Harnessing Epoch-based Reclamation for Efficient Range Queries

Maya Arbel-Raviv
Technion - Israel Institute of Technology

Trevor Brown
Institute of Science and Technology Austria

Abstract

Concurrent sets with range query operations are highly desirable in applications such as in-memory databases. However, few set implementations offer range queries. Known techniques for augmenting data structures with range queries (or operations that can be used to build range queries) have numerous problems that limit their usefulness. For example, they impose high overhead or rely heavily on garbage collection. In this work, we show how to augment data structures with highly efficient range queries, without relying on garbage collection. We identify a property of epoch-based memory reclamation algorithms that makes them ideal for implementing range queries, and produce three algorithms, which use locks, transactional memory and lock-free techniques, respectively. Our algorithms are applicable to more data structures than previous work, and are shown to be highly efficient on a large scale Intel system.

1 Introduction

Concurrent *sets* are of great interest in applications such as *relational databases*, where they can be used as fast indexes. Realistic database workloads make heavy use of *range query* operations (RQs), which return all of the keys in some range [*low*, *high*] that are in the set. For example, consider the well known TPC-C benchmark, which simulates a large scale online transaction processing application for a wholesale supplier. In TPC-C, approximately 45% of all database transactions perform RQs over database indexes. Unfortunately, most concurrent set implementations do not support RQs, so they cannot be used for this purpose.

In this work, we develop a general technique for adding support for linearizable RQs to an existing concurrent set. The main challenge in implementing RQs is appropriately handling concurrent modifications to the data structure. If the data structure does *not* change throughout a RQ operation, then a RQ can be accomplished by a simple traversal over the relevant key range in the data structure. However, in the presence of concurrent updates, a traversal is unlikely to be linearizable.

Some of this work was done while Trevor Brown was a postdoctoral researcher at the Technion, and some was done while Maya Arbel-Raviv was visiting the University of Toronto and Trevor Brown was a PhD student there.

PPoPP '18, February 24–28, 2018, Vienna, Austria
2018. ACM ISBN 978-1-4503-4982-6/18/02…$15.00
https://doi.org/10.1145/3178487.3178489

We observe that it is conceptually straightforward to implement RQs with two assumptions: (1) given a pointer to a node, a process is able to determine whether the node was inserted (or deleted) before or after a given RQ, and (2) a process performing a RQ has pointers to all nodes deleted during the RQ. Given these assumptions, a RQ can be implemented as follows. First, traverse the data structure, and save each key whose node was inserted before the RQ and either not deleted, or deleted after the RQ. Then, for each node deleted during the RQ, save the key if the node was inserted before the RQ and deleted after the RQ. Finally, return the set of saved keys.

It turns out that assumption (2) is easily satisfied by any algorithm that reclaims memory using epoch-based reclamation (EBR). EBR can be used with any data structure, and is close in spirit to automatic garbage collection. It requires little effort on the part of a programmer, and can be extremely fast [9]. EBR is a popular choice in unmanaged languages such as C and C++, where one must manually perform memory reclamation.

The fascinating thing about EBR is that, in the process of ensuring that nodes are not freed until no process can access them, it essentially implements an efficient algorithm for saving all nodes deleted during a given operation. Each deleted node is placed in a *limbo list* before being freed, and it remains accessible until each process has started a new operation, since the node was deleted. So, for example, any nodes deleted during a RQ remain accessible until the entire RQ terminates. (This is also true for read-copy-update (RCU) [22], which has limbo lists and satisfies the same property.)

In this work, we present a novel technique for adding RQs to concurrent data structures that reclaim memory using EBR. A process performs a RQ by traversing the data structure and *collecting* the set of keys it contains. Then, it traverses the limbo lists to locate any keys that it may have missed due to concurrent deletion operations. In order to satisfy assumption (1), whenever a RQ encounters a node in the data structure, or in a limbo list, it needs a way to determine whether the node was inserted or deleted before or after the RQ. We solve this problem by adding a global *timestamp*, which is incremented by each RQ (using a fast fetch-and-increment instruction), and augmenting each node with an *insertion time* and a *deletion time* that contains the value of the timestamp at the exact moment the node was inserted or deleted.

This timestamping mechanism allows us to linearize RQs at the increment of the timestamp. Explicitly knowing when each RQ is linearized allows us to design a novel technique for experimentally validating the *correctness* of our RQ implementations. Furthermore, it led us to discover (and fix) some extremely subtle bugs in our code (e.g., with validation errors that appeared once every 1,000 executions).

We present several different variants of our technique: one that uses locks, one that uses hardware transactional memory (HTM), and one that offers wait-free RQs (for data structures with wait-free traversals). Our technique can be applied to many more data structures than previous approaches.

We demonstrate each variant of our technique by showing how RQs can be added to a large variety of data structures. This includes several advanced data structures (studied in our experiments) that *cannot* be used with the leading techniques for adding RQ support. We also perform experiments which demonstrate that our RQs are extremely fast for small ranges, and are also fast for extremely large ranges, matching the performance of techniques designed specifically to take a snapshot of an entire data structure.

Contributions

- We present three new algorithms for adding RQ support to existing concurrent sets, using locks, HTM and lock-free techniques, respectively.
- Our new algorithms can be used with many more data structures than previous techniques (see §2). We also address a hole in the theoretical underpinnings of two previous approaches [13, 24] (see §3).
- We demonstrate our algorithms by applying them to *six* different data structures. Experiments over a variety of workloads, including a realistic database application, show that our algorithms are extremely fast.

2 Related work

A small selection of data structures have been carefully designed with support for RQs [6, 26] or iteration [7, 8, 25]. Their respective techniques are closely tied to the underlying data structures, and it is difficult to see how they could be generalized. Instead, we are interested in *general techniques* for *adding* RQs to an existing concurrent data structure.

One easy way to implement RQs is to use transactional memory (TM). TM allows blocks of code to be executed as transactions that either commit and take effect atomically, or abort and have no effect on shared memory. TM can be implemented in software (STM) [14, 16] or hardware (HTM) (on recent Intel and IBM processors). STM currently has high overhead, and HTM has limitations that make transactions more likely to abort as they access more memory (making large RQs exceedingly likely to abort).

Read-log-update (RLU) [21] is a synchronization technique that combines locking, read-copy-update (RCU) [22] and techniques borrowed from STM. It yields relatively simple implementations, but it requires redesigning a data structure from scratch. Readers see a snapshot of the data structure, so RQs are easy. However, like RCU, updates suffer from high overhead, since *every* update must invoke an operation called *RLUSync*, which waits for all concurrent operations to finish. (The authors suggest accelerating updates by deferring *RLUSync* calls, but this makes *all operations non-linearizable.*)

One can also implement RQs by explicitly taking a *snapshot* of the data structure, and then collecting keys from the snapshot. Several techniques have been proposed for taking snapshots of a vector [1, 5, 20].

Petrank and Timnat generalized the approach in [20] to take a snapshot of a concurrent set [24], and implemented an *iteration* operation. They also gave correctness and progress proofs, and demonstrated their technique on a linked list and a skip-list.

At a high level, they introduced a Snap-collector object, which multiple threads can use to collaboratively build a snapshot of the data structure. The Snap-collector is essentially a collection of *reports*, each of which represents the insertion or deletion of a key during an iteration operation. Whenever there is an active iteration operation, updates that change the data structure must report their changes, and searches and other updates must also report changes they observe (to facilitate a complex linearization argument).

The Snap-collector has a few significant downsides.

1. It assumes the existence of a *data structure traversal* procedure, but the authors do *not* explain what such a procedure should guarantee. (We address this in §3.)
2. It can only be used with data structures in which deletions *mark* nodes as logically deleted before physically deleting them, and are linearized when the node is marked.
3. It is not clear how the transformation could be applied to data structures that offer *group updates*, which insert/delete many keys atomically, or to balanced trees and other, more complex data structures.
4. Since each iteration operation takes a snapshot of the entire data structure, small range queries cannot be implemented efficiently from iteration operations.
5. The Snap-collector creates many small auxiliary objects, including reports, wrappers for reports, and wrappers for nodes. This increases its space complexity and adds allocation and reclamation overhead.

Recently, the Snap-collector was extended to support more efficient RQs [13] by addressing the fourth downside, above. However, it still suffers from the others. Moreover, this extended algorithm causes all updates and searches to suffer additional overhead for *each* concurrent RQ.

3 Defining correct traversals

We consider data structures in which the keys of nodes are not changed directly. (Instead, to change a node's key, one replaces the node with a new copy.) As in [24], we assume the existence of a data structure traversal procedure. This traversal procedure serves as the skeleton of a range query algorithm. The traversal determines which nodes are *visited*, and in which order. When the traversal *visits* a node, it invokes a function *Visit* provided by the range query algorithm. We now address the omission in [24] by giving a property that this procedure must satisfy.

COLLECT: The traversal procedure must visit each node that contains a key in $[low, high]$ and is in the data structure *at all times* throughout the traversal. The traversal must *not* visit any node whose key was *never* in the data structure throughout the traversal.

We believe this is also the correct property for traversals in [24] (where $[low, high]$ is the universe). COLLECT shares some similarities to a complex property called *locality* that was recently proposed in an unpublished manuscript [2].

It is straightforward to argue that sorted lists (resp., skip-lists) satisfy the COLLECT property, since it follows almost immediately from the structural properties of the list (resp., bottom level in the skip-list). However, it is much more difficult to reason about trees. For instance, consider an internal (node-oriented) BST in which a key k at an internal node u is deleted by locating its successor k' at a leaf v, swapping the two nodes, and deleting v. In such a data structure, a traversal can visit u *before* it is swapped with v, then visit u again *after* they are swapped, missing v altogether, and violating COLLECT. One can imagine similar problems arising because of rotations in balanced trees.

In the rest of this section, we show how one can perform traversals that satisfy the COLLECT property for a large class of tree-based data structures.

3.1 Depth-first search (DFS) satisfies COLLECT for many trees

Consider a binary search tree that implements a set (possibly with group updates). Suppose search operations in this tree are exactly the same as in a sequential BST. Note that this describes many state-of-the-art concurrent search trees (e.g., [11, 15, 17, 23]). We show that a simple DFS-based traversal satisfies COLLECT for any such data structure, regardless of how its updates are implemented.

Figure 1 shows the search operation and traversal procedure for an internal BST. We give a short proof that this traversal procedure satisfies COLLECT. This procedure and proof are easily extended to handle external (leaf-oriented) BSTs, and trees in which nodes contain many keys.

The crucial observation is that each path followed during a traversal can be viewed as a search. Thus, if a traversal *misses* a node in the data structure, we can construct an execution that performs an incorrect search (yielding a contradiction).

```
1  Search(key)
2     node = root
3     while node ≠ ⊥
4        nodekey = node.key
5        if key = nodekey then return node.value
6        else if key < nodekey then node = node.left
7        else if key > nodekey then node = node.right
8     return ⊥

9  Traversal(low, high)
10    s = empty stack
11    s.push(root)
12    while not s.isEmpty()
13       node = s.pop()
14       if node ≠ ⊥ then
15          nodekey = node.key
16          if low ≤ nodekey ≤ high then Visit(node)
17          if low < nodekey then s.push(node.left)
18          if high > nodekey then s.push(node.right)
```

Figure 1. Internal BST search operation, and DFS-based traversal procedure.

Proof. To obtain a contradiction, suppose an invocation T of *Traversal*$(low, high)$ violates COLLECT. Two cases arise.

Case 1: some node u containing a key $k \in [low, high]$ is in the data structure at all times throughout T, but T does not visit u. Suppose we add, to this execution, an invocation S of *Search*(k) that begins at the same time as T. S begins by reading the root pointer (at line 2). We schedule this read so it happens at the same time as T reads the root pointer (at line 11). Thus, S and T both visit the same root node.

Next, S and T both read the key *nodekey* contained in that node (lines 4 and 15 respectively). Without loss of generality, suppose $k < nodekey$, so S will read *node.left* (at line 6). Since $low \le k$, we have $low < nodekey$, so T will also read *node.left* (at line 17). We schedule S's read of *node.left* so it happens at the same time as T's read. Thus, both the search and the traversal visit the same node.

By repeating this process inductively, we can schedule all of S's reads so they return the same values as reads performed in T. Consequently, S visits a subset of the nodes visited by T. Therefore, since T does not visit u, neither does S. Since we have assumed that the data structure is a search tree, no other node in the tree can contain k. So, S will return $\perp$, despite the fact that u is in the data structure at all times throughout the search. This contradicts our assumption that searches are correct.

Case 2: T visits a node u whose key k' was *never* in the data structure throughout T. Using the same approach as above, we can construct a *Search*(k') operation that visits u, and consequently returns u's value, contradicting our assumption that searches are correct. □

We stress that our technique for adding RQ operations will work with **any** traversal procedure that satisfies COLLECT. The data structure constraints described above are sufficient, **but not necessary**, for a traversal that satisfies COLLECT.

4 Algorithm

We now describe our technique for adding range query (RQ) operations to a data structure. We define a new *RQ Provider* abstract data type (ADT) and give three implementations of it. Then, we explain how it can be used to implement RQs. We make the following assumptions about the data structure:

1. There is a traversal algorithm that satisfies COLLECT.
2. Every update operation *that changes the set of keys* is linearized at a write or CAS, and the programmer knows where this write or CAS occurs in the code.

4.1 RQ Provider ADT

All processes operating on a data structure share a RQ Provider, and RQs use it to collect the sets of keys that they will return. A RQ Provider offers operations: *TraversalStart*(*low*, *high*), *TraversalEnd*(), *Visit*(*node*), *UpdateWrite*(*addr*, *newval*, *inodes*, *dnodes*) and *UpdateCAS*(*addr*, *oldval*, *newval*, *inodes*, *dnodes*).

A process invokes *TraversalStart*(*low*, *high*) at the start of a RQ. (In our implementations, the RQ is linearized at this invocation.) Then, the process traverses the data structure, and invokes *Visit*(*node*) for each *node* it visits. After the traversal, the process invokes *TraversalEnd*(), which returns all keys in [*low*, *high*] that were in the data structure when the RQ was linearized.

UpdateWrite and *UpdateCAS* are not used by RQ operations. Instead, they are used by update operations to relay information to range queries about when nodes are inserted and/or deleted. In each update operation *that changes the set of keys in the data structure*, a process must invoke *UpdateWrite*(*addr*, *newval*, *inodes*, *dnodes*) or *UpdateCAS*(*addr*, *oldval*, *newval*, *inodes*, *dnodes*) instead of the original write $*addr = newval$ or *CAS*(*addr*, *oldval*, *newval*) where the update would have been linearized. The argument *inodes* (resp., *dnodes*) should contain pointers to all nodes that will be inserted (resp., deleted) by the update.

4.2 Epoch-based reclamation ADT

We specified the RQ Provider ADT without mentioning EBR to allow implementations that *do not* rely on EBR. However, all of our implementations harness EBR to achieve high efficiency. So, we briefly describe the EBR ADT that we assume.

EBR provides four operations: *StartOp*, *EndOp*, *Retire* and *GetLimboLists*. The first three are standard in all implementations of EBR. *StartOp* and *EndOp* are invoked at the beginning and end of each data structure operation, and *Retire*(*node*) is invoked *after* a node is removed from the data structure. *Retire* and *GetLimboLists* can be invoked only between *StartOp* and *EndOp*. Intuitively, *GetLimboLists* returns all limbo lists containing nodes deleted during the current data structure operation. The programmer need not be aware of *GetLimboLists*, as it is only invoked by our RQ Provider implementations. We produced an efficient implementation of this ADT. Details appear in the full version of this paper.

4.3 Lock-based implementation

This implementation uses *unbounded timestamps*, and *locks* which can be acquired either by several processes at once in *shared mode* or by a single process in *exclusive mode*. There is a global timestamp TS, which is protected by a global lock L. Each *node* is augmented with fields $itime$ and $dtime$, which record the timestamp when the node was inserted and deleted, respectively. Initially, each of these fields contains a special value $\perp$, which indicates that the timestamp has not been set. For simplicity, we assume that each *node* contains one key, which we denote by $node.key$. (However, it is straightforward to support multiple keys per node, and we have implemented support for this in our code.)

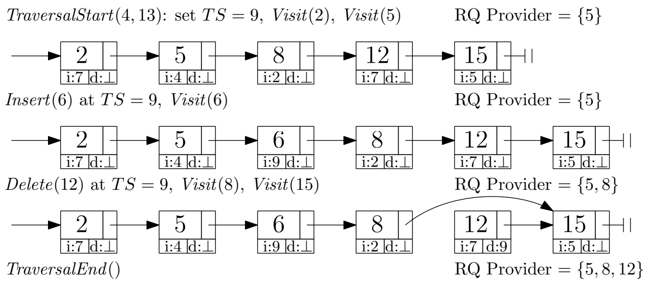

Figure 2. Lock-based RQ Provider: example execution.

High-level description Each update acquires L in *shared mode* (allowing concurrency) and reads TS atomically with the write or CAS where the update is linearized. Then, after this write or CAS, it sets the $itime$ (resp., $dtime$) fields of the nodes it inserted (resp., deleted).

Each RQ acquires L in *exclusive mode* to increment TS from t to t', and is linearized at this increment. We think of the RQ as occurring *between* t and t', so that updates at time t occur before the RQ, and updates at time t' occur after the RQ. The RQ traverses the data structure, invoking *Visit*(*node*) for each *node* it encounters. *Visit*(*node*) uses the *node*'s $itime$ and $dtime$ to determine whether it was in the data structure when the RQ was linearized. If so, *Visit saves* the node in the RQ Provider so it can be returned by *TraversalEnd*. Finally, it traverses the limbo lists maintained by EBR to locate any keys in the range that were deleted after the RQ and were missed during the traversal.

Example execution Figure 2 shows an example execution with a RQ over [4, 13], which is linearized *before* an *Insert*(6) and *Delete*(12). The RQ invokes *Visit*(2) and *Visit*(5), saving 5 in the RQ Provider since its node was inserted at time 4, which is *before* the RQ. It then invokes *Visit*(6), but 6 is not saved since its node was inserted at time 9, after the RQ. After the *Delete*(12), the RQ invokes *Visit*(8) and *Visit*(15), saving 8 since its node was inserted at time 2, before the RQ. Finally, *TraversalEnd* visits all nodes deleted during the RQ, and sees that 12 was deleted at time 9, after the RQ. So, it returns 12, along with the previously saved keys.

Detailed description Pseudocode for our implementation appears in Figure 3. We first assume that the data structure physically deletes nodes at the same time as it logically deletes their keys. Then, we will show how to handle data structures in which nodes are first logically deleted, and later physically deleted (possibly by a different thread).

Recall that a process invokes *Retire*(*node*) to add *node* to a limbo list only *after node* is removed from the data structure. Thus, one must take care to avoid the following situation. Suppose a process p performs an *UpdateCAS* or *UpdateWrite* that deletes a node during a RQ by another process q, and q's RQ misses this node during its traversal. Before p can add the node to a limbo list (using *Retire*), q traverses the limbo lists, and fails to find the node. In this case, q's RQ will fail to return the key stored in this node, even though it was in the data structure when the RQ was linearized.

To avoid this situation, *UpdateCAS* begins by announcing any nodes that would be deleted by the update in a single-writer multi-reader array, *before* deleting these nodes. By having *UpdateCAS* announce the nodes it will attempt to delete, we ensure that any node deleted during a RQ can always be found, either in the data structure, or in process' announcements, or in the limbo lists.

After announcing, *UpdateCAS* acquires L in shared mode, reads TS, performs the CAS where the update is linearized (the *update CAS*), and releases L. If the CAS succeeds, then *UpdateCAS* sets the $itime$ ($dtime$) fields of the nodes it inserted (deleted). Any deleted nodes are added to the current limbo list using *Retire* (performing memory reclamation for the underlying data structure). Then, *UpdateCAS* deletes its announcements.

One might wonder why nodes' $itime$ fields are not set *before* the update CAS. This approach would also work, as long as the nodes being inserted can only be accessed by the process that successfully inserts them. However, in some data structures (especially lock-free ones), processes can *help* one another perform operations, so several processes can all attempt to insert the same node(s), and end up writing conflicting values to their $itime$ fields. By having a process write to these nodes' $itime$ fields only after performing the update CAS successfully, we ensure that only one process writes to the $itime$ field of each node. (We assume that only one process will perform a successful update CAS or update write for each operation.)

UpdateWrite is very similar to *UpdateCAS*, so it is omitted. (Note that we do **not** change the linearization points of updates. We simply add extra steps surrounding them.)

TraversalStart(low, $high$) simply acquires L in exclusive mode, then increments TS and immediately releases L. It is linearized at the increment of TS.

Visit($node$) invokes a helper function *TryAdd*($node$, $\bot$, *FromDataStructure*). *TryAdd* first waits until $node.itime$ has been set. Then, it uses the $itime$ to determine whether the node was inserted before or after the RQ. If it was inserted after the RQ, then *TryAdd* terminates *without* saving $node.key$ in the RQ Provider. Finally, if $node.key$ is in [low, $high$], *TryAdd* saves $node.key$ in the RQ Provider.

```
Global variables: $TS$, $L$
Shared variables for process $p$: $announce_p[1\ldots]$
Private variables for process $p$: $time_p$, $low_p$, $high_p$, $result_p$

UpdateCAS(addr, oldval, newval, inodes, dnodes)
    Announce all of the nodes in dnodes in $announce_p$
    $L.acquireShared()$                          // Read TS atomically with CAS
        $ts = TS$
        $res = CAS(addr, oldval, newval)$
    $L.release()$
    if $res = oldval$ then
        for each $node \in inodes$ do $node.itime = ts$
        for each $node \in dnodes$
            $node.dtime = ts$
            Retire(node)             // Add node to the appropriate limbo list
    Remove all announcements in $announce_p$
    return $res$

TraversalStart($low$, $high$)
    $result_p = \emptyset$ ; $low_p = low$ ; $high_p = high$
    $L.acquireExclusive()$                                         // Update TS
        $TS = TS + 1$
        $time_p = TS$
    $L.release()$

Visit(node)
    return TryAdd(node, $\bot$, FromDataStructure)

TraversalEnd()
    Collect pointers $p_1, \ldots, p_k$ to other processes' announcements
    for each $annptr \in p_1, \ldots, p_k$        // note: $*annptr$ is a node pointer
        TryAdd($*annptr$, annptr, FromAnnouncement)
    $l_1, \ldots, l_m$ = GetLimboLists()          // Collect pointers to all limbo lists
    for each $list \in l_1, \ldots, l_m$                        // Traverse limbo lists
        for each $node \in list$ do TryAdd(node, $\bot$, FromLimboList)
    return $result_p$

Private functions
TryAdd(node, annptr, source)
    while $node.itime = \bot$ do wait
    if $node.itime \geq time_p$ then return           // node inserted after RQ
    // Check if/when node was deleted
    if $source = FromDataStructure$ then
        do nothing             // node was not deleted when RQ was linearized
    else if $source = FromLimboList$ then
        while $node.dtime = \bot$ do wait
        if $node.dtime < time_p$ then return // node deleted before RQ
    else if $source = FromAnnouncement$ then
        $dtime = \bot$
        while $dtime = \bot$ and $*annptr = node$ do $dtime = node.dtime$
        if $dtime = \bot$ then
            // loop exited because the process removed this announcement
            // if the process deleted node, then it has now set node.dtime
            $dtime = node.dtime$                                  // reread dtime
            if $dtime = \bot$ then return   // the process did not delete node,
                                            // but another process might have
        if $dtime < time_p$ then return            // node deleted before RQ
    if $node.key \in [low, high]$ then
        add ($node.key$, $node.value$) to $result_p$
```

Figure 3. Lock-based RQ Provider

TraversalEnd starts by collecting pointers to all of the announcements made by other processes. (Each announcement is a pointer to a node, and *TraversalEnd* collects *pointers* to these announcements, so it can re-read them later, to see if they still point to the same nodes.) Note that it must collect the announcements before searching the limbo lists for missing nodes, since processes first announce nodes they are planning to delete, and only later add them to the limbo lists, *after* successfully deleting them. Thus, by searching first in the data structure, then in the announcements, then in the limbo lists, we guarantee that we will not miss any nodes deleted during the RQ.

For each announcement pointer *annptr* that *TraversalEnd* collects, it invokes *TryAdd*($*annptr$, *annptr*, *FromAnnouncement*). Each invocation of *TryAdd* waits until $itime$ is set, and terminates if the node was inserted after the RQ. Then, it additionally waits at line 50 until either $dtime$ is set, or the node is no longer announced (by process that originally announced it). Note that we cannot simply wait until $dtime$ is set, because the node was found in the announcements of some process, which only indicates that the process would *like* to delete the node. It does not guarantee that the node will be deleted. Thus, $dtime$ might never be set.

After waiting in the loop at line 50, if $dtime$ is still not set, then the loop exited because the node is no longer announced by p. If p had deleted the node, then it would have set its $dtime$ *before* removing its announcement. Thus, *TryAdd* rereads $node.dtime$, after seeing that the announcement was removed, to determine whether p deleted the node.

If $node.dtime$ is still not set, then p has not deleted the node. So, either (1) a different process deleted the node, or (2) it has not been deleted. In case (1), *TraversalEnd* will find the node in another process' announcements, or in a limbo list. In case (2), the node will have been found during the data structure traversal, and saved in the RQ Provider by a previous call to *TryAdd*. In all other cases, $node.dtime$ is set. If it shows that the node was deleted before the RQ, then *TryAdd* terminates *without* saving $node.key$ in the RQ Provider. Otherwise, *TryAdd* checks if $node.key \in [low, high]$, and, if so, saves it in the RQ Provider.

Next, *TraversalEnd* traverses the limbo lists, and invokes *TryAdd*(*node*, $\perp$, *FromLimboList*) for each node. These invocations of *TryAdd* are similar to the invocations for announced nodes, except that we can simply wait until $dtime$ is set, since we know that each node has already been deleted. Finally, *TraversalEnd* returns the saved nodes at line 36.

Correctness Observe that the linearizability argument for the original data structure operations goes through unchanged, since we do not change the linearization point of any operation. We argue that RQs are linearizable.

First, we argue that nodes' timestamps are accurate. Each update that changes the data structure holds L in shared mode when it reads $ts = TS$ (at line 8) and performs the write or CAS where it is linearized. Therefore, the timestamp ts, which is written to any nodes inserted or deleted by the update, is the exact value of the timestamp when the update was linearized.

Next, we argue that, for each key $k \in [low, high]$ in the data structure when the RQ was linearized, there is an invocation I of *TryAdd* on some node u that contains k and was in the tree when the RQ was linearized. Since we have assumed a traversal procedure that satisfies COLLECT, if u was in the data structure throughout the entire traversal, then we will invoke *TryAdd* on it. If u was in the data structure when the RQ was linearized, but was subsequently deleted (and the traversal did not visit it), then *TraversalEnd* will find it either in the announcements of the process that deleted it, or in a limbo list, and invoke *TryAdd* on it.

Finally, we argue that I saves k in the RQ Provider (for each I and k). Suppose, to obtain a contradiction, that I does not add $u.key = k$ in the RQ Provider. Three cases arise, the first two of which are straightforward. Case 1: I saw at line 41 that u was inserted after the RQ, or saw at line 47 or line 57 that u was deleted before the RQ. This contradicts our assumption that u was in the data structure when the RQ was linearized. Case 2: I saw at line 58 that $u.key \notin [low, high]$. This contradicts our assumption that $u.key = k \in [low, high]$.

Case 3: I saw $dtime = \perp$ at line 55. In this case, some process p announced u and later removed its announcement without setting $u.dtime$. This implies that p did not delete u. Note that I must have been invoked by *TraversalEnd*, after the data structure traversal. If no other process deleted u, then the traversal must have already invoked *Visit*(u), which invoked *TryAdd*, which saved $u.key$ in the RQ Provider (a contradiction). On the other hand, if another process deleted u, then u will be found in another process' announcements, or in a limbo list, and it will be passed to another invocation of *TryAdd*. Thus, each case leads to a contradiction. □

Optimizations Recall that *TryAdd* does *not* save nodes with $dtime < time_p$ in the RQ Provider. Suppose each process has its own limbo lists, and they are implemented as linked lists, with insertion at the head, as in the distributed EBR algorithm DEBRA [9]. Then the nodes in limbo lists will be sorted in descending order by $dtime$. Consequently, while *TraversalEnd* is traversing the limbo lists, before it invokes *TryAdd*, it can check if $node.dtime \neq \perp$ and $node.dtime < time_p$. If so, it can immediately stop traversing that limbo list, since all other nodes in the list were deleted before the RQ.

Additionally, we can modify *TraversalEnd* so that it attempts to skip any nodes deleted after the data structure traversal has finished (meaning they were either inserted after the RQ, or were definitely visited during the data structure traversal). We do this by reading TS at the start of *TraversalEnd*, storing it in a variable $endTS$, and skipping any nodes in the limbo lists with $dtime > endTS$.

Supporting logical deletion We now show how our lock-based RQ Provider can be modified to support data structures in which nodes are *marked* as logically deleted, and are later physically removed.

In such data structures, a deletion operation is typically linearized at the step where it marks a node (the *marking step*). Consequently, *UpdateWrite* or *UpdateCAS* should be used to perform the marking step, not the physical deletion. And, the *dnodes* argument to these operations should contain pointers to all nodes to be *logically deleted*.

We briefly describe the changes to the pseudocode in Figure 3. *UpdateCAS* should no longer announce nodes in *dnodes*, or invoke *Retire* to add nodes in *dnodes* to the current limbo list, because these nodes are now only *logically* deleted by the CAS. Instead, each update that *physically* deletes node(s) must perform the announcements before it physically deletes any node, invoke *Retire* after deleting them, and then remove the announcements.

The remaining changes appear in *TryAdd*. At line 44, we previously knew that the node was not deleted when the RQ was linearized, simply because we found the node while traversing the data structure. With logical deletion, this is not necessarily true, so we check whether the node has been logically deleted. (We assume the data structure provides a predicate called *IsLogicallyDeleted*(*node*) for this purpose.) If the node is not logically deleted, then it was not deleted when the RQ was linearized. Otherwise, we wait until $node.dtime$ is set, and use it to determine whether the node was deleted before or after the RQ (similar to line 46 and line 47).

Note that, in certain algorithms with logical deletion, one process can physically delete a node that was logically deleted by another process. In such an algorithm, the nodes in limbo lists will not necessarily be ordered by $dtime$, so the first optimization described above might not be applicable.

4.4 HTM-based implementation

Current HTM implementations are very fast, but they offer no progress guarantees, and transactions can abort for any reason. Thus, it is necessary to pair code that uses HTM with a *fallback* code path to be executed if a hardware transaction aborts. (Typically, transactions are attempted several times in hardware before the fallback code is executed.)

We show how HTM can be used to substantially improve the performance of our lock-based RQ Provider. The most significant bottleneck in our implementation is the acquisition of locks (in shared mode) in *UpdateCAS*. We use HTM to mostly avoid the acquisition of locks in *UpdateCAS*.

More specifically, we replace lines 7 to 10 of Figure 3 with an improved transactional version of that code, and execute the original code only if a transaction fails to commit after a fixed number of attempts (30 in our experiments). Each transaction first checks if L has been acquired in exclusive mode, and aborts if it has. Then, the transaction performs lines 8 and 9 and commits. If L is acquired in exclusive mode at any time during the transaction, the transaction will abort.

Note that the transaction only needs to *read* L. Consequently, it scales much better than the original algorithm in update-heavy workloads (as we will see in the experiments).

4.5 Lock-free implementation

There are two blocking components of the original lock-based RQ Provider: the lock L, and the loops in *TryAdd* that wait until a node's $itime$ and $dtime$ fields are set. In this section, we show how these blocking components can be modified to produce a lock-free RQ Provider.

One easy way to produce a lock-free RQ Provider is to use a powerful lock-free synchronization primitive such as k-CAS [18]. A k-CAS operation atomically performs CAS on k distinct memory addresses. So, for example, we could replace the lock-based code between line 7 and line 14 of Figure 3 with code that reads TS, then uses k-CAS to atomically: perform the original update CAS, set all $itime$ and $dtime$ fields in nodes being inserted/deleted, and verify that TS has not changed since it was last read. This would eliminate the use of locks, as well as the waiting in *TryAdd* (since $itime$ and $dtime$ would be set atomically, with the update). However, k-CAS is relatively expensive, so this approach would be slow in practice.

Instead, we opt for a more lightweight approach. The idea is to replace the critical sections in *UpdateCAS* and *TraversalStart* with lock-free code that accomplishes the same thing, and then to handle missing $itime$ and $dtime$ values in a more intelligent way. (We omit a discussion of *UpdateWrite*, but it is similar.)

We use the lock-free double-compare-single-swap (DCSS) primitive of Harris et al. [18]. DCSS takes 5 arguments: two addresses, two expected values, and one new value. It atomically reads the two memory addresses, checks if they contain the two expected values, and if so, writes the new value into the second address. DCSS allows us to perform the update CAS *only if* TS contains a specific timestamp. Thus, we can replace the critical section in *UpdateCAS* with a loop that repeatedly reads TS and performs DCSS until the DCSS either succeeds, or fails because *the original update CAS* performed by the underlying algorithm would also have failed (not simply because TS has changed). The critical section in *TraversalStart* can simply be replaced with a fetch-and-add instruction on TS.

It remains to eliminate any waiting for $itime$ and $dtime$ values to be set. The lock-free DCSS implementation guarantees progress by using *helping*. Before performing a DCSS, a process first creates a *descriptor* object, which contains all of the arguments to the DCSS operation. If a process is prevented from making progress by some DCSS operation O, then the process will use the information in the *descriptor* for O to perform O, on behalf of the process that started O.

We make a straightforward modification: we add a *payload* to each descriptor when it is created. The payload contains pointers to all nodes that the DCSS would insert/delete if it succeeds. We also have each process *announce* its descriptor at the beginning of each DCSS, so that all processes can see what it is doing.

In *TryAdd*, whenever a process encounters a node with an *itime* or *dtime* field that is not set, it inspects the announced descriptors (and their payloads) to find the process that is currently trying to insert/delete the node. It then uses the information in the descriptor to *help* the DCSS complete (using the basic DCSS helping mechanism). In the process of helping the DCSS complete, it learns whether the DCSS was successful. When it finds the DCSS that successfully inserted/deleted the node, it learns the correct time *itime* or *dtime* value from the descriptor. (Recall that the arguments to the DCSS are stored in the descriptor, and one of these arguments is the value of *TS* when the DCSS took place.)

We briefly explain the progress guarantees for the RQ Provider's operations. Note that *UpdateCAS* is not wait-free, because it invokes DCSS, which is only lock-free. However, we argue that the other RQ Provider operations are all wait-free. *TraversalStart* is straightline code, so it is wait-free. *TryAdd* does not wait for *itime*/*dtime* values to be set, but it *helps* up to n DCSS operations, where n is the number of processes. The helping procedure in Harris' DCSS implementation has no loops, jumps or recursion, and is wait-free. Thus, *TryAdd* is wait-free (and, hence, so is *Visit*).

Since *TraversalEnd* traverses the limbo lists, its progress guarantee depends on the implementation of EBR. Suppose deleted nodes are inserted at the head of each limbo list. Then, when *TraversalEnd* collects pointers to the (head nodes of the) limbo lists of other processes at line 33, it fixes a finite number of nodes that it will traverse at the next line. Therefore, under this assumption (which is satisfied by DEBRA [9]), *TraversalEnd* is wait-free. (Note that an RQ built from *TraversalStart*, *Visit* and *TraversalEnd* will be wait-free only if its traversal procedure is also wait-free.)

5 Experiments

Our experiments were run on a 2-socket Intel E7-4830 v3 with 12 cores per socket and 2 hyperthreads (HTs) per core, for a total of 48 threads, and 128GB of RAM. In all of our experiments, we pinned threads to alternating sockets, and equally distributed threads between cores on each socket. (So, hyperthreading is only used with more than 24 threads.)

The machine runs Ubuntu 14.04 LTS. All code was compiled with the GNU C++ compiler (G++) 6.3.0 with compilation options `-std=c++11 -mcx16 -O3`. We used the scalable allocator jemalloc 5.0.1, which greatly improved performance.

We applied our lock-based RQ algorithm (*Lock*), our HTM-based algorithm (*HTM*), and our lock-free algorithm (*Lock-free*) to a variety of state-of-the-art data structures. Figure 4 gives an overview. This is in stark contrast to previous approaches, which have only been demonstrated to work only with a very limited set of data structures.

Name	Type	Sync.	Log. del.
LFList	Linked-list [19]	Lock-free	Yes
LazyList	Linked-list [19]	Locks	Yes
SkipList	Skip-list [19]	Locks	Yes
LFBST	External BST [11]	Lock-free	No
Citrus	Internal BST [3]	Locks & RCU	No
ABTree	Ext. (a, b)-tree [10]	Lock-free	No

Figure 4. Data structures studied in our experiments.

LazyList and SkipList use optimistic concurrency control techniques to avoid locking during searches. Citrus uses RCU to avoid locking during searches, and to enable fast concurrent updates. ABTree is a concurrency-friendly generalization of a B-tree. It is a balanced tree in which nodes contain between a and b keys.

We compare our RQ algorithms with the Snap-collector, RLU, and a *non-linearizable* RQ algorithm (*Unsafe*) that simply traverses the data structure and returns the keys that it sees (in the desired range). Unsafe serves as an upper bound on the performance of any RQ algorithm. Note that the Snap-collector *cannot* be used with LFBST, Citrus or ABTree, since they do not perform logical deletion (separately from physical deletion). We implemented RQs using the Snap-collector for LFList, LazyList and SkipList. Since RLU requires completely redesigning the data structure, we only applied it to two data structures: Citrus and LazyList. The reason we chose Citrus is that it synchronizes using locks and RCU, which is quite similar to RLU. Note that we did not use deferred synchronization in RLU, since it is *not linearizable*.

All data structures reclaimed memory using DEBRA [9]. Our experiments were implemented in C++, which does not have automatic garbage collection. Thus, we also used DEBRA to reclaim objects created by the Snap-collector. DEBRA has extremely low overhead, so the Snap-collector is not penalized for allocating many objects. Our experiments are, thus, quite charitable towards the Snap-collector.

We optimized our lock-free RQ Provider by using the technique of Arbel-Raviv and Brown [4] to allow DCSS operations to efficiently reuse a single descriptor object, instead of allocating a new descriptor for each operation.

Experiment 1: one thread performing RQs. We start by running a simple microbenchmark to compare the performance of the different RQ algorithms. For each data structure and RQ algorithm, we run five 3-second *trials* for several thread counts n. In each trial, $n - 1$ threads perform 50% insertions and 50% deletions on keys drawn uniformly from $[0, K)$ (where $K = 10^6$ for ABTree, $K = 10^5$ for LFBST, Citrus and SkipList, and $K = 10^4$ for LFList and LazyList), and one thread performs RQs on range $[low, low + 100)$, where low is uniform in $[0, K - 100)$. Data structures are prefilled with

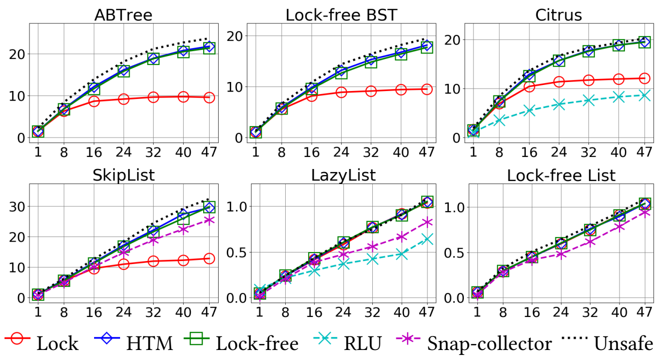

Figure 5. Experiment 1: one thread performs RQs. Total operations per μs (y-axis) are plotted versus the number of threads *performing insertions and deletions* (x-axis).

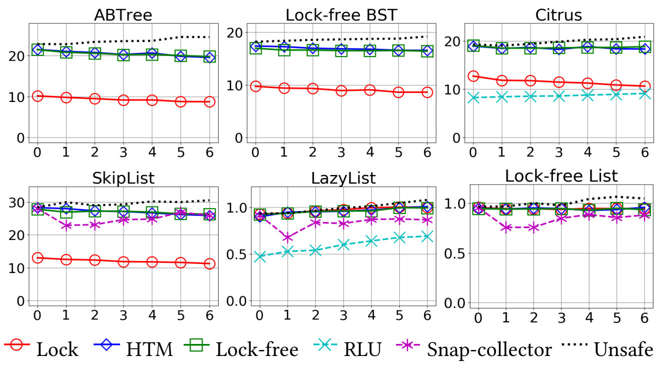

Figure 6. Experiment 2: overhead caused by RQs. Total operations per μs (y-axis) are plotted versus the number of threads *performing range queries* (x-axis).

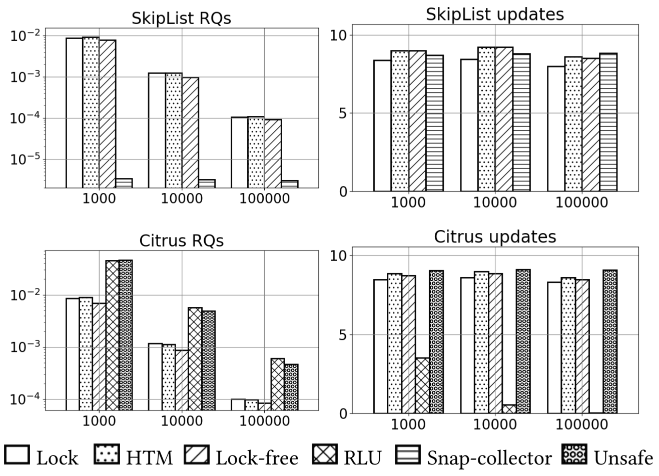

Figure 7. Experiment 3: varying RQ size. **[Left]** RQs per μs (*logarithmic* y-axis) versus size of range (x-axis). **[Right]** Updates per μs (y-axis) versus size of range (x-axis).

approximately $K/2$ keys before each trial (and were found to contain approximately $K/2$ keys at the *end* of each trial).

Results appear in Figure 5. The results demonstrate that Lock-free and HTM are almost as fast as Unsafe. The Snap-collector is sometimes slower than Lock-free and HTM, and sometimes performs similarly.

In some cases, Lock is considerably slower than Lock-free and HTM. This is because of high contention on the lock L. This contention is greatly improved by the use of transactions in HTM. It could also be improved by using a scalable NUMA-aware cohort r/w-lock [12], instead of the simplistic single-word fetch-and-add r/w-lock that we used.

Additionally, Lock-free and HTM are significantly faster than RLU in Citrus, and in LazyList with 100% updates. This is because each update in RLU must synchronize with *all* other operations, regardless of whether they access the same data. This effect is not as noticeable in LazyList with 20% updates, because of the low update rate.

To study the overhead of traversing limbo lists in our algorithms, we also measured (a) how many nodes were visited in limbo lists by the first 10,000 RQs, and (b) the total size of limbo lists at the end of each trial. For example, in LFBST, 99.7% of these RQs only visited between 128 and 1,024 nodes in limbo lists, whereas the limbo lists contained up to 780,000 nodes. This suggests that the optimizations described in §4.3 are highly effective. Additionally, we found that the size of the data structure did not significantly change how many nodes are visited in limbo bags.

It turns out that we can use our timestamps to validate the correctness of iteration operations (RQs over the entire data structure). Due to lack of space, we defer a description of our validation technique to the full version of this paper.

Experiment 2: overhead caused by RQs. We are also interested in understanding how much overhead is added by RQs in Experiment 1. Figure 6 shows the results of a similar experiment in which 42 threads performed 100% updates, and the number of threads that perform 100% RQs varies. The results show that the impact of adding more RQ threads is relatively small. The trees are more affected than the lists. This is because RQs in a list are no slower than updates (since both must traverse the list), so the loss in update throughput is obscured by increased RQ throughput.

Experiment 3: varying RQ size. We also wanted to study how RQ size affects the performance of RQs and updates. Figure 7 shows results for a workload in which 47 threads performed 20% updates and 80% searches, and one thread performed 100% RQs, where the size of the range varies. We provide two graphs each for SkipList and Citrus showing *RQ throughput* (left) and *update throughput* (right). We use a *logarithmic* y-axis for the RQ throughput graphs, since the larger RQs have much lower absolute throughputs.

We first compare our techniques with the Snap-collector. As the left SkipList graph shows, the Snap-collector is orders of magnitude slower, even for RQs over the *entire data structure* (i.e., for *iteration*). This demonstrates that our techniques are dramatically faster, not simply because they can avoid taking a full snapshot of the data structure, but also because

	Lock	HTM	Lock-free	RLU	Snap.
ABTree	12.00	29.46	32.94	-	-
Lock-free BST	12.81	27.19	26.28	-	-
Citrus	14.07	31.78	32.26	22.26	-
SkipList	13.74	27.95	29.18	-	0.05
LazyList	1.20	1.10	1.10	1.01	0.45
Lock-free List	0.77	0.77	0.68	-	0.46

Figure 8. Experiment 4: mixed workload (operations per μs).

	Lock	HTM	Lock-free	RLU	Unsafe
ABTree	0.92	0.92	0.92	-	0.97
Lock-free BST	0.70	0.70	0.71	-	0.73
Citrus	0.77	0.78	0.77	0.68	0.80
SkipList	0.62	0.63	0.62	-	0.65

Figure 9. TPC-C benchmark (database transactions per μs).

they incur lower overhead per visited node. (Although we did not compare with the extended Snap-collector of Chatterjee [13], these results suggest that our techniques would be faster, since [13] only improves the performance of the Snap-collector by eliminating the need to take full snapshots, and actually adds substantial overhead.) The right graph shows that our techniques also impose less overhead on updates.

Next, we compare our techniques with RLU and Unsafe. The left Citrus graph shows that Unsafe provides much faster RQs than our techniques. Interestingly, RLU's RQs are as fast as Unsafe's RQs. However, as the right graph shows, RLU heavily prioritizes RQs over updates, to the point of *crippling* update throughput. This is not a reasonable trade-off for many applications.

Experiment 4: mixed workload. Figure 8 shows results for a more realistic microbenchmark in which 48 threads each perform 10% insertions, 10% deletions, 78% searches, and 2% RQs (over ranges of size 100). In this workload, Snap-collector (Snap.) was up to several orders of magnitude slower than the other algorithms (although the effect was much smaller in the lists, since traversing the entire data structure is only twice as expensive searching for a key, on average).

Application benchmark: TPC-C. Figure 9 shows the results of running the TPC-C database workload, with 48 threads and 48 warehouses, on a simple database management system called DBx (as in [27]). TPC-C features complex transactions over nine tables with widely varying row types and population sizes, and with varying degrees of non-uniformity in the data. Transactions perform a large number of searches, updates and RQs. Each table is indexed by up to three different indexes on different key fields. We replaced the database indexes with each of our data structure implementations. (Note that DBx originally used hash tables for its indexes, so it did not support arbitrary range queries. We implemented true range query support in DBx with our indexes. Because of this, it does not make sense to compare our algorithms with the original DBx implementation.)

We only show results for the trees and skip-list. (The other data structures are linked lists, which would have taken days to simply initialize, due to their linear traversals.) The results show that our algorithms perform almost as well as Unsafe. When RLU is used, performance is much lower than when the other algorithms are used, although the difference is smaller than in our other experiments. This is because index accesses make up a relatively small part of the total runtime.

The Snap-collector is not shown, because it was 1000x slower than the other algorithms. This is a consequence of the fact that TPC-C has a large dataset (occupying up to 40GB of memory), and some of the indexes are very large. Since the Snap-collector must take a snapshot of an entire index to compute a RQ, it is simply not practical.

6 Conclusion

In this work, we presented three new algorithms for adding RQ support to existing concurrent set implementations, using locks, HTM and lock-free techniques. Our new algorithms can be used with many more data structures than previous approaches. We demonstrated the use of our algorithms by applying them to six different data structures (including three that *cannot* be used with previous approaches). We have also used our algorithms to add RQ support to a lock-free relaxed B-slack tree, a space-efficient balanced tree [10] (not shown in our experiments).

Our experiments showed that our algorithms are extremely fast, often matching the performance of an unsafe (non-linearizable) RQ algorithm that simply traverses the data structure once and collects keys. Moreover, our algorithms significantly outperformed previous approaches in a variety of workloads, including a realistic database application.

Finally, we addressed a hole in the theoretical underpinning of two previous approaches [13, 24]. Each of these approaches assumed the existence of a traversal procedure that collects nodes in the data structure, but did not consider what kind of property such a procedure must satisfy to be useful for constructing RQs. We defined the COLLECT property, proved that it holds for a large class of data structures, and proved that it can be used to construct linearizable RQs.

Acknowledgments

This work was supported by the Israel Science Foundation, grant no. 1749/14. Maya Arbel-Raviv was supported in part by Global Affairs Canada and the Technion Hasso Platner Institute Research School. Trevor Brown was funded in part by PhD and post-doctoral fellowships from the Natural Sciences and Engineering Research Council of Canada, and grant no. RGPIN-2015-05080.

A Artifact description

A.1 Abstract

This artifact includes C++ code for all data structures, range query techniques and benchmarks studied in the paper, as well as bash scripts for running the experiments shown in the paper, and Python scripts for producing graphs. It is intended to be run on a large scale machine with support for Intel's transactional synchronization extensions (TSX). We suggest a machine with at least 16 threads, although a 48-thread 2-socket machine will make it easiest to reproduce the experiments in the paper. The artifact will run on machines without TSX support, but it will not produce data for the transactional memory based algorithms that we studied. One should be able to replicate the experiments, and see that the relative performance relationships between algorithms reflect what we present in our experiments.

A.2 Description

A.2.1 Check-list (artifact meta information)

- **Algorithm:**
  We implement the following data structures:
  1. External binary search tree - lock-free
  2. Internal binary search tree - fine-grained locks and read-copy-update (RCU)
  3. Internal binary search tree - read-lock-update (RLU)
  4. Relaxed b-slack tree - lock-free
  5. Relaxed (a,b)-tree - lock-free
  6. Skip list - fine-grained locks
  7. Linked list - lock-free
  8. Linked list - fine-grained locks and optimistic validation
  9. Linked list - read-log-update (RLU)

  We implement several methods for adding range queries to these data structures:
  (a) Our lock-based technique
  (b) Our HTM and lock-based technique
  (c) Our lock-free technique
  (d) A non-linearizable single-collect technique
  (e) The snap collector of Petrank and Timnat [24]
  (f) RLU snapshots (can only be used with RLU-based data structures)

  Table 1 present the pairs of data structures and range query techniques that can be used.

  The epoch-based memory reclamation for our range query techniques is a slightly modified version of DEBRA [9].
- **Program:** Two benchmarks are included. *Microbench* is a simple set/dictionary benchmark. *Macrobench* is a modified version of the DBx1000 TPC-C database benchmark. We replaced the original DBx1000 database indexes with our data structure implementations.
- **Compilation:** Using `gcc` version 4.8 or higher.
- **Binary:** We include the following libraries, which were compiled for x86/64 on Ubuntu 14.04 LTS.
  1. jemalloc memory allocator version 5.0.1 (`libjemalloc.so`)
  2. papi performance counters, optional use (`libpapi.a`)
- **Run-time environment:** Recent Linux (tested on Ubuntu 14.04 LTS). Software dependencies: GCC 4.8+, Make, Python 2.7+ (NOT 3.x) with libraries: numpy 1.12+, matplotlib 2.1+, pandas 0.13.1+. If software is up to date then no root access is required (otherwise, root access may be needed to install these dependencies).
- **Hardware:** Multicore system (preferably with at least 16 hardware threads). Memory model: total store order (standard on modern Intel/AMD/Oracle processors) For HTM results, an Intel system with TSX support is mandatory. (On other systems, the benchmarks will run, but data will not be produced for the HTM-based algorithms.)
- **Execution:** Our benchmarks should be the only userspace application running on the machine.
- **Output:** For the microbenchmark, a human readable text file is produced for each trial, in each experiment. Relevant parts of this output are parsed and stored in an SQLite database (`microbench/results.db`). The macrobenchmark also produces a human readable text file for each trial, in each experiment. It parses relevant parts of this output and stores them in a CSV file. A Python script for producing graphs from the experimental results is also provided.
- **Experiment workflow:** Edit one configuration file, specifying the number of threads and speed of the experimental machine, then compile, run experiments and produce graphs via a set of bash and Python scripts.
- **Experiment customization:** Experiment customization is possible; see instructions below.
- **Publicly available?:** Yes, the artifact is publicly available via a Git repository.

		Data Structure								
		1	2	3	4	5	6	7	8	
RQ	a	x	x		x	x	x	x	x	
	b	x	x		x	x	x	x	x	
	c	x	x		x	x	x	x	x	
	d	x	x		x	x	x	x	x	(non-linearizable)
	e						x	x	x	
	f			x						x

Table 1. Pairs of data structures and range query techniques that can be used. Note that RLU (used in data structures 3 and 9) is a complete synchronization methodology that dictates how all queries and updates are implemented. Conceptually, data structures 2 and 3 are the same (and 8 and 9 are the same).

A.2.2 How the artifact is delivered

The artifact is available at: http://implementations.tbrown.pro (path to this artifact: `/cpp/range_queries`).

A.2.3 Hardware dependencies

Minimum requirements: multicore system with 16+ hardware threads and total store order (TSO) memory model (standard on modern Intel/AMD/Oracle processors). The macrobenchmark requires up to 2GB of RAM *per hardware thread*. (Otherwise, the working set might not fit in memory).

Recommended requirements: Intel system with $n \geq 32$ hardware threads and support for transactional synchronization extensions (TSX), with $2n$ GB of RAM.

An Intel system with TSX support is **mandatory** for algorithms that use hardware transactional memory (HTM). On other systems, the benchmarks will run, but data will not be produced for HTM-based algorithms.

A.2.4 Software dependencies

Recent Linux distribution (tested on Ubuntu 14.04 LTS). GCC 4.8+ (we used 6.3.0). Make. Python 2.7+ (NOT 3.x) with libraries: numpy 1.12+, matplotlib 2.1+, pandas 0.13.1+.

A.3 Installation

1. Install software dependencies. The list above contains links to installation instructions available online.
2. Clone the GIT repository by executing:
   `git clone https://bitbucket.org/trbot86/implementations.git`
   This clones the repository into a new directory.
3. Navigate to this directory by executing:
   `cd implementations/cpp/range_queries/`

A.4 Experiment workflow

1. Edit `config.mk`, following the instructions therein.
2. Run: `find . -type f -print0 | xargs -0 dos2unix`
3. Microbench:
```
cd microbench
make -j
chmod +x *.sh
./runscript.sh
python graphs.py
```
4. Macrobench:
```
cd macrobench (or cd ../macrobench)
chmod +x *.sh
./compile.sh
./runscript.sh
./makecsv.sh > dbx.csv
python graph.py
```

A.5 Evaluation and expected result

After following the experiment workflow for the microbenchmark and macrobenchmark, one should analyze the graphs produced (which are located in `microbench/graphs` and `macrobench/graphs`).

Broadly speaking, one should be able to confirm that these graphs reproduce the results shown in the experiments section of this paper. In other words, one should see that the relative performance relationships between algorithms studied in our experiments hold (even if the absolute performance measured differs from what we presented in our experiments).

In the graphs for Experiment 1, one should see that our HTM and Lock-free algorithms match or exceed the performance of the other algorithms. Depending on the system architecture, one might see that our lock based algorithm is just as fast as, or is considerably slower than, the other algorithms. (We have noticed that our lock-based implementation, which favors threads acquiring the lock in shared mode, performs poorly on some AMD systems. This is a performance problem affecting the lock implementation, not our range query technique.) One should see that RLU is substantially slower than the other algorithms when there are many updates, and in the skiplist (since it is a logarithmic data structure).

In the graphs for Experiment 2, one should see that Unsafe is slightly faster than the other algorithms. One might see that Lock is substantially slower than the other algorithms in the logarithmic data structures (ABTree, Lock-free BST, Citrus and SkipList). One should see that RCU is much slower than the other algorithms, with the possible exception of the Lock-based algorithm in Citrus. One should see that the curves are fairly flat, meaning that performance does not degrade severely as the number of range query threads increases.

In the graphs for Experiment 3, one should see that the update throughput in SkipList does not substantially degrade as the size of range queries increases. Additionally, all algorithms should have similar update throughputs in SkipList, with the possible exception of Lock, which might have a somewhat smaller update throughput. One should see that Lock, HTM and Lock-free have similar range query throughput in SkipList, and that Snap-collector's range query throughput is orders of magnitude smaller. In Citrus, the performance of updates should not substantially degrade for all algorithms, except RLU, which should see severe degradation. One should see that RLU's updates are substantially slower than Lock, HTM and Lock-free for Citrus. However, RLU's range queries should be faster than the other techniques'. In both data structures, for all algorithms, one should see that range query throughput approximately degrades by a factor of 10 with each increase in range query size.

In the results for Experiment 4, one should see that the Snap-collector is substantially slower than HTM and Lock-free in the Lock-free List. One might also see that the Snap-collector is much slower than Lock in the Lock-free List. In the SkipList, one should see that the Snap-collector is orders of magnitude slower than the other algorithms.

In the results for the macrobenchmark, one should see that HTM and Lock-are slightly slower than Unsafe. One should see that RLU is substantially slower than the other algorithms (with the possible exception of Lock).

A.6 Experiment customization

Experiment customization is available by changing the arguments to the microbench and macrobench binaries. A full description of the run-time arguments and command line examples are available in the README file.

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, Sept. 1993. ISSN 0004-5411. doi: 10.1145/153724.153741. URL http://doi.acm.org/10.1145/153724.153741.

[2] A. Agarwal, Z. Liu, E. Rosenthal, and V. Saraph. Linearizable iterators for concurrent data structures. *CoRR*, abs/1705.08885, 2017. URL http://arxiv.org/abs/1705.08885.

[3] M. Arbel and H. Attiya. Concurrent updates with RCU: Search tree as an example. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 196–205, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2944-6. doi: 10.1145/2611462.2611471. URL http://doi.acm.org/10.1145/2611462.2611471.

[4] M. Arbel-Raviv and T. Brown. Reuse, don't recycle: Transforming lock-free algorithms that throw away descriptors. In *Proceedings of the 31st International Symposium on Distributed Computing*, DISC 2017, 2017.

[5] H. Attiya, R. Guerraoui, and E. Ruppert. Partial snapshot objects. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 336–343, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-973-9. doi: 10.1145/1378533.1378591. URL http://doi.acm.org/10.1145/1378533.1378591.

[6] H. Avni, N. Shavit, and A. Suissa. Leaplist: Lessons learned in designing tm-supported range queries. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 299–308, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2065-8. doi: 10.1145/2484239.2484254. URL http://doi.acm.org/10.1145/2484239.2484254.

[7] D. Basin, E. Bortnikov, A. Braginsky, G. Golan-Gueta, E. Hillel, I. Keidar, and M. Sulamy. Kiwi: A key-value map for scalable real-time analytics. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 357–369, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4493-7. doi: 10.1145/3018743.3018761. URL http://doi.acm.org/10.1145/3018743.3018761.

[8] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 257–268, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-877-3. doi: 10.1145/1693453.1693488. URL http://doi.acm.org/10.1145/1693453.1693488.

[9] T. Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 261–270, 2015.

[10] T. Brown. *Techniques for Constructing Efficient Data Structures.* PhD thesis, University of Toronto, 2017.

[11] T. Brown, F. Ellen, and E. Ruppert. A general technique for non-blocking trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 329–342, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2656-8. doi: 10.1145/2555243.2555267. URL http://doi.acm.org/10.1145/2555243.2555267.

[12] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, and N. Shavit. Numa-aware reader-writer locks. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 157–166, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1922-5. doi: 10.1145/2442516.2442532. URL http://doi.acm.org/10.1145/2442516.2442532.

[13] B. Chatterjee. Lock-free linearizable 1-dimensional range queries. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, ICDCN '17, pages 9:1–9:10, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4839-3. doi: 10.1145/3007748.3007771. URL http://doi.acm.org/10.1145/3007748.3007771.

[14] L. Dalessandro, M. F. Spear, and M. L. Scott. Norec: Streamlining stm by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 67–78, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-877-3. doi: 10.1145/1693453.1693464. URL http://doi.acm.org/10.1145/1693453.1693464.

[15] T. David, R. Guerraoui, and V. Trigonakis. 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 '15, pages 631–644, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2835-7. doi: 10.1145/2694344.2694359. URL http://doi.acm.org/10.1145/2694344.2694359.

[16] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-44624-9, 978-3-540-44624-8. doi: 10.1007/11864219_14. URL http://dx.doi.org/10.1007/11864219_14.

[17] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 131–140, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-888-9. doi: 10.1145/1835698.1835736. URL http://doi.acm.org/10.1145/1835698.1835736.

[18] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02, pages 265–279, 2002.

[19] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. ISBN 0123705916, 9780123705914.

[20] P. Jayanti. An optimal multi-writer snapshot algorithm. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, STOC '05, pages 723–732, New York, NY, USA, 2005. ACM. ISBN 1-58113-960-8. doi: 10.1145/1060590.1060697. URL http://doi.acm.org/10.1145/1060590.1060697.

[21] A. Matveev, N. Shavit, P. Felber, and P. Marlier. Read-log-update: A lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 168–183, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3834-9. doi: 10.1145/2815400.2815406. URL http://doi.acm.org/10.1145/2815400.2815406.

[22] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[23] A. Natarajan and N. Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 317–328, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2656-8. doi: 10.1145/2555243.2555256. URL http://doi.acm.org/10.1145/2555243.2555256.

[24] E. Petrank and S. Timnat. Lock-free data-structure iterators. In *Proceedings of the 27th International Symposium on Distributed Computing - Volume 8205*, DISC 2013, pages 224–238, New York, NY, USA, 2013. Springer-Verlag New York, Inc. ISBN 978-3-642-41526-5. doi: 10.1007/978-3-642-41527-2_16. URL http://dx.doi.org/10.1007/978-3-642-41527-2_16.

[25] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky. Concurrent tries with efficient non-blocking snapshots. *SIGPLAN Not.*, 47(8):151–160, Feb. 2012. ISSN 0362-1340. doi: 10.1145/2370036.2145836. URL http://doi.acm.org/10.1145/2370036.2145836.

[26] K. Sagonas and K. Winblad. Efficient support for range queries and range updates using contention adapting search trees. In *Revised Selected Papers of the 28th International Workshop on Languages and Compilers for Parallel Computing - Volume 9519*, LCPC 2015, pages 37–53, New York, NY, USA, 2016. Springer-Verlag New York, Inc. ISBN 978-3-319-29777-4. doi: 10.1007/978-3-319-29778-1_3. URL http://dx.doi.org/10.1007/978-3-319-29778-1_3.

[27] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one

thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, Nov. 2014. ISSN 2150-8097. doi: 10.14778/2735508.2735511. URL http://dx.doi.org/10.14778/2735508.2735511.