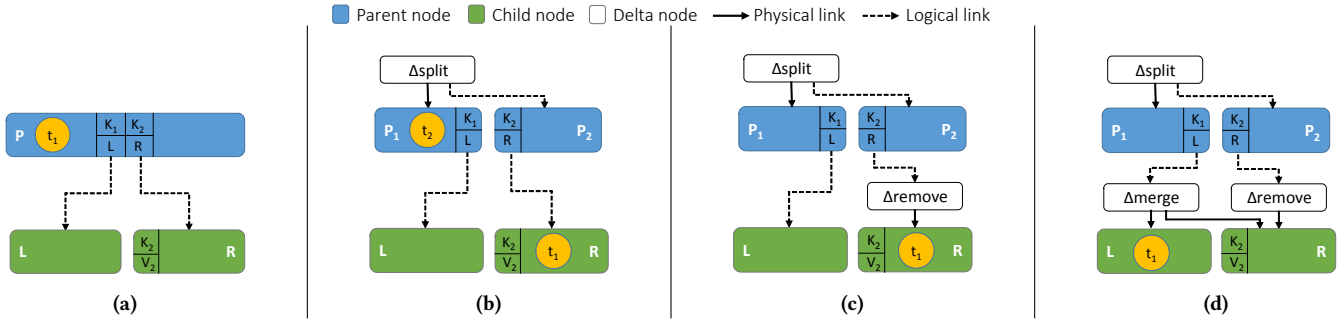
(a) Stage I **(b)** Stage II **(c)** Stage III

Figure 19: Node Split – This diagram illustrates a node split and how attributes are derived during the split. N_0 and N_1 are node IDs. For clarity, not all attributes are shown. (LK: low-key; HK: high-key; RS: right-sibling)



(a) Stage I **(b)** Stage II **(c)** Stage III

Figure 20: Node Merge – This diagram illustrates a node merge and how attributes are derived during the merge. N_0 and N_1 as node IDs. For clarity, not all attributes are shown. (LK: low-key; HK: high-key; RS: right-sibling)



(a) **(b)** **(c)** **(d)**

Figure 21: Concurrent Split and Merge – This diagram shows a concurrency bug caused by concurrent split and merge of parent node and child nodes respectively. After the merge in the last step, L and R are logically in the same node, while their parent node P splits into P_1 and P_2.

that attempts to append a new record to that Delta Chain will abort. If t_1 intends to append a remove delta on R (before Fig. 21c), it first attempts to append a Δ**abort** on P. If the CaS on the Mapping Table fails, t_1 will abort because P has been modified. If the CaS succeeds, then the node merge process from Appendix A.2 continues. After the Δ**merge** is appended, t_1 can remove the Δ**abort** from P's Delta Chain, unlocking P.

C FORWARD AND BACKWARD ITERATION

We give a brief overview of the implementation of forward and backward iterators. The general design idea of iterators has been covered by Section 3.2. Here we present our algorithm for iteration.

C.1 Forward Iteration

For a given search key K, a thread traverses the tree to find the smallest item in a leaf node with the key K', where $K' \geq K$. At the leaf level, the thread consolidates the Delta Chain into its leaf base node and copies the logical node (i.e., leaf base node plus all attributes) into IC as L. If there are no delta records, then the leaf level logical node is copied directly into IC as L. The thread then performs a binary search on L's data item array to find the item with key $K' \geq K$ and sets C in IC to the offset of this item.

When the thread moves the iterator forward, it increments C by one. C typically refers to a valid data item in L. If C equals the

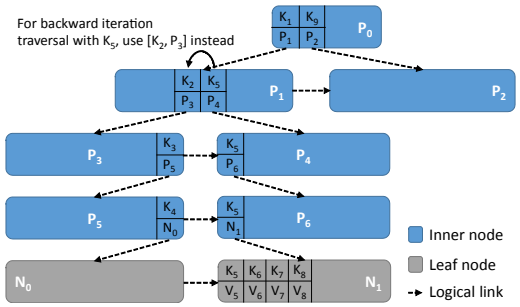


Figure 23: Backward Iteration – For backward iteration using K_5 as the low key, the path is $[(K_1, P_1), (K_2, P_3), (K_3, P_5), (K_4, N_0)]$. This is achieved by always going left when a separator item with key K_5 is seen during inner node search.

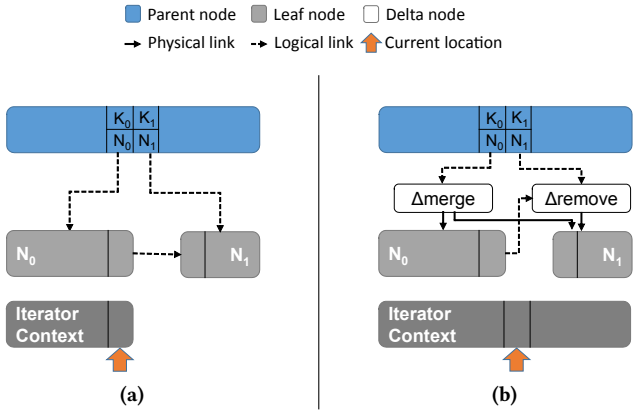


Figure 22: Forward Iteration with Concurrent Merge – In this example, the leaf node N_0 is merged into its left sibling (N_1) while the iterator scans forward. The arrow indicates the current location of the iterator.

size of L after the increment, then the thread initiates a new tree traversal using L's high-key (K_{high}) and builds a new IC.

Fig. 22 illustrates a case where a concurrent node merge happens while the iterator moves forward across leaf nodes. Fig. 22a shows the last state before the iterator advances from N_0 to N_1, and Fig. 22b shows the desired state of the iterator after the iterator advances. Based on our algorithm above, the thread traverses the tree using N_0's high-key K_1. It reaches the **Δremove** record and then the **Δmerge**, as described in Appendix A.2. After consolidating the Delta Chain, the new L contains the data items from both N_0 and N_1. A binary search on the new L using K_1 computes the value of C (i.e., the location of the first item with key K where $K \geq K_1$), thereby correctly locating the next data item.

C.2 Backward Iteration

To begin backward iteration starting at key K, a thread traverses the tree to find the starting leaf node with the item with key $K' \leq K$. It then instantiates IC using the same method described above. As long as the IC's counter $C > 0$, backward iteration will decrement C by one. But when C is zero, the thread cannot apply the same strategy used in the forward iteration to jump from one leaf node to the next. In the OpenBw-Tree, any logical node with low-key K_{low} and high-key K_{high} must be reachable by threads traversing the tree using key K, where $K_{low} \leq K < K_{high}$. Therefore, if K_{low} is the low-key of a logical leaf node, traversing to K_{low} will reach the same leaf node, rather than its left sibling.

There are two useful properties of a tree traversal that we can leverage to overcome this problem. For a search key K_{low}, let a path p be a sequence of (separator key, inner logical node) pairs that are generated during tree traversal. Recall that the OpenBw-Tree uses the low-key K_{low} of any leaf logical node N as the separator key, which allows finding N in its parent node. Thus, when there are no concurrent tree updates, the first property is that there always exists a path p from the root node to N such that K_{low} is the key of at least one separator item on p. Furthermore, if K_{low} is the separator item on p, then the second property is that all items below it (i.e., closer to the leaf level) must have K_{low} as its separator key.

A thread can find a node's left sibling based on the two above observations. When a thread on logical leaf L needs to find L's left sibling, it begins a backward tree traversal by using L's low-key K_{low} as the search key. At each inner node that the thread visits during the backward traversal, it checks whether K_{low} is equal to the separator key K' that are selected for traversing to the next level. If they are equal, then the thread uses the largest key K'', where $K'' < K'$, in the same inner logical node as the actual separator instead of K' for the traversal. At the leaf node, the thread uses the right-sibling ID to locate the node whose low-key is smaller than K_{low} and larger than the high-key of all other leaf nodes. Note that during the backward traversal, although the thread does not explicitly compute or store p, it maintains an invariant that every separator key K'' in p is smaller than K_{low}, by always choosing a smaller separator item in the same inner node.

Fig. 23 provides an example of this. Suppose that a thread starts a backward iteration using K_5 as the search key. When the thread traverses the tree the first time, it generates the search path p as $[(K_1, P_1), (K_5, P_4), (K_5, P_6), (K_5, N_1)]$, and reaches leaf node N_1. When the iterator moves to N_1's left sibling, however, the thread must start a backward traversal using the high-key of N_1, and this time it generates the path $[(K_1, P_1), (K_2, P_3), (K_3, P_5), (K_4, N_0)]$, reaching leaf node N_0. The reason for generating different paths is that when the thread picks K_5 on inner node P_1 during the second traversal, it will move left and use K_2 instead because the separator key K_5 is equal to the search key.

The leaf node whose low-key is K_{low} may be merged concurrently into its left sibling during the traversal. This can cause a thread to be unable to find a separator key $K' = K_{low}$ because K' was deleted from its parent node in Stage III of the node merge process. The thread will still reach a leaf node that contains a low-key that is smaller than K_{low}, guaranteeing correctness. More complicated concurrent modifications, however, can happen during the traversal. Should the thread reach a node whose high-key is greater than or equal to K_{low}, it must abort its traversal and restart.

D ACKNOWLEDGEMENTS

This work was supported (in part) by the Intel Science and Technology Center for Visual Cloud Systems, and the U.S. National Science Foundation (CCF-1438955, CCF-1535821). We thank the researchers at Microsoft Research for their constructive advice.

This paper is dedicated to the memory of Leon Wrinkles.
May his tortured soul rest in peace.