Lock-free Contention Adapting Search Trees

KJELL WINBLAD, Ericsson AB, Sweden
KONSTANTINOS SAGONAS, Uppsala University, Sweden
BENGT JONSSON, Uppsala University, Sweden

Concurrent key-value stores with range query support are crucial for the scalability and performance of many applications. Existing lock-free data structures of this kind use a fixed synchronization granularity. Using a fixed synchronization granularity in a concurrent key-value store with range query support is problematic as the best performing synchronization granularity depends on a number of factors that are difficult to predict, such as the level of contention and the number of items that are accessed by range queries. We present the first linearizable lock-free key-value store with range query support that dynamically adapts its synchronization granularity. This data structure is called the lock-free contention adapting search tree (LFCA tree). An LFCA tree automatically performs local adaptations of its synchronization granularity based on heuristics that take contention and the performance of range queries into account. We show that the operations of LFCA trees are linearizable, that the lookup operation is wait-free, and that the remaining operations (insert, remove and range query) are lock-free. Our experimental evaluation shows that LFCA trees achieve more than twice the throughput of related lock-free data structures in many scenarios. Furthermore, LFCA trees are able to perform substantially better than data structures with a fixed synchronization granularity over a wide range of scenarios due to their ability to adapt to the scenario at hand.

Additional Key Words and Phrases: concurrent data structure, range query, lock-freedom, wait-freedom, linearizability, adaptivity

1 INTRODUCTION

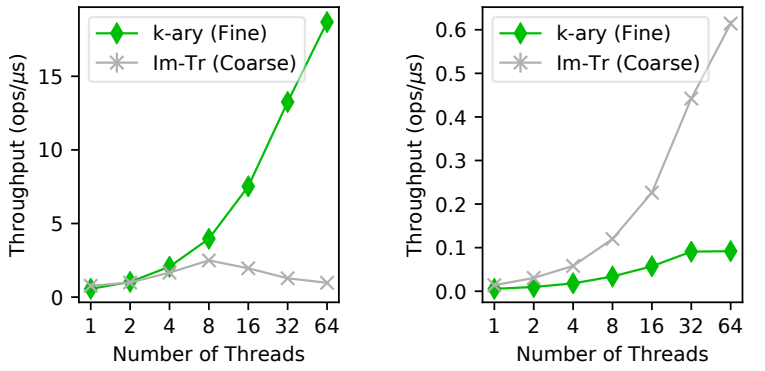
On multicore machines, concurrent key-value stores (maps) with range query support are crucial for the scalability of applications such as big scale data processing and in-memory databases (e.g., Google's F1 [Shute et al. 2013] and Yahoo's Flurry [Yahoo! Developer Network 2017]). It is thus of no surprise that the multicore revolution has motivated researchers (e.g., Avni et al. [2013]; Basin et al. [2017]; Brown and Avni [2012]; Sagonas and Winblad [2017]) to propose a number of data structures of this type. A key-value store represents a set of items (keys), each with an associated value. Sets can be seen as a simplification of key-value stores that do not have any values associated with the items. From here on, we will discuss sets but we note that what applies to sets also applies to key-value stores as sets can trivially be modified to become key-value stores.

Concurrent sets that support both single-item operations (insert, remove and lookup[1]) and multi-item operations (e.g., range query and clone[2]) face the following dilemma: Single-item operations usually benefit from as fine-grained synchronization as possible, as this leads to few conflicts. In contrast, multi-item operations usually benefit from more coarse-grained synchronization, as this leads to less time spent on synchronization-related overhead (e.g., fewer locks need to be acquired). We say that the *conflict time* for an operation is the amount of the time during which the operation

[1]An insert operation inserts an item (replacing an existing item if one with an equal key already exists), the remove operation removes an item with the given key if such an item exists and the lookup operation returns an item with the given key if such an item exists.
[2]A range query operation returns all items with keys within the given range (specified by two keys) and clone makes a clone of the data structure.

Authors' addresses: Kjell Winblad, Ericsson AB, Kista, Sweden, kjellwinblad@gmail.com; Konstantinos Sagonas, Uppsala University, Uppsala, Sweden, kostis@it.uu.se; Bengt Jonsson, Uppsala University, Uppsala, Sweden, bengt@it.uu.se.

2021. 1539-9087/2021/4-ART42 $15.00
https://doi.org/10.1145/???.??



(a) Scenario with small range queries (b) Scenario with large range queries

Fig. 1. Throughput of coarse- vs fine-grained synchronization.

can conflict with other concurrent operations. More coarse-grained synchronization can also lead to short conflict times for operations in concurrent sets that internally use and exploit immutable data in a way that we will now explain. We will use the lock-free k-ary search tree of Brown and Avni [2012] to illustrate how immutable data can be used to make the conflict time for range queries short. Lock-free k-ary search trees store all items inside immutable leaf nodes that can contain k items each. The insert and remove operations of such trees work by replacing leaf nodes. A range query Q in a k-ary search tree first collects all the immutable leaf nodes that Q needs. The range query Q ends its conflict time and linearizes once this collection phase finishes. The items that Q needs to return are scanned from the collected leaf nodes after Q's conflict time. This removal of this scanning from the conflict time is possible due to the immutability of the leaf nodes.

An even more extreme way to exploit immutability in range queries is to store all items in a single immutable data structure. Such a data structure, that we call *Im-Tr*, is constructed from a single mutable reference pointing to an immutable balanced search tree T. A new instance of T reflecting an update (insert or remove) can be constructed in $O(\log n)$ time (where n is the number of items before the update) as one only needs to "copy" nodes on a path from the root to a leaf to create the new instance [Okasaki 1999]. The update operations of *Im-Tr* change the mutable reference using an atomic compare-and-swap (CAS) operation so the reference points to a new immutable instance reflecting the update. Using this scheme, which is also described by Herlihy [1990], it is trivial to perform range queries with constant conflict time as they only need to get a snapshot by reading the mutable reference and then perform the range query in the snapshot.

Figure 1[3] illustrates the scalability of a data structure that uses a fixed relatively fine-grained synchronization (the lock-free k-ary search tree with the k parameter set to 64) and one that uses coarse-grained synchronization (*Im-Tr*) in two scenarios that only differ in the sizes of the ranges involved in the range queries. As shown in the graphs, fine-grained synchronization achieves superior scalability when the size of ranges in range queries is small (Fig. 1a) as the mutable reference of *Im-Tr* gets heavily contended, while *Im-Tr* with coarse-grained synchronization has superior scalability when range queries are large (Fig. 1b) as the k-ary search tree's range queries have long conflict times in this scenario.

[3]The benchmark used to produce the graphs in Fig. 1 was executed on an Intel machine with 64 logical cores using the Oracle JVM. With small and large range queries we refer to range queries that on average include about 2.5 items and 25k items respectively. Further details about the experiment can be found in Section 7.

As the above example illustrates, having concurrent sets with a fixed synchronization granularity is far from ideal. A natural way to deal with this problem is to design sets that can adapt their synchronization granularity to the workload at hand. In earlier work [Sagonas and Winblad 2017], we have described the first data structure that dynamically changes its synchronization granularity based on heuristics that take both the performance of single-item operations and multi-item operations into account, called the contention adapting search tree (*CA tree* for short). The CA tree is lock-based.

The CA tree performs well in a variety of scenarios due to its ability to adapt its synchronization granularity to the workload at hand. However, lock-based data structures are prone to a number of problems that are inherent from the use of locks, e.g., waiting, priority inversion, convoying, and lock overhead (memory usage, acquire and release time). The performance of lock-based algorithms is also heavily dependent on the lock implementation itself and on OS scheduling. For all these reasons, lock-free data structures that guarantee system-wide progress even in the presence of adversary scheduling are preferable to lock-based ones. In particular, because in many applications lookups are very common, it is important that the lookup operation is wait-free (i.e., finishes in a finite number of steps [Herlihy 1991]), which is something that the lock-based CA tree is lacking.

This article presents a lock-free variant of the CA tree, called the *lock-free contention adapting search tree* (*LFCA tree* for short). LFCA trees support lock-free insert, remove and range query operations as well as a wait-free lookup operation. Operations update the data stored in an LFCA tree by swapping immutable leaf nodes pointing to immutable balanced search trees storing the actual items. The granularity of the LFCA tree is adjusted by splitting and joining such leaf nodes. Initiation of split and join operations is based on a statistics value, which is stored in each leaf node, and which is updated based on `CAS` successes or failures as well as the number of leaf nodes accessed by range queries.

The technique that the LFCA tree uses for supporting range queries is interesting in its own right, as it is applicable to other lock-free data structures such as the lock-free k-ary search tree of Brown and Avni [2012]. A range query operation is performed by replacing the needed leaf nodes with nodes of a special node type which contain the information that other threads need when helping to complete the operation. As noted by both Basin et al. [2017] and Winblad [2018], the previously proposed method for doing range queries in the k-ary search tree is prone to starvation.

Overall, we claim that LFCA trees are important concurrent data structures as they provide a unique set of desirable properties:

Efficient non-blocking operations Our experimental comparison shows that LFCA trees achieve substantially better throughput than the best of the competing lock-free data structures over a wide range of scenarios. Also, LFCA trees perform better than lock-based CA trees in many scenarios; especially in scenarios with more threads than hardware threads.

Configuration-less As an LFCA tree automatically adjusts its structure using heuristics, there is no need for the user to configure the LFCA tree to use a certain synchronization granularity.

Adaptive As the synchronization granularity changes with the workload, the data structure can perform very well even when the workload changes during the lifetime of the data structure. As the adaptations of the granularity happen through local changes, LFCA trees can even adapt to scenarios where the workload is different in different parts of their structure.

Flexible Performance characteristics of an LFCA tree can be changed by providing a different set implementation. We do not experiment with this property here, but see no reason why the LFCA tree would be different from the lock-based CA tree [Sagonas and Winblad 2017] in this regard.

This article is the journal version of our SPAA 2018 conference paper with the same title [Winblad et al. 2018]. It has both been extended and differs from it in the following ways:

- The LFCA tree presented in the SPAA 2018 paper does not have a wait-free lookup operation even though that paper claims so. We explain the problem (Fig. 3 in Section 4), and fix it by updating the code that implements LFCA tree's operations. Consequently, all experiments presented in this article have been conducted with the updated code to properly measure the LFCA tree's performance. By comparing the graphs in the two papers, one can indeed notice that the fix does not cause any significant change to the performance of the LFCA tree.
- The related work (Section 3) has been extended with descriptions and discussions of recently proposed data structures and methods for concurrent range queries that were either unknown to us at the time of the SPAA 2018 submission, or appeared after the final version of our conference paper was submitted for publication.
- The correctness argument (Section 5) has got a major rewrite and that section has been significantly extended to make it more detailed and easier to follow.
- The performance evaluation (Section 7), besides being done from scratch due to the updated code, has been significantly extended to consider many more scenarios, and now spans several pages.

Outline. We start with a brief high-level description of how LFCA trees work (Section 2) followed by an overview of related work (Section 3). Subsequently, we describe the algorithm in detail (Section 4), present an argument for its correctness (Section 5) and describe an optimization for range queries (Section 6). Finally, we experimentally compare the performance of LFCA trees against state-of-the-art concurrent data structures with similar functionality (Section 7), and end with some concluding remarks (Section 8).

2 A BIRD'S EYE VIEW OF LFCA TREES

A lock-free CA tree (LFCA tree) consists of *route nodes* (round boxes in Fig. 2a) and *base nodes* (square shaped boxes in Fig. 2a). The route nodes form a binary search tree with the base nodes as leaves. The actual items that are stored in the set represented by an LFCA tree are located in immutable data structures rooted in the base nodes, called *leaf containers*. All operations use the binary search tree property of the route nodes to find the base node(s) whose leaf container(s) should contain the items involved in the operation if they exist. An update operation (insert or remove) is illustrated by Figs. 2a and 2b. An update operation uses a compare-and-swap (CAS) to attempt to replace a base node b_1 with a new base node b_4 reflecting the update, until the update succeeds. The update can be made efficient even though the data structure in the base node is immutable, since many immutable self-balancing binary search tree data structures support creating a new instance with an update in $O(\log n)$ time, where n is the number of items in the data structure [Okasaki 1999]. Before an update operation returns, it checks whether the statistics value stored in the updated base node indicates that a structural adaptation should happen. The first kind of adaptation, called *split*, is illustrated by Figs. 2a and 2c. A split aims at reducing the contention in the LFCA tree and replaces a base node b_1 with a route node linking together two new base nodes (b_4 and b_5) so that approximately half of the original items are in each of them. The second kind of adaptation, called *join*, is illustrated with Figs. 2a and 2d and aims at optimizing the structure of the LFCA tree for range queries that span multiple base nodes and for situations where the contention is low. A join splices out a base node b_2 and its parent and replaces the base node b_3 with a new base node b_4 containing the items of both b_2 and b_3. Splits and joins of the base nodes can also be supported efficiently (i.e., in $O(\log n)$ time, where n is the number of items in the involved instances) in many immutable balanced tree data structures; for example, in *treaps* that

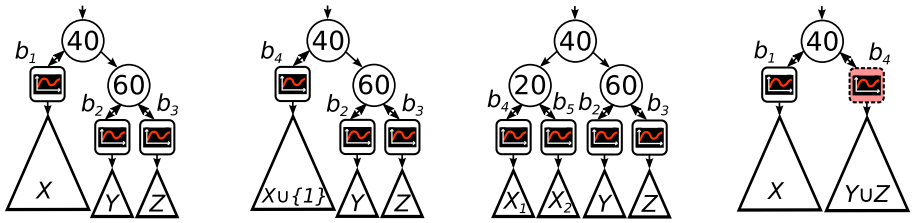


Fig. 2. LFCA Trees illustrating various operations.

are used by our LFCA tree implementation. Treaps [Seidel and Aragon 1996] are self-balancing binary search trees that order tree nodes by a priority number as well as the node's key, thereby achieving $O(\log n)$ height with high probability.

3 RELATED WORK

There are several data structures with range query support. The *SnapTree* by Bronson et al. [2010] has an efficient linearizable clone operation that returns a copy of the data structure from which a range query operation can easily be derived. *SnapTree*'s clone operation waits for active update operations to complete and forces subsequent update operations to copy nodes lazily before node modifications, so that the clone is not modified.

The lock-free k-ary search tree is an external (i.e. the items are stored in leaf nodes) unbalanced search tree with up to k keys stored in every node [Brown and Helga 2011]. Range queries in k-ary search trees are performed by doing a read scan followed by a validation scan of the immutable leaf nodes containing items in the range [Brown and Avni 2012]. The range query operation needs to retry if the validation scan fails. The k-ary search tree is an example of the fixed synchronization granularity approach discussed in the introduction. Another example of this approach based on software transactional memory is the Leaplist of Avni et al. [2013]. Both the k-ary search tree and the Leaplist make use of immutable data structures to reduce the conflict time of range queries in the way explained in the introduction. As they both use arrays as their immutable data structures, updates become very expensive when the parameter that decides both the synchronization granularity and the maximum size of the immutable data structures is set too high. Even though this problem could be fixed by using immutable balanced search trees instead of the arrays, these two data structures would still use a fixed synchronization granularity and would thus only perform well in certain scenarios.

Chatterjee [2017] has proposed a general method for performing range queries in lock-free ordered set data structures based on an idea for doing snapshots by Petrank and Timnat [2013]. Chatterjee's method makes use of a list of so-called *range collector objects* that all updates and range queries need to access. Unfortunately, the scalability of Chatterjee's method suffers from a global sequential hot spot in the list of range-collector objects that all range queries have to modify in the worst case.

The *KiWi* data structure by Basin et al. [2017] supports wait-free range queries and lookup operations as well as lock-free update operations. Update operations help range queries by storing additional versions of inserted items when it is needed for the range queries. Similarly to a data structure by Robertson [2014], *KiWi*'s range queries atomically increment a global version counter which is used by update operations to decide whether storing an additional version for an item is necessary or not. *KiWi*'s global version number counter is bound to become a scalability bottleneck

with a high enough level of parallel range queries. LFCA trees do not suffer from such a global scalability bottleneck as their range query operation only needs to synchronize with update operations that operate on items in the same range as the range query.

Arbel-Raviv and Brown [2018] recently proposed a general method for extending concurrent data structures with linearizable range query support. As they report, their implementation of this new method appears to perform substantially better compared with both the general method for linearizable iterators proposed by Petrank and Timnat [2013] and the method for range queries proposed by Chatterjee [2017]. With their method, range queries increment a global timestamp variable. Update operations write down the timestamps in the relevant nodes when nodes are inserted and removed. Range queries traverse the data structure to collect nodes with items in the range. Range queries also traverse nodes that have been removed during such a traversal (such nodes can be found in the epoch-based memory reclamation system of Brown [2015] that the method relies upon). Range queries figure out which of the traversed nodes are relevant for the results based on the timestamps in the nodes. Unfortunately, the global timestamp counter is bound to become a scalability bottleneck once parallel range queries become frequent enough. Furthermore, the global timestamp counter induces an overhead for all update operations, especially when this counter is frequently modified, as all update operations have to read this counter. In contrast, our CA trees do not have such a global counter, and do not rely on being able to access the internals of the memory reclamation system.

In an unpublished technical report, Agarwal et al. [2017] describe an extension to the method by Petrank and Timnat [2013] for creating linearizable iterators that make the method applicable to more data structures. The results provided by that report [Agarwal et al. 2017] indicate that the extended method suffers from similar performance overhead as the original method.

While our article was under review, Fatourou et al. [2019] described an extension to a non-blocking binary search tree [Ellen et al. 2010] that gives it support for linearizable range queries. This extension is similar to the method by Arbel-Raviv and Brown [2018] in that range queries need to increment a global counter. On the other hand, their extension differs from the method by Arbel-Raviv and Brown in that it does not piggyback on the used memory reclamation technique, but instead lets update operations save links to nodes that have got spliced out from the tree in the nodes that replace the spliced out nodes.

All the non-adaptive set data structures with support for efficient range queries and scalable updates proposed in the literature [Agarwal et al. 2017; Arbel-Raviv and Brown 2018; Avni et al. 2013; Basin et al. 2017; Brown and Avni 2012; Chatterjee 2017; Fatourou et al. 2019; Robertson 2014] have range queries with a conflict time that depends at least linearly on the number of items covered by the range given to the range query. The CA trees (both the lock-based and the lock-free) can do much better over a wide range of scenarios as the conflict time of their range queries can be constant (i.e., independent of the number of items covered by the range query) and their heuristics work towards getting a good trade-off between range queries conflict time and the scalability of updates.

Even though the fundamental ideas behind the lock-based CA tree of Sagonas and Winblad [2017] and the lock-free CA tree are the same, lock-freedom gives LFCA trees better progress guarantees that are of importance for real-time systems. As we will see, the lookup operation of LFCA trees is wait-free and performs efficiently regardless of how contended the data structure is, which is crucial for many applications as lookups often dominate the workload. Still, the lock-based CA tree has a few advantages over the lock-free CA tree. The use of locks makes it possible to use mutable sequential data structures to store the items. This can be advantageous in systems where memory management is expensive and when range queries are infrequent, as it reduces the

number of memory allocations that are needed. The use of locks also makes it easier to extend the interface of the data structure with more linearizable operations.

The technique for joining base nodes in LFCA trees has some similarities with the replace operation of the non-blocking Patricia tries of Shafiei [2013]. The replace operation, in a non-blocking Patricia trie, deletes an item and adds another item in a way that appears atomic.

Several works have previously explored the idea of dynamically switching between a data structure that uses coarse-grained synchronization to a data structure that uses fine-grained synchronization in one transformation step [Chen et al. 2017; Newton et al. 2015; Österlund and Löwe 2014]. The drawback of the global mode switching approach proposed in these papers compared to LFCA tree's approach is that the switch between the modes is time-consuming and coarse-grained, whereas the LFCA tree can smoothly transition between different levels of synchronization granularity. Work has also been done to adapt to contention in other types of data structures; for example by Ha et al. [2007].

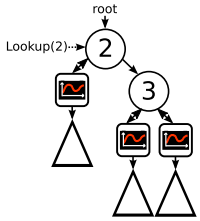
4 ALGORITHM

Pseudocode for all the non-trivial parts of the lock-free CA tree can be found in Figs. 4 to 6 and 8. The pseudocode is derived from a model of the LFCA tree implemented in the C programming language with some minor adjustments for readability. This section contains a detailed description of the algorithm and its pseudocode. In the next section, a detailed proof sketch will be given, showing that the operations are linearizable and have the stated progress guarantees.

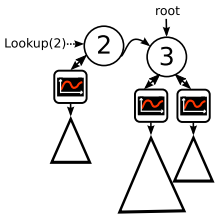
Node types. An LFCA tree is built from instances of the node types that are defined in Fig. 4, lines 15–53. Note that the keyword `with_fields_from` (on lines 27, 33, and 40) is used to add fields from another structure definition. All route nodes are of type `route_node` (lines 15–21). The route nodes contain a key field (line 16) which is used to direct searches for a specific item in the tree. Together, they form a binary search tree. Leaves are called *base nodes* (lines 22–43) and have a `data` field (line 23) that points to immutable data structure instances, called *leaf containers*, that contain the items that are in the represented set. We use an immutable treap for leaf containers in the pseudocode, but this can of course be replaced by any other immutable data structure that supports the same operations. That a base node B is of type `normal_base` (lines 22–26) indicates that B is not involved in an ongoing operation. A base node of type `range` is a node that currently is or has been involved in a range query. Similarly, a base node of type `join_main` or `join_neighbor` is a node that currently is or has been involved in a join operation. The LFCA tree structure itself is called `lfcatree` (lines 50–53) and only contains two fields. The root field of the `lfcatree` structure points to the root of the LFCA tree. Initially, this is a node of the `normal_base` type that is containing an empty leaf container. The `lfcatree` structure's field `no_join_count` is used for a trick that gives the lookup operation the wait-freedom property. This trick is explained where we describe the lookup operation later in this section.

Data in nodes that can be modified by more than one thread are marked with the modifier `atomic`. These fields can only be accessed by the atomic and sequentially consistent functions `aload` (that loads the value at a given address), `astore` (that stores the given value at the given address), `atomic_add` (that increments the value at the given address), `atomic_sub` (that decrements the value at the given address), and `CAS` (a compare-and-swap that stores the value of its third parameter at the location of a given address in its first parameter iff the value at the given address is equal to the second parameter and returns true, or returns false otherwise). All these functions have corresponding hardware instructions on modern multicores.

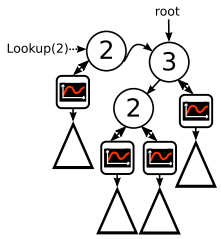
Lookup. The lookup operation (lines 134–137) calls the function `find_base_node` (lines 287–308), which traverses the route nodes using binary search until a base node is found, and then performs



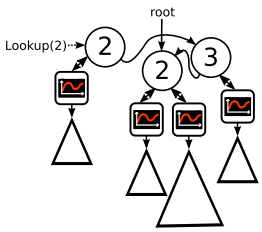
(a) A lookup operation has reached the route node with key 2.



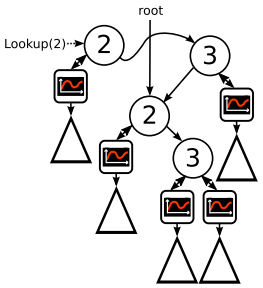
(b) A join has spliced out the route node with key 2.



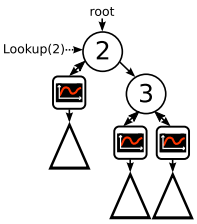
(c) A split has added a route node with key 2.



(d) A join has spliced out the route node with key 3.



(e) A split has added a route node with key 3.



(f) The lookup operation has traversed two more pointers and reached a route node with key 2 again!

Fig. 3. A sequence of changes to an LFCA tree showing why the lookup operation would not be wait-free if the lookup operation did not temporarily disable joins after traversing `NO_JOIN_LIMIT` route nodes. Notice that the sequence of events illustrated by Figs. 3a to 3f could repeat forever if joins were never disabled. This shows that the lookup operation would not be wait-free if the lookup never disabled joins.

the lookup in the corresponding immutable data structure. To ensure that the lookup operation is wait-free, we let the `find_base_node` function atomically increment the `no_join_count` field of the LFCA tree by one (in line 299) if and when it has traversed exactly `NO_JOIN_LIMIT` nodes. In calls where the `no_join_count` field is incremented, it is also atomically decremented by one once a base node has been found (line 305). This temporally disables joins of base nodes (see line 268) in extreme cases when lookups need to traverse a large number of nodes. Without such code for temporally disabling joins, it could be possible for joins and splits of base nodes in the tree to starve a lookup operation (such starvation is exemplified in Fig. 3) which would disqualify the lookup operation from being wait-free.[4]

The observant reader might wonder what will happen if a lookup thread goes to sleep indefinitely between lines 299 and 305 so that joins are disabled indefinitely. First of all, none of the theoretical properties that we claim for the LFCA tree depend on that joins of base nodes can start. (This is made clear in Section 5.) Secondly, we note that the scenario illustrated by Fig. 3 is unlikely to

[4]The code presented in the SPAA 2018 version of this article never disables joins. As a consequence, the lookup operation presented in that paper [Winblad et al. 2018] is not wait-free.

happen in practice. Thirdly, the likelihood that joins will be disabled (by executing line 299) can be made extremely low by setting the constant NO_JOIN_LIMIT to an even larger value. Thus, the fact that joins may be disabled by a lookup that goes to sleep has very little effect in practice (given that NO_JOIN_LIMIT is set to a high enough value), both for the adaptiveness of the data structure and the progress of lookup operations. There is however a trade-off between having the code that temporally disables joins and not having this code. Having this code makes the data structure less adaptive, but makes the lookup operation wait-free. Without this code, the lookup operation would only be lock-free, but the data structure would be more adaptive. We have opted for having the code that temporary disables joins in the presentation of the LFCA tree as it gives the lookup operation the wait-free property and has very little negative impact to the adaptiveness in practice. However, we note that implementations may omit the code for temporally disabling joins without observing much difference in practice[5].

Insert and Remove. The functions for inserting and removing a single item (Fig. 5, lines 128–133) call the do_update function (lines 106–126) with the LFCA tree, the remove or insert function for the immutable data structure that is used for the leaf containers, and the item to insert or remove as arguments. The functions treap_insert and treap_remove that are passed to the do_update function on lines 129 and 132 return a new leaf container reflecting the insert or remove respectively. Obviously, they do not modify the input leaf container (the first argument) as it is an immutable data structure. Their third parameter is a write back boolean value reflecting if the update was successful or not (i.e., in the case of insert that the value is not present in the input and in the case of remove if the value exists in the input). The second argument of treap_insert and treap_remove is the value to be inserted or removed (an int in our pseudocode).

The do_update function repeatedly searches for a base node using the given key, and then tries to replace that base node, if it is replaceable, with a new base node in which the key has been removed or inserted (lines 108–125). Line 109 finds the base node base using binary search for the key. A replacement attempt is made only if the found base node is replaceable (line 110 and lines 64–73). The is_replaceable function (lines 64–73) returns true if its input is a base node that is guaranteed to not be involved in an ongoing join (see the description of low-contention adaption below) or in an ongoing range query (see the description of range queries below). Essentially, is_replaceable returns true only if it is of type normal, if it is of one of the join types and comes from a completed or failed join, or if it is of the range type and the corresponding range query has completed. Such nodes cannot be changed and they can only be linked out from the tree by a CAS operation. If the base node is irreplaceable, then it may be involved in another operation; in this case the do_update function will first attempt to help this operation in the help_if_needed call (line 124 and lines 75–87) before retrying. (The help_if_needed function will be described in more detail later in this section.)

The replacement is done on line 118 using the try_replace function (lines 55–63), which uses a CAS to attempt to change the pointer of the base node's parent to a new base node with the updated leaf container. If successful, the CAS operation is the linearization point of the operation; if unsuccessful, the found base node has already been linked out from the tree by some other operation, and the whole operation is retried.

Note that the new base node gets a value for its stat field that is based on the replaced node's stat field and type as well as on whether conflicting operations have been detected (i.e., a base node which was not replaceable has been found or a try_replace call has failed; see line 116 and the new_stat function on lines 88–98). The new_stat function will be described in more

[5]When we compared experimental results of our LFCA tree with and without the functionality to temporarily disable joins, we did not observe any performance difference.

```
// Constants
#define CONT_CONTRIB     250      // Added to stat counter when base is contended
#define LOW_CONT_CONTRIB 1        // Added to stat counter when base is uncontended
#define RANGE_CONTRIB    100      // Added to stat when part of query with > 1 bases
#define HIGH_CONT        1000     // Limit for doing high-contention split
#define LOW_CONT         -1000    // Limit for doing low-contention join
#define NOT_FOUND        (node*)1 // Value representing that a node was not found
#define NOT_SET          (treap*)1// Indicates that a result storage is not set
#define PREPARING        (node*)0 // Represents that a join is ongoing
#define DONE             (node*)1 // Represents that a join is completed
#define ABORTED          (node*)2 // Represents that a join is aborted
#define NO_JOIN_LIMIT    500      // Threshold to ensure lookups are wait-free
enum contention_info { contended, uncontended, noinfo }
// Data Structures
struct route_node {
  int key;                       // Split key
  atomic node* left;             // < key
  atomic node* right;            // >= key
  atomic bool valid = true;      // Used for join
  atomic node* join_id = NULL;   // ...
}
struct normal_base {
  treap* data = empty_treap();   // Items in the set
  int stat = 0;                  // Statistics variable
  node* parent = NULL;           // Parent node or NULL (root)
}
struct join_main with_fields_from normal_base {
  node* neigh1;  // First (not joined) neighbor base
  atomic node* neigh2 = PREPARING; // Field for state of join and joined neighbor
  node* gparent; // Grand parent
  node* otherb;  // Other branch
}
struct join_neighbor with_fields_from normal_base {
  node* main_node  // The main node for the join
}
struct rs { // Result storage for range queries
  atomic treap* result = NOT_SET; // Set after the linearization of range query
  atomic bool more_than_one_base = false; // Helps adaptation know if > 1 bases
}
struct range_base with_fields_from normal_base {
  int lo; int hi; // Low and high key
  rs* storage;    // The result storage that is set after completion
}
enum node_type {
  route, normal, join_main, join_neighbor, range
}
struct node with_fields_from normal_base, range_base, join_main, join_neighbor {
  node_type type;
}
struct lfcatree { // The LFCA tree data structure
  atomic node* root = new node {type = normal_base};
  atomic uint no_join_count = 0;
}
// Help functions
bool try_replace(lfcatree* t, node* b, node* new_b) { // Try CAS b with new_b
  if ( b->parent == NULL )
    return CAS(&t->root, b, new_b); // Parent is root
  else if (aload(&b->parent->left) == b)
    return CAS(&b->parent->left, b, new_b); // b is left child of parent
  else if (aload(&b->parent->right) == b)
    return CAS(&b->parent->right, b, new_b); // b is right child of parent
  else return false;
}
bool is_replaceable(node* n) { // Check if it is allowed to replace base n
  return (n->type == normal || // Yes, if n is a normal base node
          (n->type == join_main && // Yes, if n is aborted join_main node
           aload(&n->neigh2) == ABORTED) ||
          (n->type == join_neighbor && // Yes, if aborted or completed join_neighbor
           (aload(&n->main_node->neigh2) == ABORTED ||
            aload(&n->main_node->neigh2) == DONE)) ||
          (n->type == range && // Yes, if range node belonging to a completed query
           aload(&n->storage->result) != NOT_SET));
}
```

Fig. 4. Data structures and help functions.

```
74 // Help functions
75 void help_if_needed(lfcatree* t, node* n) { // Help op make n replaceable if needed
76   if (n->type == join_neighbor) n = n->main_node; // main node describes the join
77   if (n->type == join_main &&
78       aload(&n->neigh2) == PREPARING) {
79     CAS(&n->neigh2, PREPARING, ABORTED); // Join is preparing so try to abort join
80   } else if (n->type == join_main &&
81              aload(&n->neigh2) > ABORTED) {
82     complete_join(t, n); // Join finished preparation so help complete join
83   } else if (n->type == range &&
84              aload(&n->storage->result) == NOT_SET) {
85     all_in_range(t, n->lo, n->hi, n->storage); // Help unfinished range query
86   }
87 }
88 int new_stat(node* n, contention_info info) { // Compute statistics value for n
89   int range_sub = 0; // Subtract to reduce chance of queries that need > 1 base node
90   if (n->type == range &&
91       aload(&n->storage->more_than_one_base))
92     range_sub = RANGE_CONTRIB; // Belonged to a query that needed > 1 base node
93   if (info == contended && n->stat <= HIGH_CONT) {
94     return n->stat + CONT_CONTRIB - range_sub; // Add if contended
95   } else if (info == uncontended && n->stat >= LOW_CONT) {
96     return n->stat - LOW_CONT_CONTRIB - range_sub; // Subtract if uncontended
97   } else return n->stat - range_sub;
98 }
99 void adapt_if_needed(lfcatree* t, node* b) {
100   if (!is_replaceable(b)) return; // b is irreplaceable: abort
101   else if (new_stat(b, noinfo) > HIGH_CONT)
102     high_contention_adaptation(t, b); // Reached high-contention adaptation limit
103   else if (new_stat(b, noinfo) < LOW_CONT)
104     low_contention_adaptation(t, b); // Reached low-contention adaptation limit
105 }
106 bool do_update(lfcatree* t, treap*(*u)(treap*,int,bool*), int i) {
107   contention_info cont_info = uncontended;
108   while (true) { // Loop until the update is completed
109     node* base = find_base_node(t, i); // Find the correct base node for i
110     if (is_replaceable(base)) { // Attempt to replace base, if base is replaceable
111       bool res;
112       node* newb = new node { // Create newb to replace base
113         type   = normal,
114         parent = base->parent,
115         data   = u(base->data, i, &res), // Apply the update operation u
116         stat   = new_stat(base, cont_info) // Use new_stat to get updated stat value
117       }
118       if (try_replace(t, base, newb)) { // Attempt to replace base with newb using CAS
119         adapt_if_needed(t, newb); // Adapt if heuristics suggests this is beneficial
120         return res; // Return value indicating if i existed or not
121       }
122     }
123     cont_info = contended; // Record that contention has been seen
124     help_if_needed(t, base); // Help make base replaceable if needed
125   }
126 }
127 // Public interface
128 bool insert(lfcatree* t, int i) {
129   return do_update(t, &treap_insert, i); // treap_insert = immutable treap insert
130 }
131 bool remove(lfcatree* t, int i) {
132   return do_update(t, &treap_remove, i); // treap_remove = immutable treap remove
133 }
134 bool lookup(lfcatree* t, int i) {
135   node* base = find_base_node(t, i);
136   return treap_lookup(base->data, i);
137 }
138 void query(lfcatree* t, int lo, int hi,
139            void (*trav)(int, void*), void* aux) {
140   treap* result = all_in_range(t, lo, hi, NULL); // Get snapshot of items in range
141   treap_query(result, lo, hi, trav, aux);  // Call trav(i, aux) for all i in range
142 }
```

Fig. 5. Help functions and public interface.

detail later in this section when we describe adaptations. Once a base node has been successfully replaced, the update operation calls the function `adapt_if_needed` to adapt the granularity of the data structure if the heuristics suggests that this is beneficial (line 119 and lines 99–105), before returning. Adaptations and the `adapt_if_needed` function (lines 99–105) will be described in detail after we have described range queries.

Range query. The range query operation (lines 138–142) first calls `all_in_range` (line 140) to create a snapshot of all base nodes in the requested range and then traverses the snapshot to complete the range query (line 141). We focus on the `all_in_range` function (lines 161–214) that does most of the range query operation's work.

First, note that the `all_in_range` function is not called only by the range query operation. It can also be called by operations that need to make base nodes which are made irreplaceable by an ongoing range query replaceable; see `help_if_needed` (lines 75–87). The parameter called `help_s` (line 161) is NULL when called by a range query operation. Otherwise, it is the result storage structure to which a helping thread should write the result of the range query, if the helper completes the range query. There is only one result storage for a particular range query. The result storage is also used to check if a base node of type `range` belongs to this range query (line 191), and to check if another thread has completed the range query (line 189). We will first give a high-level description of the `all_in_range` function, and then describe the traversal in more detail.

Function `all_in_range` (lines 161–214) goes through all base nodes that may contain items in the range in ascending key order, to replace them, using a CAS, by base nodes of type `range` (thus making them temporally irreplaceable). This traversal of all the base nodes in the range is done on lines 167 to 206. The `all_in_range` function then collects all the items in the traversed base nodes into a new data structure (lines 207–208). This resulting data structure is then written to the result storage, if this has not been done by any helping thread already (line 209).

A range query's linearization point is the execution step when the `result` field of its result storage is set in a successful CAS on line 210. Just before the replacement happens, all concerned base nodes have been replaced and are thus irreplaceable. Directly after the result storage has been set, these base nodes become replaceable, as a `range` node is only irreplaceable if its `result` field is not set (line 72).

A special field in the result storage associated with the range query is set to a value indicating if more than one base node were needed to complete the range query on line 210. This information is used by the `new_stat` function (lines 88–98) when calculating a statistics value for a base node. Finally, before `all_in_range` returns, the `adapt_if_needed` function is called on a random base node in the range to adapt the structure of the tree, if the heuristics suggests that this is beneficial (line 211).

Let us now discuss the code that finds and replaces the base nodes that may contain keys between `lo` and `hi` in more detail (lines 167–206). The base nodes are found by an in-order traversal of the concerned portion of the tree. The first base node in the range is found on line 167. The `find_base_stack` function that is called on line 167 also populates the search stack. We then check if this call to `all_in_range` is to help a range query complete (line 168). If this is the case and the base node that we found does not belong to this range query, we know that the range query has already completed, so we can return on line 170. Otherwise, we set the result storage for the call to the one given as a parameter on line 171. When we are not helping an already started range query, the first base node in the range needs to be replaced and we need to create a new storage (lines 172–176). When the base node that we are trying to replace is already removed from the tree, we have to try again (line 175). An optimization that we have applied can be seen on line 178. It makes use of the fact that we can piggyback on the first base node's range query if this base node

```
143 node* find_next_base_stack(stack* s) {
144   node* base = pop(s);
145   node* n = top(s);
146   if (n == NULL) return NULL;
147   if (aload(&n->left) == base)
148     return leftmost_and_stack(aload(&n->right), s);
149   int be_greater_than = n->key;
150   while (n != NULL) {
151     if (aload(&n->valid) && n->key > be_greater_than)// Skip invalid route nodes
152       return leftmost_and_stack(aload(&n->right), s);
153     else { pop(s); n = top(s); }
154   }
155   return NULL;
156 }
157 node* new_range_base(node* b, int lo, int hi, rs* s) {
158   return new node {... = b, // Assign fields from b
159                    type = range_base, lo = lo, hi = hi, storage = s};
160 }
161 treap* all_in_range(lfcatree* t, int lo, int hi, rs* help_s) {
162   stack* s = new_stack();
163   stack* backup_s = new_stack();
164   stack* done = new_stack();
165   node* b;
166   rs* my_s;
167  find_first:b = find_base_stack(aload(&t->root),lo,s);// First base in range
168   if (help_s != NULL) { // Are we helping?
169     if (b->type != range || help_s != b->storage) {
170       return aload(&help_s->result); // Helping and b does not belong to query
171     } else my_s = help_s; // my_s is the storage for the query we are helping
172   } else if (is_replaceable(b)) { // We can try to replace b
173     my_s = new rs;
174     node* n = new_range_base(b, lo, hi, my_s);
175     if (!try_replace(t, b, n)) goto find_first; // Try again if unsuccessful
176     replace_top(s, n); // Replacement successful (change top of stack)
177   } else if ( b->type == range && b->hi >= hi) {
178     return all_in_range(t, b->lo, b->hi, b->storage); // Piggyback
179   } else {
180     help_if_needed(t, b); // Help the operation that has made b irreplaceable
181     goto find_first; // Try to find first base node in range again
182   }
183   while (true) { // Find remaining base nodes
184     push(done, b); // Add current b to done
185     copy_state_to(s, backup_s); // Backup s in case we have to retry
186     if (!empty(b->data) && max(b->data) >= hi) break; // Done
187   find_next_base_node: b = find_next_base_stack(s); // Find next base node
188     if (b == NULL) break; // Done, end of tree
189     else if (aload(&my_s->result) != NOT_SET) {
190       return aload(&my_s->result); // Done, we have been helped
191     } else if (b->type == range && b->storage == my_s) {
192       continue; // b belongs to this query (another thread helped)
193     } else if (is_replaceable(b)) {
194       node* n = new_range_base(b, lo, hi, my_s);
195       if (try_replace(t, b, n)) { // Try to replace b with irreplaceable node n
196         replace_top(s, n); continue; // Replacement successful
197       } else {
198         copy_state_to(backup_s, s);
199         goto find_next_base_node; // Try again
200       }
201     } else {
202       help_if_needed(t, b); // Help the operation that has made b irreplaceable
203       copy_state_to(backup_s, s);
204       goto find_next_base_node; // Try again
205     }
206   }
207   treap* res = done->array[0]->data;
208   for (int i = 1; i < done->size; i++) res = treap_join(res, done->array[i]->data);
209   if (CAS(&my_s->result, NOT_SET, res)) { // Set result storage
210     if (done->size > 1) astore(&my_s->more_than_one_base, true); // For heuristics
211     adapt_if_needed(t, done->array[thread_local_random_int() % done->size]);
212   }
213   return aload(&my_s->result);
214 }
```

Fig. 6. Helper function for the range query operation.

belongs to an uncompleted range query that has a maximum key which is larger or equal to `hi`. We help the operation that is working with `b` and then try to find the first base node in the range again, if `b` is irreplaceable (line 180). Lines 183–206 repeat a similar process as we have done for the first base node in the range until all base nodes in the range are covered. Notice that we are taking a backup of the current search stack before we try to find the next base node (line 185). This is done to make it easier to recover the old stack when we need to retry the search for the next base node.

Adaptations. The granularity of the immutable parts of a LFCA tree can be changed with two different types of adaptations. The first one, called *high-contention adaptation* (or *split*), splits the items in a base node into two new base nodes in order to decrease the contention in a part of the tree where the contention has been high. The second type, called *low-contention adaptation* (or *join*), joins the contents of two base nodes into a new base node, in order to improve the performance of range queries. Joins can potentially also improve the performance of the LFCA tree for uncontended single-item operations as a join can make the search paths to items shorter (because the part of the tree consisting of route nodes may be unbalanced), but joins may also make updates slightly more expensive (due to increased amount of memory allocation and copying when creating new instances of the leaf containers). An adaptation is initiated by the function `adapt_if_needed` (lines 99–105) that is executed by the update operations (line 119) and by range queries (line 211). Whether an adaptation should occur and what kind of adaptation it should be is decided based on a statistics value calculated by the `new_stat` function (lines 88–98). High-contention or low-contention adaptation is initiated if this statistics value is above (line 101) or below (line 103) a threshold, respectively. The `new_stat` function calculates the statistics value based on its two parameters: a base node and a parameter that is encoding information about detected contention. The core idea behind the heuristics is to make the synchronization more fine-grained in parts of the data structure where contention has been common and to make it more coarse-grained in parts where contention has been uncommon or where range queries often need to access more than one base node.

If no contention information is given to the `new_stat` function (as is the case when this function is called by `adapt_if_needed`), then the statistics value is the value of the `stat` field in the base node subtracted by x, where x is a positive constant if it is a base node of type `range` whose corresponding range query was completed by reading more than one base node (lines 90–92). When update operations call `new_stat` to get the value for the `stat` field of the new base node they create, they also pass information whether contention was detected (cf. line 116). This information is used to increase the statistics value if contention has been detected (line 94), and decrease the statistics value otherwise (line 96). The constant that is used to increase the statistics value when contention is detected is larger than the constant that decreases the statistics value when no contention has been detected, so that adaptations happen quickly when contention is common and also to avoid frequent adaptations back-and-forth. The constants used in our heuristics can be found in lines 2 to 6.

High-contention adaptation. The function for high-contention adaptation (lines 276–286) splits the content of a base node `b` into two new base nodes that are linked together with a route node `r`. The function attempts to replace the base node `b` with `r` using a `CAS` operation (line 285). This replacement is atomic to other operations and does not change the contents of the tree.

Low-contention adaptation. The function for low-contention adaptation (lines 266–275) intuitively replaces two neighboring base nodes `b` and `n0` by a new node `n2`, which contains the union of the items in `b` and `n0`. It splices out `b` and its parent route node from the tree and replaces `n0` by `n2`. Figure 7 illustrates the main steps of a successful low-contention adaptation.

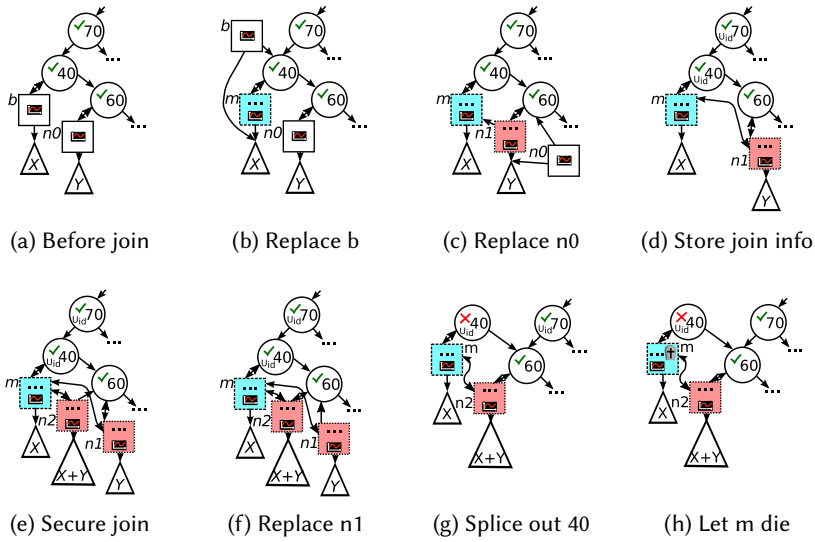


Fig. 7. Illustration of the main steps of the low-contention join operation. A route node with a green tick mark (✓) has its valid field set to true and a route node with a red mark (✗) has its valid field set to false.

A successful low-contention adaptation (lines 266–275) is done in two phases. The first phase, illustrated in Figs. 7a to 7e and performed by `secure_join_left` in lines 215–248 (or the corresponding function for the right case), "marks" the part of the tree that will be involved in the join, to prevent other threads from changing it while the join is ongoing. Other threads cannot help complete this phase, but system-wide-progress is guaranteed as other threads can interrupt this phase by killing the join (line 79). The second phase (Figs. 7f to 7h), which is performed by function `complete_join` (lines 249–265), is executed only if the first phase was successful (line 271). Other threads can help the join to complete this second phase (line 82). The second phase completes the join by splicing out `b` and its parent and replaces `n1` by `n2`.

We will describe how the low-contention adaptation works by going through a successful join of the base node `b` in Fig. 7a. As `b` is the left child of its parent, `secure_join_left` is called (line 269). Lines 216–221 find the neighbor `n0` of `b` (the leftmost leaf of `b`'s parent's right branch) and replace `b` with a new base node `m` of type `join_main` (Fig. 7b). Note that the join would have aborted if `n0` would have been irreplaceable or if the replacement of `b` would have failed (which would have meant that `b` was no longer in the tree). Next, `n0` is replaced with a new base node `n1` of type `join_neighbor`, which is linked to `m` with the field `main_node` (line 226 and Fig. 7c). Both `m` and `n1` are now irreplaceable (lines 64–73). The only way for other threads to make them replaceable at this point is to set the field `neigh2` of `m` to `ABORTED`. On lines 227–231, the `join_id` field of both the parent and the grandparent of `m` is set to the reference `m` to make sure that they are not modified by any other join operation. Using the reference `m`, which is a unique identifier for the join operation, to mark the route nodes involved in the join makes it easy for threads to collaboratively change this field in the second phase. In lines 232–234, more information that function `complete_join` needs to finish the operation is stored in `m` (cf. Fig. 7d). The final step of the first phase is done in lines 235-241 (cf. Fig. 7e). These lines attempt to set `m`'s field `neigh2` to a base node `n2` which can replace both `m` and `n1` using a `CAS` operation. If this `CAS` is successful, we know that the following are true directly after the change:

```
215 node* secure_join_left(lfcatree* t, node* b) { // secure_join_right is symmetric
216   node* n0 = leftmost(aload(&b->parent->right)); // leftmost base of parent->right
217   if (!is_replaceable(n0)) return NULL; // Give up if n0 is irreplaceable
218   node* m = new node { // Create join_main base
219     ...  = b, // assign fields from b
220     type = join_main};
221   if (!CAS(&b->parent->left, b, m)) return NULL; // Give up if CAS fails
222   node* n1 = new node { // Create join_neighbor base
223     ...       = n0, // assign fields from n0
224     type      = join_neighbor,
225     main_node = m};
226   if (!try_replace(t, n0, n1)) goto fail0; // Abort and give up if CAS fails
227   if (!CAS(&m->parent->join_id, NULL, m)) goto fail0; // " " " " " " "
228   node* gparent = parent_of(t, m->parent); // grand parent of m
229   if (gparent == NOT_FOUND ||
230       (gparent != NULL && // Abort and give up if CAS on next line fails
231        !CAS(&gparent->join_id,NULL,m))) goto fail1;
232   m->gparent = gparent; // Store info about join op in m
233   m->otherb = aload(&m->parent->right);
234   m->neigh1 = n1;
235   node* joinedp = m->otherb == n1 ? gparent: n1->parent; // Parent of n2 (Fig. 7)
236   if (CAS(&m->neigh2, PREPARING,
237           new node {... = n1, // assign fields from n1
238                     type = join_neighbor,
239                     parent = joinedp,
240                     main_node = m,
241                     data = treap_join(m, n1)}))
242     return m; // Join secured (complete_join will complete join)
243   if (gparent == NULL) goto fail1; // Abort and give up
244   astore(&gparent->join_id, NULL);
245  fail1: astore(&m->parent->join_id, NULL);
246  fail0: astore(&m->neigh2, ABORTED);
247   return NULL;
248 }
249 void complete_join(lfcatree* t, node* m) { // Complete secured join
250   node* n2 = aload(&m->neigh2);
251   if (n2 == DONE) return; // Another thread has helped
252   try_replace(t, m->neigh1, n2);
253   astore(&m->parent->valid, false); // Mark m->parent invalid (will/has spliced out)
254   node* replacement =
255     m->otherb == m->neigh1 ? n2 : m->otherb;
256   if (m->gparent == NULL) { // The following lines splices out m->parent
257     CAS(&t->root, m->parent, replacement);
258   } else if (aload(&m->gparent->left) == m->parent) {
259     CAS(&m->gparent->left, m->parent, replacement);
260     CAS(&m->gparent->join_id, m, NULL);
261   } else if (aload(&m->gparent->right) == m->parent) {
262     ... // Symmetric case
263   }
264   astore(&m->neigh2, DONE); // Mark join as completed
265 }
266 void low_contention_adaptation(lfcatree* t, node* b) {
267   if (b->parent == NULL ||
268       aload(&t->no_join_count) > 0) return; // Trick to make lookup wait-free
269   if (aload(&b->parent->left) == b) {
270     node* m = secure_join_left(t, b); // Try to make sure join will happen
271     if (m != NULL) complete_join(t, m); // Complete join if secured successfully
272   } else if (aload(&b->parent->right) == b) {
273       ... // Symmetric case
274   }
275 }
276 void high_contention_adaptation(lfcatree* t, node* b) {
277   if (less_than_two_items(b->data)) return; // Do not split if b has < 2 items
278   node* r = new node { // Create route node that will replace base node b
279     type  = route,
280     key   = split_key(b->data),
281     left  = new node{type = normal, parent= r, stat= 0,
282                      data = split_left(b->data)}),
283     right = ..., // Symmetric case
284     valid = true};
285   try_replace(t, b, r); // Replace b with the new route node
286 }
```

Fig. 8. Low and high contention adaptation.

```
node* find_base_node(lfcatree* t, int i) { // Find base node with i in its range
  uint traversed_nodes = 0;
  node* n = aload(&t->root);
  bool no_join_reached = false;
  while (n->type == route) { // Binary search in t to find base node with i in range
    if (i < n->key) {
      n = aload(&n->left);
    } else {
      n = aload(&n->right);
    }
    traversed_nodes++;
    if (traversed_nodes == NO_JOIN_LIMIT) { // For trick to make lookup wait-free
      atomic_add(&t->no_join_count, 1);
      no_join_reached = true;
      n = aload(&t->root);
    }
  }
  if (no_join_reached) { // For trick to make lookup wait-free
    atomic_sub(&t->no_join_count, 1);
  }
  return n;
}
node* find_base_stack(node* n, int i, stack* s) { // For all_in_range
  stack_reset(s); // Similar to find_base_node but records search path in s
  while (n->type == route) {
    push(s, n);
    if (i < n->key) {
      n = aload(&n->left);
    } else {
      n = aload(&n->right);
    }
  }
  push(s, n);
  return n;
}
node* leftmost_and_stack(node* n, stack* s) { // For all_in_range
  while (n->type == route) { // Find leftmost base in n and record search path in s
    push(s, n);
    n = aload(&n->left);
  }
  push(s, n);
  return n;
}
node* parent_of(lfcatree* t, node* n) { // Parent of route node (secure_join_left)
  node* prev_node = NULL; // Return NULL if parent is the root
  node* curr_node = aload(&t->root);
  while (curr_node != n && curr_node->type == route) {
    prev_node = curr_node;
    if (n->key < curr_node->key) {
      curr_node = aload(&curr_node->left);
    } else {
      curr_node = aload(&curr_node->right);
    }
  }
  if (curr_node->type != route) {
    return NOT_FOUND; // Return NOT_FOUND if n is not in tree
  }
  return prev_node;
}
```

Fig. 9. Auxiliary code for the LFCA tree.

- Both `m` and `n1` must have been irreplaceable at the time of the change, as this means that the field `neigh2` has not been set to `ABORTED` by any other thread.
- The node referenced by `m`'s `parent` field is the parent of `m`, the node referenced by `m`'s `gparent` field is the grandparent of `m`, and `m`'s field `otherb` is set to the sibling of `m`. This follows from the observations that (i) the only type of change that can happen to a path from the root of the tree to an irreplaceable base node that is inside the tree is that a route node gets spliced out (which can only happen if both the spliced out node and its parent have their `join_id` fields set to something different from `NULL`), (ii) the `join_id` field of the spliced out node is

never set to NULL again, and (iii) no other threads can change the join_id field of parent and gparent while the neigh2 field of m is set to PREPARING.

Once the neigh2 field of m has been successfully set to the new base node, the first phase of the operation is finished, and the second phase can start. Note that the changes done by the operation would have been rolled back on lines 243–246 if some CAS operations had failed.

As mentioned, the second phase of a low-contention adaptation is done by complete_join (lines 249–265). Multiple threads can execute this function with the same base node of type join_main as input parameter. This happens when another thread needs to change a base node of type join_main or join_neighbor and the join associated with the base node has finished the first phase but not yet the second (which means that the base node is irreplaceable; see lines 64–73). That other thread will call the help_if_needed function, that in turn will call complete_join (line 82). The first modification that is done by the second phase is to replace the base node n1 with the base node n2 that is referenced to by the field neigh2 in m which was set in the first phase (cf. Fig. 7f). Note that any thread that executes complete_join with the base node m as parameter can perform this step as it is done with a CAS operation (line 252 and the try_replace function). The next change is to set the valid flag of the parent to false (line 253). The sole purpose of this is so that the range query operation can avoid traversing branches that are no longer relevant for the range query (line 151). On lines 254–255 it is determined what will be the replacement of m's parent. If m and n2 share the same parent (in this case b->otherb, which is set on line 233, will be equal to n1, a.k.a. b->neigh1) the replacement will be n2, otherwise the replacement will be the branch of the parent that does not lead to m. (This case is illustrated in Fig. 7f.) The parent of m and m itself are spliced out from the tree on lines 256–263 (cf. Fig. 7g). Note that only one thread can succeed with the splice out as it is done with a CAS operation. Likewise, only one thread can succeed in resetting the join_id of the grandparent of m (line 260). The only remaining step after these lines have been executed is to set the neigh2 field of m to DONE, which is done on line 264 (cf. Fig. 7h). This is done to indicate that the join has completed and to make n2 replaceable.

5 CORRECTNESS

In this section, we present a proof that the algorithm is correct, in the sense of being a linearizable implementation of a set of items which also supports range queries, and that it has the stated progress guarantees. Linearizability is considered in Section 5.1, and progress properties in Section 5.2.

5.1 Linearizability

Linearizability [Herlihy and Wing 1990] is a standard correctness criterion for concurrent data structure implementations. It states that each operation on a concurrent data structure can be considered as being performed atomically at some point, called the *linearization point*, between its invocation and return. Our proof of linearizability follows the following standard pattern.

We first define a function from global states of the data structure to abstract logical states. Each abstract logical state is a set of items, which is the client threads' view of the current contents of the data structure. Having been invoked, each operation performs a sequence of execution steps before it returns. For each such sequence of execution steps, we prove that

- one of the execution steps, called the linearization point, affects the abstract logical state according to the sequential semantics of the operation,
- the other execution steps (which are not at the linearization point) leave the abstract logical state unchanged, and
- the return value of the operation reflects the abstract logical state at its linearization point.

For instance, an execution of a lookup operation which returns `true` must contain a linearization point at which the item is contained in the abstract logical state.

To support the linearizability proof, we formulate a number of inductive invariants of the algorithm which can be used when establishing that the abstract logical state is affected in the appropriate way at linearization points, and is unchanged otherwise. These invariants must be inductive, i.e., we must prove that they are preserved by each execution step.

In this section, we first define the function from states of the data structure to abstract logical states (Section 5.1.1), thereafter we define supporting inductive invariants (Section 5.1.2), and finally we prove that each operation is linearizable (Section 5.1.3).

5.1.1 Definition of Abstract Logical States. Let us first define a function from global states of the algorithm to abstract logical states. This function must of course exploit the binary search tree structure of the LFCA tree. The route nodes in this binary search tree partitions the set of possible items (here the set of integers) so that each base node is responsible for an interval. First, define a route or base node to be *reachable from* n_s (in a given global state) if it can be reached from n_s by following `left` and `right` pointers. Next, define a function *relrange*$(\mathsf{n}_s, \mathsf{n}_t)$ from pairs of nodes n_s, n_t to intervals, in such a way that a search for an item `i`, starting from n_s will visit a node n_t in the search tree precisely when `i` is in *relrange*$(\mathsf{n}_s, \mathsf{n}_t)$ (provided that the tree remains unchanged during the traversal). The function *relrange* is defined as follows.

- *relrange*$(\mathsf{n}, \mathsf{n}) = [-\infty, \infty]$.
- If n_t is a route node, reachable from n_s, with *relrange*$(\mathsf{n}_s, \mathsf{n}_t) = [\mathtt{lo},\ \mathtt{hi}]$, then
  - *relrange*$(\mathsf{n}_s, \mathsf{n}_t\mathtt{->left}) = [\mathtt{lo},\ \min(\mathsf{n}_t\mathtt{->key} - 1,\ \mathtt{hi})]$, and
  - *relrange*$(\mathsf{n}_s, \mathsf{n}_t\mathtt{->right}) = [\max(\mathsf{n}_t\mathtt{->key},\ \mathtt{lo}),\ \mathtt{hi}]$.
- If n_t is not reachable from n_s, then *relrange*$(\mathsf{n}_s, \mathsf{n}_t) = \emptyset$.

Here $[\mathtt{lo},\ \mathtt{hi}]$ denotes the closed interval from `lo` to `hi`, which we also view as a set of integers. The min and max operations above are used for the case that the `key` value is outside $[\mathtt{lo},\ \mathtt{hi}]$. Note that we slightly abuse terminology by referring to `n` as a node rather than as a pointer to a node.

Define a route or base node to be *reachable* if it is reachable from the root of the tree. Define a base node `n`, to be *replaceable* if `is_replaceable(n)` returns `true` (lines 64–73). For a base node `n`, define *contents*(`n`) as the set of items in `n`'s leaf container. For a node `n`, define its *range*, denoted *range*(`n`), as *relrange*(`t->root`, `n`). For a given global state, we define the abstract state of each node `n` in the tree, denoted *abstrstate*(`n`), as follows.

- If `n` is a base node, then *abstrstate*(`n`) = *contents*(`n`) ∩ *range*(`n`).
- If `n` is a route node, then *abstrstate*(`n`) = *abstrstate*(`n->left`) ∪ *abstrstate*(`n->right`).

It follows that *abstrstate*(`n`) ⊆ *range*(`n`). This can be proven by induction over the tree structure of nodes, as follows.

- For base nodes `n`, it follows directly from the definition of *abstrstate*(`n`).
- For route nodes, it follows from the definition of *abstrstate*(`n`) for route nodes, and observing that by the second bullet in the definition of *relrange* we have *range*(`n`) = *range*(`n->left`) ∪ *range*(`n->right`).

We define the abstract state of an LFCA tree `t` as *abstrstate*(`t->root`), i.e., as the union of the abstract states of the reachable base nodes.

From the preceding definitions, we can now prove a property saying how the responsibility for storing items in the abstract state is partitioned among the nodes, namely: that for all items `i` and nodes `n`:

$$\mathtt{i} \in \mathit{range}(\mathtt{n}) \Rightarrow (\mathtt{i} \in \mathit{abstrstate}(\mathtt{t->root}) \Leftrightarrow \mathtt{i} \in \mathit{abstrstate}(\mathtt{n})) \qquad (\mathit{PartAbsState})$$

To prove this property, we first note that the corresponding property for adjacent nodes, i.e., for all items i and route nodes n:

$$\texttt{i} \in \mathit{range}(\texttt{n->left}) \Rightarrow (\texttt{i} \in \mathit{abstrstate}(\texttt{n}) \Leftrightarrow \texttt{i} \in \mathit{abstrstate}(\texttt{n->left}))$$

follows from the definition of $\mathit{abstrstate}(\texttt{n})$ for route nodes n together with the just established property $\mathit{abstrstate}(\texttt{n->left}) \subseteq \mathit{range}(\texttt{n->left})$. This property together with the corresponding property for n->right is then used to establish Property *PartAbsState* by induction over the length of the path from the root node to n.

5.1.2 Definition of Inductive Invariants. We next define a global invariant as the conjunction of six invariants (I1–I6), which together capture important properties of LFCA trees. The first four invariants (I1, I2, I3, and I4) state general properties about nodes. Invariant I5 formulates an essential property of the function find_base_node, while Invariant I6 formulates an essential property of range queries. The invariants are:

I1. If a route node n is unreachable, then its valid field is set to false, and its join_id field is set to the identity of a child base node of type join_main, whose field neigh2 points to a node of type join_neighbor.
I2. A base node n is reachable if it is obtained as either t->root, as n->parent->left, or as n->parent->right, and furthermore is not of type join_main with a neigh2 field pointing to a node of type join_neighbor and with a parent whose valid field is set to false.
I3. $\mathit{contents}(\texttt{n}) \subseteq \mathit{range}(\texttt{n})$ whenever n is a reachable base node that is not of type join_neighbor, nor obtained as n->main->neigh2.
I4. Whenever n is a reachable route node, then n->key is in $\mathit{range}(\texttt{n})$.
I5. During each execution of find_base_node, the value of its argument i and local node variable n, declared at line 289, satisfy the property that: if n is reachable then (i) $\texttt{i} \in \mathit{range}(\texttt{n})$, and (ii) for the unique base node $\texttt{n}_t$ with $\texttt{i} \in \mathit{relrange}(\texttt{n}, \texttt{n}_t)$, we have $\texttt{i} \in \mathit{range}(\texttt{n}_t)$.
I6. For each result storage structure my_s whose result field is NOT_SET, the base nodes of type range whose storage field is my_s are all reachable and have the same parameters lo and hi; moreover the union of their ranges forms a contiguous interval whose lower end is at most lo.

We comment that Invariant I2 is central for establishing correctness of many operations that update the abstract state of the tree. For instance, an insert operation inserts a new element by replacing a base node n. The operation succeeds only if n is replaceable, and is either obtained as t->root, as n->parent->left, or as n->parent->right. By Invariant I2, n is then reachable, whence by Invariant I5 we have $\texttt{i} \in \mathit{range}(\texttt{n})$. Hence, Property *PartAbsState* ensures that the inserted element will correctly enter the abstract state of the tree.

The proof of the six invariants involves to check that they are preserved by each execution step of the algorithm. To support this check, we state and prove general properties that constrain how the global state can be modified by any execution step. These properties, which will be used both when establishing the above invariants and when proving each operation linearizable, are as follows.

P1. A replaceable base node cannot become irreplaceable (but can be replaced by an irreplaceable node).
P2. For a reachable route or base node n, $\mathit{range}(\texttt{n})$ never decreases as long as n is reachable.
P3. An unreachable node cannot become reachable. Furthermore, no field or data in an unreachable node is ever changed.
P4. The pointer from a route node n1 to a route node n2 can change only if the valid field of n2 is false, and if both n1 and n2 have their join_id fields set to the identity of a child of n2

that is an irreplaceable base node of type `join_main`, whose field `neigh2` points to a node of type `join_neighbor`.

Let us provide the arguments why these properties hold.

P1: This property follows by inspecting the definition of function `is_replaceable` (lines 64–73), and noticing that no statement in the algorithm changes a field of a base node in such a way that makes a replaceable base become irreplaceable.

P2: This property follows by checking that the `key` field of a node never changes, and that the only statements that changes a `left` or `right` pointer to another node are either
- a CAS operation that replaces a replaceable base node (at lines 55–63); this leaves *range* unchanged for all nodes except the replaced one (which becomes unreachable), or
- the splicing out of a route node in the join operation (lines 256–263); this can increase *range* of some nodes, and will be considered below when analyzing the join operation.

P3: This property follows by checking that the only statements that change a `left` or `right` pointer to another node are either
- a CAS operation that replaces a replaceable base node by a fresh one, thus satisfying P3, or
- the splicing out operation of the join operation (lines 256–263); when analyzing the join operation below, we will show that this does not make any unreachable node reachable.

P4: The only change to a pointer to a route node is in the join operation, so we establish this property when analyzing the join operation.

We now establish the above six invariants jointly as follows.

I1: The invariant follows directly from Property P4, using Property P3.

I2: We first consider how a base node can enter the tree. Whenever a base node `n` is linked to a route node, this is done by replacing an existing base node using a CAS (at lines 55–63). We note that the replaced node cannot be of type `join_main` with a `neigh2` pointing to another node, so it was reachable; hence the replacing node must be reachable. We then consider how a base node `n` can be made unreachable. This can happen in two ways:
- It can be replaced by another node using a CAS (at lines 55–63), in which case it is no longer obtained as `t->root`, as `n->parent->left`, or as `n->parent->right`.
- Its parent route node can be made unreachable. By Invariant I1, the unreachable parent node has its `valid` field is set to `false`, and a child of type `join_main` with a `neigh2` field pointing to a node of type `join_neighbor`. From this, we infer that the child base node must be `n`, thus concluding the proof.

I3: Whenever the insert operation inserts a new item `i` into the leaf container of `n`, then `i` is in *range*(`n`): this is proven when analyzing the insert operation. Invariant I3 then follows from Property P2. Since *range*(`n`) can be changed by the join operation, we will show that join preserves Invariant I3 when analyzing low-contention adaptation (page 23).

I4: Whenever a new route node `r` is created in a high-contention adaptation operation, then the generated key is in *range*(`r`): this will be shown when analyzing high-contention adaptation (page 23). The invariant then follows using Property P2.

I5: We note that property (i) is trivially preserved by execution steps of `find_base_node` (where the crucial ones are the traversal steps in lines 293 and 295), and by execution steps of other operations by Property P2. Property (ii) follows from property (i) by using the definition of *relrange*, without any need for inspecting the algorithm.

I6: This will be established when analyzing range queries (page 22).

5.1.3 Establishing Linearizability of Operations. We now continue to prove linearizability of each operation.

Lookup. The lookup operation for item `i` uses `find_base_node` to locate a base node n_b and thereafter checks whether `i` is in n_b's leaf container. In the case that n_b is reachable when `find_base_node` traverses the link to n_b, then we take this traversal step (the execution of line 293 or 295 that reaches n_b) as the linearization point, since the leaf container that will be inspected by `treap_lookup` is the one associated with n_b. From Invariant I5, we infer $i \in range(n)$ at the linearization point, which by Property *PartAbsState* ensures that the correct result is returned. (Note that there can be many update operations that replace n_b after the linearization point and before the lookup operation returns.) In the case that n_b is not reachable when `find_base_node` traverses the link to n_b, we take as linearization point the step in the algorithm that caused the route node `n` which was then visited by `find_base_node` to become unreachable. (The only statements that can make a route node unreachable are at lines 257, 259 and 262.) By Property P3, any node visited by `find_base_node` after the linearization point does not change thereafter and, by Invariant I5 and Property *PartAbsState*, the lookup operation will find item `i` if and only if `i` is in the abstract state at the linearization point. (Note that, as will be proven when analyzing the join operation below, lines 257, 259 and 262 do not change the abstract state.) It follows that the return value of the lookup operation is consistent with the abstract state at the linearization point.

Insert and Remove. By the test at line 110, an update operation can only replace a base node `n` that is replaceable, using a CAS operation (in the `try_replace` function, line 55), which succeeds only if `n` is pointed to by `t->root`, or is obtained as `n->parent->left` or `n->parent->right`. By Invariant I2 and Properties P1 and P2, this ensures that the replaced node is reachable when the replacement is performed. Invariant I5 then implies that $i \in range(n)$ when the CAS operation is applied. We let the linearization point of an update operation coincide with the successful CAS (lines 55–63). Since the operation replaces `n`, Property *PartAbsState* implies that the abstract state is changed correctly at the linearization point. The property $i \in range(n)$, together with Property P2, also directly implies Invariant I3.

Range query. The `query` function (lines 138–142) first calls `all_in_range` (line 140) that first replaces all base nodes in its range by nodes of type `range`, and thereafter collects the items in their leaf containers into a new data structure (lines 207–208) which is then written to the result storage (line 209). The function `all_in_range` can also be called by helper threads, which refer to the range query by its uniquely defined result storage structure (created on line 173). Based on the created result storage, the query is answered to the calling thread (line 141).

We must first establish Invariant I6. To do so, consider a particular storage structure `my_s` whose result field is `NOT_SET`. We examine the execution steps that create or affect nodes of type `range`, and whose storage field is `my_s`.

- We first consider the replacement of an existing base node by a new one `n` of type `range` at line 175. This replacement is immediately preceded by the creation of a new result storage `my_s`, for which `n` is the first base node of type `range`. The replacement succeeds if the replaced base node `b` is reachable (this follows from Invariant I2) and if `lo` is in *range*(`b`) (this follows from the fact that `b` is found using the function `find_base_stack`, which can be proven to satisfy an invariant analogous to I5, using an analogous proof). Thus, the first created base node of type `range` conforms to Invariant I6.
- We then consider the replacement of an existing base node by a new one `n` of type `range`, whose `storage` is set to `my_s`, at line 195. This replacement succeeds if there is a preceding node `r` whose `storage` is also set to `my_s`. We first conclude that the location for `n` is found by the function `find_next_base_stack` by invoking `leftmost_and_stack` from the closest

ancestor of r that reaches r by first following its left field; this follows by inspection of find_next_base_stack, keeping in mind that
– by Invariant I4, the second conjunct on line 151 implies that we found an ancestor which reaches r through its left field,
– if the first conjunct on line 151 is false and the second is true, then by Invariant I1 the node n (defined on line 145) is not reachable, and its parent reaches r through its right pointer, hence it is correct to continue the search with the parent of n.

We then note that if the replacement at line 195 succeeds, then by Invariant I2 the node r is reachable. Furthermore, the just established properties of find_next_base_stack, together with Property P2, imply that the range of the new node is adjacent to *range*(r). This concludes the proof of Invariant I6.

We now note that done will contain precisely the replaced base nodes. We next note that line 209 is reached only if either a base node has been found which contains a data item larger than hi (line 186), or if no more base nodes with larger data items exist in the tree (line 188). In both cases, Invariant I6 implies that all reachable base nodes with items in the range of the query are of type range, with the field storage set to my_s. As linearization point we now take the successful CAS on line 210. Then, by Property *PartAbsState*, the range query properly reflects the abstract state at the linearization point, which implies that the range query operation is linearizable.

High-contention adaptation. This operation atomically replaces a base node by a route node n from which the same set of items can be reached. Thus, the operation does not change the abstract state. We must prove Invariant I4, i.e., that the new key is in *range*(n). For this, we rely on the assumption the the key returned by split_key (line 280) is in the range of items in the replaced leaf container, and hence in *range*(n) by Invariant I3. This establishes Invariant I4.

Low-contention adaptation. A successful join operation replaces two reachable base nodes by a new one whose leaf container contains the elements of the two replaced leaf containers. To prove linearizability, we must check that no execution step of the operation changes the abstract state (note that other concurrent operations may change the abstract state independently of the join), and that it preserves Invariant I1 and satisfies Properties P1–P4. We need only consider a successful join operation, i.e., one where the CAS in lines 235–241 succeeds, since otherwise the effects of the operation are undone by resetting join_id of m's parent and grandparent to NULL, and thereafter making m replaceable (line 246).

Consider the case where the argument b of low_contention_adaptation is the left child of its parent, so that secure_join_left is called (line 269). Let m, n1, and n2 be the base nodes created by secure_join_left. To check that no execution step changes the abstract state, we will consider each of the statements that could possibly do so; these are the operations that change the tree. We consider each in turn:

- The replacement of b by m, using the CAS operation at line 221, succeeds only if b is b->parent->left. It follows from Invariant I2 and Property P3 that b has been reachable during the execution of secure_join_left up until it is replaced by m. By the construction of m, we have *contents*(m) = *contents*(b), so the replacement does not change the abstract state.
- Next we consider the replacement of n0 by n1 at line 227. Since n0 is found at line 216 and checked to be replaceable before the replacement of b (at line 217), it follows that when the replacement at line 227 succeeds, n0 has been replaceable since line 216. By Invariant I2, the node n0 is still reachable when it is replaced by n1 (line 226 and Fig. 7c). Since by the

construction we have *contents*(n1) = *contents*(n0), the replacement does not change the abstract state.
We also show that n0 is the leftmost neighboring base node of m. This follows by noting that n0 was the leftmost neighbor of m when n0 was found at line 216. Since by Property P2 the range of n0 does not decrease, the interval *range*(n0) is still adjacent to *range*(m) when n0 is replaced, which proves that n0, and hence n1 is also the leftmost neighboring base node of m.

- Next we consider the replacement of n1 by n2 in lines 235–241 (cf. Fig. 7e). This replacement is preceded by finding the parent and grandparent of m, and setting their join_id fields to m, after checking that they are NULL. By Invariant I1, m and n1 are still reachable when the replacement happens. Since the field neigh2 has not been set to ABORTED by any other thread before the CAS, we infer that both m and n1 must have been irreplaceable from their creation until the replacement, implying that they have not been replaced by any other base node. By construction, *contents*(n2) = *contents*(m) ∪ *contents*(n1). When the replacement is performed, the ranges of the involved base nodes are unchanged. Therefore, the extension of *contents*(n2) over *contents*(n1) by adding *contents*(n1) does not affect the abstract state, since by Invariant I3 the added elements from *contents*(m) are all in *range*(m) which is disjoint from *range*(n2).

The first phase of the operation is now finished, and complete_join can be invoked by any thread. It consists of a sequence of assignments that can be performed an arbitrary number times before m is made replaceable at line 264. We should check that they are idempotent, and do not affect the abstract state. Let us consider these assignments.

- The setting of m->neigh1 to n2 (line 252) and the setting of m->parent->valid to false (line 253) are idempotent operations that do not change the abstract state.
- The swinging of the left pointer of m->gparent from m->parent to the node pointed by the right pointer of m->parent is the only operation that modifies a pointer pointing to a route node. It makes m->parent unreachable and makes the right pointer of m->gparent point to the node previously pointed to by the right pointer of m->parent. This implies that *range*(n2) is changed to become what was previously *range*(m) ∪ *range*(n2). Since also *contents*(n2) = *contents*(m) ∪ *contents*(n1), it follows from Property *PartAbsState* that the abstract state is preserved.

We can now check that the join operation preserves Invariant I3 and satisfies Properties P1–P4. We consider the statements where the join is potentially dangerous.

I3: The only statement that can violate Invariant I3 is the change of the link (at lines 256–263) followed by the setting of m->neigh2 to DONE. Above we have established that both *contents*(n2) and *range*(n2) are extended by the contents and range of m, which shows that the invariant is preserved.

Let us next consider the properties that can potentially be violated by a join operation.

P2: Here we should check the splicing out operation (lines 256–263). As shown above, the only change of a range for a reachable node extends it, thereby satisfying the property.

P3: This property follows by checking that the only statements that change a left or right pointer to another node are either
- a CAS operation that replaces a replaceable base node by a fresh node, thus satisfying P3, or
- the splicing out operation of the join operation (lines 256–263); however, the node which becomes a new child of m->gparent was reachable also before the splicing out operation.

This concludes the proof of Property P3.

P4: The only statements that can change a pointer to a route node is at lines 257, 259 and 262. We already established that when `complete_join` unlinks a route node, both the unlinking and unlinked nodes have their `join_id` fields set to the identity of the left or right child of the unlinked node (which must be of type `join_main` with a `neigh2` field pointing to a node of type `join_neighbor`), and that the unlinked node has its `valid` field set to `false`.

5.2 Progress Properties

Let us now prove that the operations on LFCA trees satisfy the stated progress properties. For this, we must establish that the insert, remove, and range query operations are lock-free, and that the lookup operation is wait-free. We make the assumptions that the functions for atomically incrementing/decrementing (`atomic_add` and `atomic_sub`) are wait-free, that the key space for the LFCA tree is finite, and that the number of concurrent threads that can access the LFCA tree is also bounded.

Lookup, Insert, Remove, and Range Query are Lock-Free. Let us first establish that the lookup, insert, remove, and range query operations are lock-free. This means to prove that in any execution where such operations are in progress, at least one of them will finish in a finite number of steps. We make a proof by contradiction. That is, we assume that there is an execution in which from some point on a number of lookup, insert, remove, and range query operations are in progress, but none of them ever finish. Since the number of threads is bounded, there is a bound on how many additional lookup, insert, remove, or range query operations can be initiated beyond this point, since once such an operation has initiated it does not finish (by the assumption that beyond the chosen point, none of them ever finish). Hence, we can assume that from some point on, no lookup, insert, remove, or range query operations finish nor start.

We first note that a high- or low-contention adaptation operation can be initiated only upon completion of an insert, remove, or range query operation (at lines 119 or 211). Since no lookup, insert, remove, nor range query operation ever finishes beyond some point in the execution, it follows that no more adaptation operations are initiated beyond this point. This means that only a bounded number of changes to the tree structure is performed, which allows us to conclude that the functions `find_base_node`, `find_base_stack`, and `find_next_base_stack` will always succeed in a bounded number of steps after the last such change.

We then list the statements that are crucial for progress of each operation. Most of these are CAS operations, which must succeed for the operation to progress. These statements are

- For an insert or remove:
  - the CAS performed by `try_replace` (line 118),
  - making the to-be-replaced node replaceable by helping (line 124).
- For a range query:
  - the insertion of the first `range` base node using a CAS (line 175),
  - making the to-be-replaced node replaceable by helping (line 180),
  - the insertion of a subsequent `range` base node using a CAS (line 195),
  - making a subsequent to-be-replaced node replaceable by helping (line 202),
  - setting the result storage (line 209).

  By Invariant I6, for each range query the CAS statements at lines 175 and 195 succeed in order of ascending ranges, with `range` base nodes being inserted from left to right.
- For a low-contention adaptation, we note that if the CAS operations in `secure_join_left` fail, then the adaptation aborts, after which the adaptation can no longer interfere with progress of other operations. We also note that an operation that is blocked by a failed but

still unaborted join operation will make it abort by helping (at line 79). We therefore need only consider potential progress obstacles in `complete_join`. These are:
– the insertion of the `join_neighbor` node (line 252), using a CAS, and
– the splicing out of the route node (lines 256–263).
For each join operation, we can assess its progress by whether it has completed one or both of the above CAS operations.

From this inventory, we see that each of the above operations progresses through a bounded number of CAS statements and helping actions. Each helping action performs the same sequence of steps as the helped thread, and finishes helping only when the helped operation finishes. Thus, in an execution where no operation finishes, eventually all operations will indefinitely remain at some stage in their progress. Whenever they perform an execution step, either the next CAS will fail, or they will resort to helping a blocking operation. Since each helping can be exited only by finishing, eventually the only blocking operations are CASes. By inspecting these CASes, we infer that one of them can fail only if another one of them succeeds after the relevant pointer has been read. This contradicts the assumption that no CASes succeed after some point in the execution. We have derived a contradiction, and hence the claim that lookup, insert, remove, and range query are lock-free is proven.

Lookup is Wait-Free. Let us now establish that the lookup operation (lines 134–137) is wait-free. This requires to prove that it always completes in a finite number of steps, regardless of what other threads do [Herlihy 1991]. The lookup operation uses `find_base_node` to locate the relevant base node, and thereafter searches it to find the element. Since the leaf container is immutable, the number of steps to locate the node in the leaf container is finite. It therefore remains to prove that the `find_base_node` function (lines 287–308) returns in a finite number of steps. We use a proof by contradiction. Assume that `find_base_node` does not return in a finite number of steps. Then, after performing `NO_JOIN_LIMIT` iterations, `find_base_node` turns off join operations by incrementing the `no_join_count` field of the LFCA tree (line 299). After this point, at most a finite number of route nodes can get linked out from the tree. (Recall that we assume a finite number of concurrent threads.) After this, the structure of the tree can change only through split operations in high-contention adaptations. Split operations do not change *range*(`n`) for any existing node `n` in the tree. From the property that there can be no duplicate keys in the tree, and since splits of base nodes are not allowed to produce empty base nodes (this follows by noting that splits are never initiated in base nodes containing less than two items; see line 277), it follows that successive iterations of `find_base_node` visit nodes with strictly decreasing ranges. From the assumption that the space of keys is finite, it follows that there is a bound on the number of iterations that `find_base_node` can perform before reaching a base node. This contradicts the assumption that `find_base_node` does not return in a finite number of steps, which concludes the proof that lookup is wait-free.

We note that without the code for turning off joins via the `no_join_count` field, the lookup operation would only be lock-free and not wait-free. The scenario illustrated in Fig. 3 shows why the mechanism for turning off joins is necessary.

6 OPTIMIZATION FOR RANGE QUERIES

If possible, it can be advantageous for range queries to avoid writing to shared memory as this induces less cache-coherence traffic. Therefore, we have applied an optimization to our LFCA tree implementation that optimistically tries to perform a range query without writing to shared memory. If this optimistic attempt fails, the range query is performed using the algorithm described in Section 4. The optimistic attempt consists of a test scan and then a validation scan of the base nodes needed for the range query. If nothing has changed between the test and the validation scan,

one can be certain that all base nodes in the scans were present at some point and the optimistic attempt can succeed. This scheme is essentially the same as the one described by Brown and Avni [2012] for doing range queries in their k-ary data structure. We refer to that paper for how to prove this scheme correct.

7 EVALUATION

We will now experimentally evaluate LFCA trees. Our implementation uses an immutable treap for the leaf containers and employs the optimization described in Section 6. To facilitate cache friendly range queries, the treap implementation stores all items in fat leaf nodes containing arrays that can store up to 64 items. The LFCA tree is compared to recent proposals for performing linearizable range queries in ordered sets with Java-based implementations: the *SnapTree* of Bronson et al. [2010], k-ary of Brown and Avni [2012], the method of Chatterjee [2017] applied to a lock-free skiplist (*ChatterjeeSL*), and *KiWi* of Basin et al. [2017].[6] We also include the lock-based CA tree in the comparison; it uses the same immutable treap as the LFCA tree in its leaf containers, and is optimized to take advantage of the immutability of the leaf containers so that range queries and lookups do not read the items in the leaf containers while holding locks [Winblad 2018]. Finally, we also include the lock-free ConcurrentSkipListMap from the Java library, which only supports non-linearizable range queries (*NonAtomicSL*), and the coarse-grained data structure (*Im-Tr-Coarse*) that we described in the introduction. All data structures are in Java as implemented by their respective authors. The maximum number of items in the nodes is set to 64 for k-ary, *LFCA Tree*, *CA Tree* and *Im-Tr-Coarse* as this value has previously been shown to give good results [Brown and Avni 2012]. *KiWi*'s constants are set as described in the *KiWi* paper of Basin et al. [2017].

Platform. All benchmarks were run on a machine with four Intel(R) Xeon(R) E5-4650 CPUs (2.70GHz each with eight cores and hyperthreading, giving a total of 32 actual and 64 logical cores), turbo boost turned off, 128GB of RAM, running Linux 4.9.0-8-amd64 and Oracle JVM 1.8.0_212 (with the JVM flags -Xmx8g -Xms8g -server -d64 -XX:+UseCondCardMark). Each data point comes from the average of three measurements runs of 10 seconds each that were preceded by three warm up runs, also of 10 seconds each, whose purpose is to give the JiT compiler enough time to compile the code. In some cases, error bars showing the minimum and maximum measurements are also visible in the graphs.

Benchmarks. The keys for the operations lookup, insert and remove as well as the starting keys for range queries are randomly generated integers from a range of size S. The data structure is pre-filled before the start of each benchmark run so that it contains $S/2$ random integers. We use $S = 10^n$, $n \in \{5, 6, 7\}$ in all experiments, which correspond to sets of size approximately $5 \times 10^{n-1}$ for each n. Range queries calculate the sum of the items in the range and the number of items in the range. As a sanity check, the average number of items that are traversed per range query is calculated and checked against the expected value.

The benchmark scenarios measure throughput of a mix of operations performed by N threads. In figure captions, the scenarios are described by strings of the form w:A% r:B% q:C%-R, meaning that the benchmark performs $(A/2)$% insert, $(A/2)$% remove, B% lookup operations and C% range queries of maximum range size R. The range sizes are randomly set to values between 1 and R.

[6] We do not compare experimentally against the recently proposed data structure by Arbel-Raviv and Brown [2018] because it does not have a Java implementation and heavily relies on epoch-based memory reclamation. We also do not compare experimentally against the Leaplist [Avni et al. 2013] whose main implementation is in C. Prototype implementations of the Leaplist in Java were sent to us by its authors, but they ended up in deadlocks when running our benchmarks which prevented us from obtaining reliable measurements. Instead, we refer to Section 3 for analytic comparisons to the Leaplist and the data structure by Arbel-Raviv and Brown [2018].

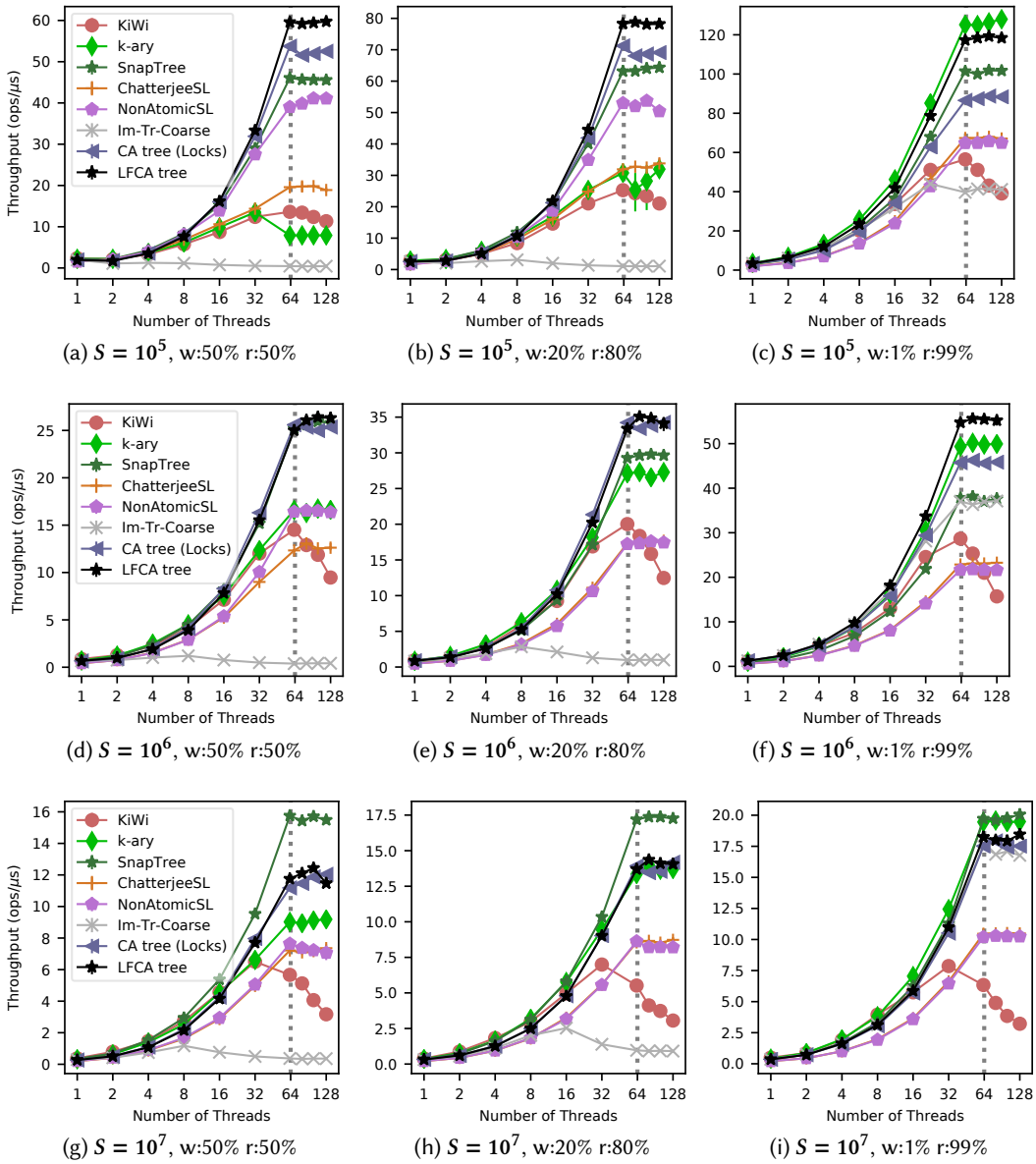


Fig. 10. Single-item operations only. Throughput (operations/μs) on the y-axis and thread count on the x-axis. The set size (S) is increasing row-wise (top-to-bottom), and the percentage of lookup operations is increasing column-wise (left-to-right). Note that occasionally lines are hidden behind other ones; e.g., the SnapTree line is hidden behind the CA tree line in Fig. 10d.

7.1 Performance in Scenarios with Single-item Operations Only

We start with scenarios without range queries. Refer to Fig. 10. As the machine supports only 64 hardware threads, thread counts above 64 (marked with a vertical doted gray line in the graphs)

show how the data structures perform when there are more threads than hardware can execute in parallel. We notice the following:

- Both the lock-based and the lock-free CA tree perform very well compared to the other data structures in scenarios with most contention; i.e., in the scenarios that have a relatively large proportion update operations and small set sizes (Figs. 10a, 10b, 10d and 10e). This shows that the LFCA tree is able to adapt its structure to cope well in highly contended scenarios.
- The LFCA tree also performs well in read-heavy scenarios with 99% lookups (cf. Figs. 10c and 10f). This shows that the LFCA tree can be a good choice even without range queries. In particular, the LFCA tree performs substantially better than the lock-based CA tree at 64 threads and beyond in the read-heavy scenarios (Figs. 10c and 10f), which is likely due to LFCA tree's wait-free lookup operation.
- On the other hand, the performance of the LFCA tree comes overall second best, and is worse than *SnapTree*'s in the scenarios with a relatively large set size (Figs. 10g to 10i). The LFCA tree creates more new objects than the *SnapTree* in these scenarios due to LFCA trees' use of immutable data structures, which means that several new nodes need to be created in each update. More new objects means more work for the garbage collector that needs to be invoked more often in the scenarios where S is large compared to the other scenarios.

Overall, however, the results in Fig. 10 show that the LFCA tree is able to perform well in a wide range of scenarios with single-item operations. The LFCA tree is the top performing data structure in most cases, and is not far behind whenever another data structure is performing better.

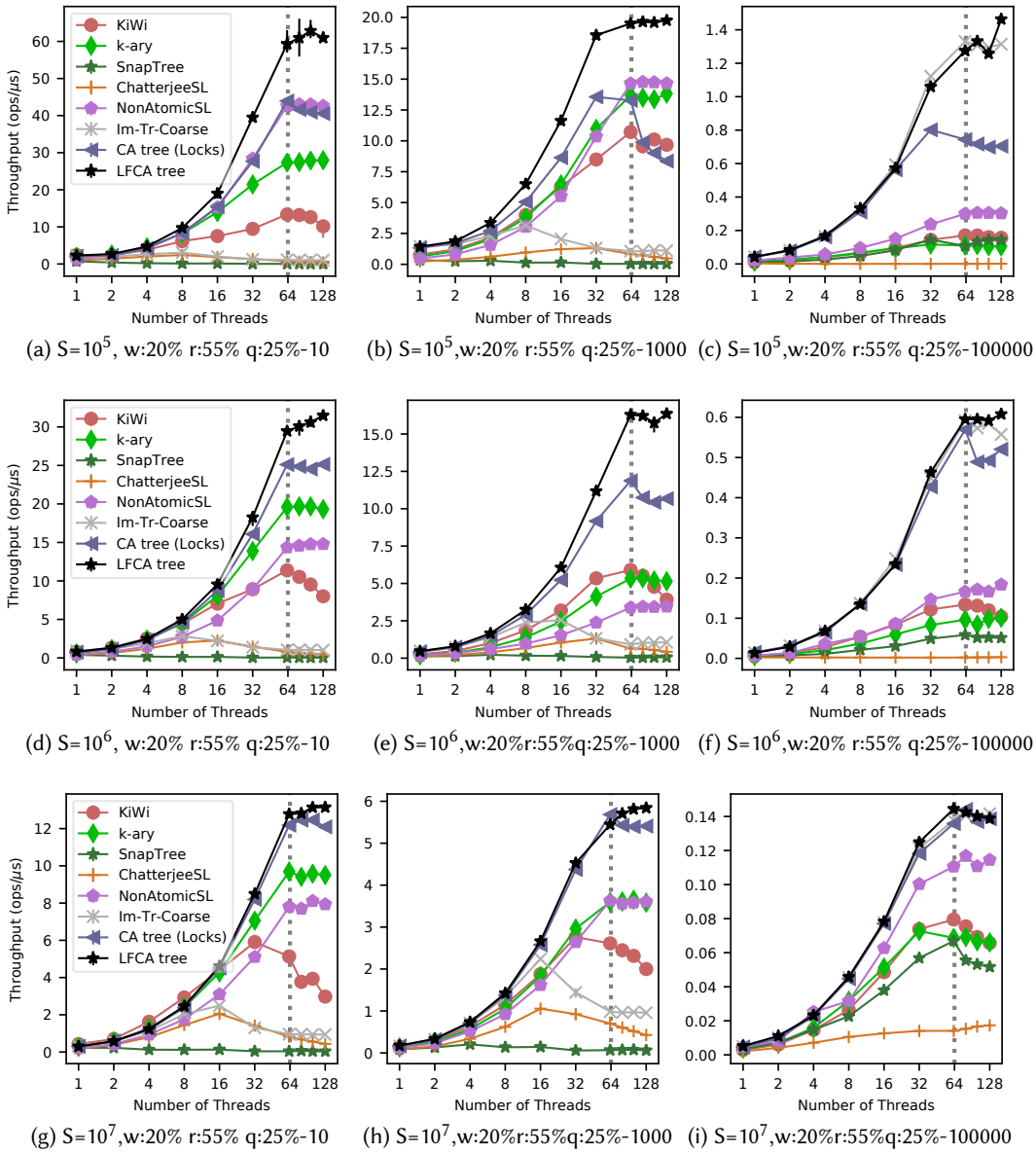
7.2 Performance in Scenarios with Range Queries

Let us now consider scenarios that also contain range queries. Refer to Fig. 11. As in the previous figure, here too the set size S gets larger as rows increase. Also, notice that all the scenarios in the same column have the same mix of operations, and the maximum range size is increasing from left to right.

These scenarios with range queries show the key strength of the CA trees. Overall, the LFCA tree provides substantially better performance than the non-adaptive data structures, likely due to its ability to adapt its structure to the workload at hand.

The three graphs in the left column of Fig. 11 (Figs. 11a, 11d and 11g) show that the LFCA tree performs better than all the other data structures in the scenarios with relatively small range queries of maximum size 10. In the scenarios with moderately-sized range queries of maximum size 1000 (middle column of Fig. 11), the LFCA tree outperforms all the other data structures, sometimes with an even wider margin (see Figs. 11e and 11h), even though the lock-based CA tree also performs very well there. Data structures clearly benefit from fine-grained synchronization in the scenarios with range queries up to a maximum size of 1000 (e.g., *Im-Tr-Coarse* scales relatively poorly in the scenarios of Figs. 11d and 11e). In contrast, in the scenario with large range queries (right column of Fig. 11), it seems that the combination of immutable data and coarse-grained synchronization is the best, as *Im-Tr-Coarse*'s performance is on par with LFCA tree's performance. Therefore, it seems that the LFCA tree can perform extremely well across a wide variety of workloads (only single-key operations, small range queries and large ones) due to its ability to adapt its structure to fit the workload. The slight performance drop that can be observed for the lock-based CA tree after 64 threads is probably due to lock-related problems that become more apparent when thread preemption becomes more common (e.g., when threads may need to wait for another thread that got preempted by the operating system).

It is also interesting to notice that the LFCA tree is substantially better than the lock-based CA tree in the scenarios with set sizes of about $5 * 10^4$ and $5 * 10^5$ (the two top rows) but very similar



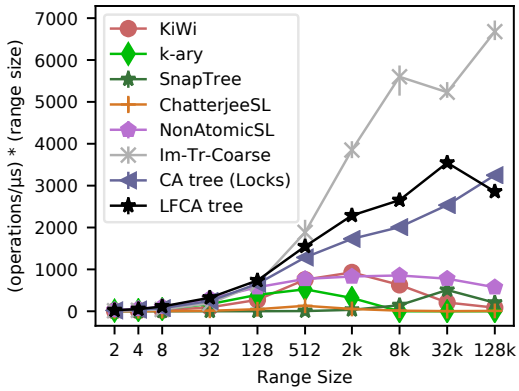
(a) S=10^5, w:20% r:55% q:25%-10 (b) S=10^5,w:20% r:55% q:25%-1000 (c) S=10^5,w:20% r:55% q:25%-100000

(d) S=10^6, w:20% r:55% q:25%-10 (e) S=10^6,w:20%r:55%q:25%-1000 (f) S=10^6,w:20% r:55% q:25%-100000

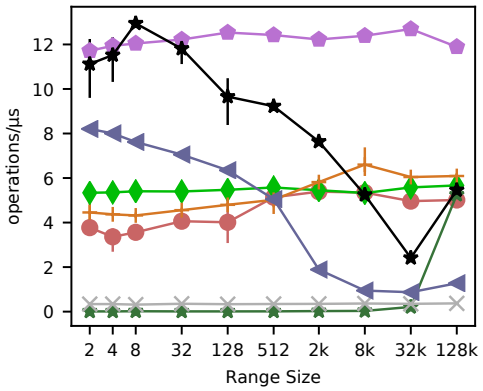
(g) S=10^7,w:20% r:55% q:25%-10 (h) S=10^7,w:20%r:55%q:25%-1000 (i) S=10^7,w:20%r:55%q:25%-100000

Fig. 11. Throughput (operations/μs) on the y-axis and thread count on the x-axis. The set size is increasing with the row number and the range sizes of the range queries is increasing with the column number.

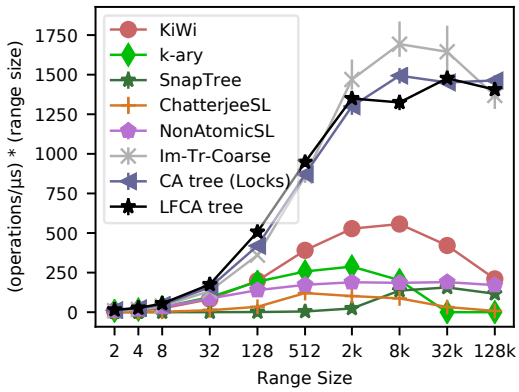
to the lock-based CA tree with a set size of about $5 * 10^6$ ($S = 10^7$). The average proportion of the items in the sets that are covered by range queries gets smaller with increased set size. As a consequence, the number of range queries with overlapping ranges also gets smaller with increased set sizes, and thus the LFCA tree gets less advantage from the fact that threads can help each other.



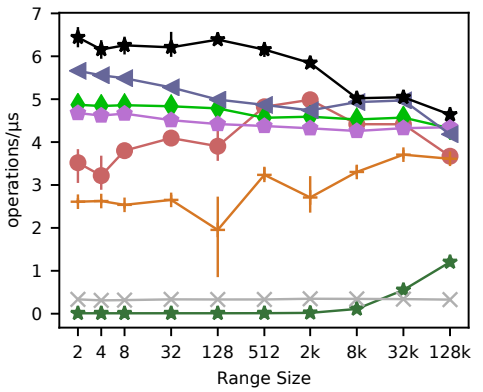
(a) $S = 10^5$, Range queries (with updates in parallel)



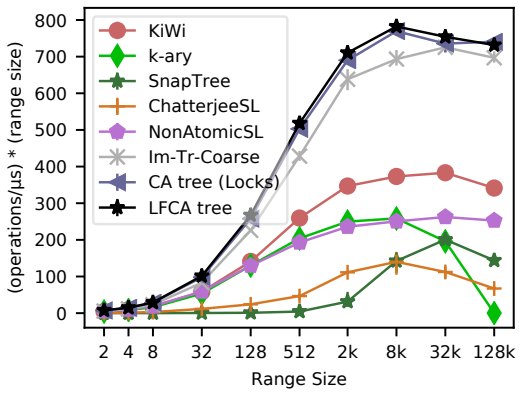
(b) $S = 10^5$, Updates (with range queries in parallel)



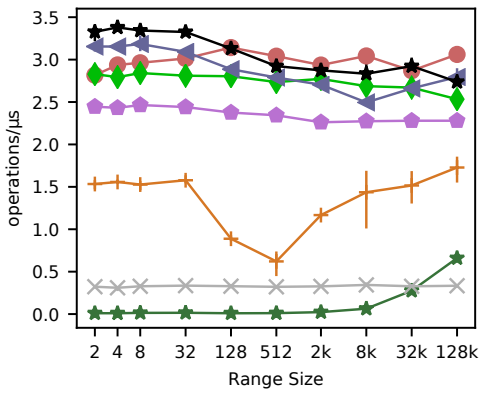
(c) $S = 10^6$, Range queries with updates in parallel



(d) $S = 10^6$, Updates (with range queries in parallel)



(e) $S = 10^7$, Range queries (with updates in parallel)



(f) $S = 10^7$, Updates (with range queries in parallel)

Fig. 12. On the left, throughput for the 16 threads that are doing range queries, and on the right, throughput for the 16 threads that are doing inserts and removes. The set size S is increasing with the row number.

7.3 Separate Threads for Range Queries and Updates

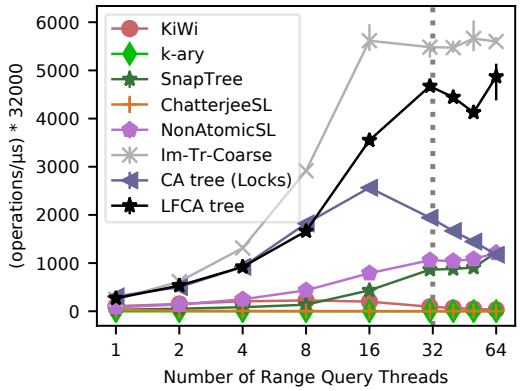
In the benchmark scenarios of the previous section, the threads spend much more time in range queries than in single-item operations when the range queries are large. Thus, another benchmark is needed to measure the data structures' ability to handle large range queries concurrently with frequent update operations. To this end, we use similar experiments to the one developed by Basin et al. [2017] in the paper that described *KiWi*. In our first experiment, we fix the number of threads to the number of physical cores of our machine ($N = 32$). Half of these threads (16) do update operations (insert and remove with equal probability) while the other half (also 16) do range queries with a range of fixed size. We present the throughput for updates (right column of Fig. 12) separately from the range query throughput (left column of Fig. 12), so that one can study the performance of these operations separately. Once again, the set size S is increasing with the row number (top-to-bottom). Note that in the graphs that show the range query throughput, the number of operations per μs is shown multiplied by the range query size on the y-axis to make the graphs more readable. The range query size used in this experiment is shown on the x-axis. Figure 13 shows results from a second experiment of the same kind. In this second experiment, we keep the range query size fixed to $32k$ and instead vary the thread count on the x-axis.

With the small set size ($S = 10^5$) and range queries of size $32k$, all the range queries span more than half the items in the data structures; see Figs. 12a, 12b, 13a and 13b. It is thus not surprising that *NonAtomicSL* has superior update performance in these scenarios, as its update operation does not pay any attention to ongoing range queries. Still, the LFCA tree and the lock-based CA tree are able to provide better throughput for range queries than *NonAtomicSL* even with the small set size, which is probably because the treaps in leaf containers have cache-friendly leaf nodes containing up to 64 items.

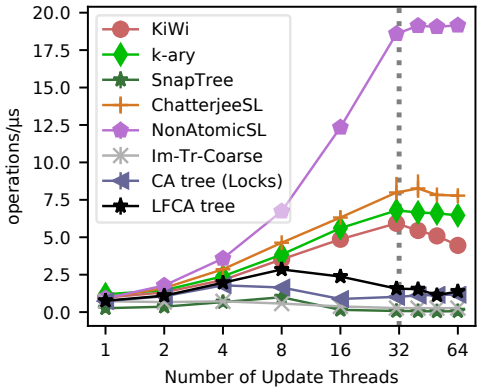
The shape of the graph for the LFCA tree in Fig. 12b is interesting as it seems to have a downward trend until range size $32k$ and then it goes up again. After looking closely at what our experiment is doing, we have come up with a plausible explanation for this apparently strange behavior. Range query threads in this experiment first randomly generate a start key and then calculate the end key by adding the fixed range size value. If the resulting range is outside the range R of keys that can be stored in the data structure, the range is adjusted to have as few items as possible outside R. In the case with $S = 10^5$ and range query size $128k$ (see Figs. 12a and 12b), where the range query size is larger than the range of possible keys, this results in that the start and the end key for the range query ranges are always exactly the same[7]. Thus, in this particular case, it is likely that a range query in an LFCA tree can piggyback on another ongoing range query. Less contention is created when range queries can take advantage of each others' work, which in turn explains why the throughput is better with the range size $128k$ than with $32k$ for the LFCA tree in Fig. 12b. One may think that this particular scenario might be rare in practice. However, note that an LFCA tree's range query does exactly what a clone operation in the LFCA tree would do in this scenario as it takes a snapshot of all items in the tree.

In the scenarios with $S = 10^5$ and a range query size of $32k$ (Figs. 12b and 13b), the update throughput of *NonAtomicSL*, *ChatterjeeSL*, k-ary, and *KiWi* are better than that of LFCA tree. However, LFCA tree makes up for this with a throughput for range queries that often is many times better than the other data structures' range query throughput (see Figs. 12a and 13a). Also, remember that *NonAtomicSL* is not in the same league as the other data structures, as it does not provide linearizable range queries. One could potentially modify the LFCA tree with the aim of increasing the throughput for updates at the expense of the throughput for range queries by letting a range query help update operations that have collided with the range query. (This could be

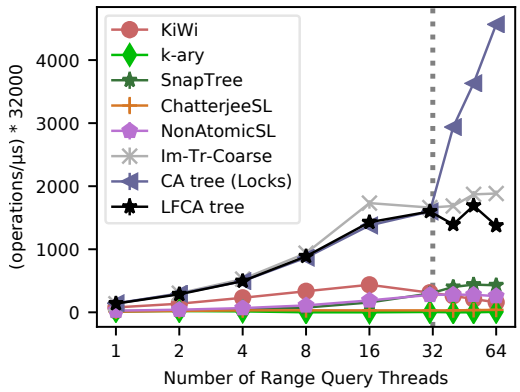
[7]We encourage the reader to take a look at the source code of our benchmark to see how this works in detail.



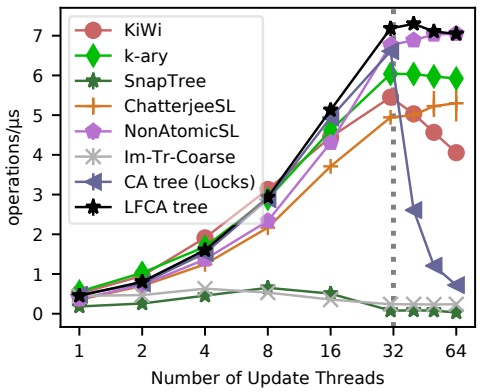
(a) S=10^5, Range queries (size $32k$) with parallel updates



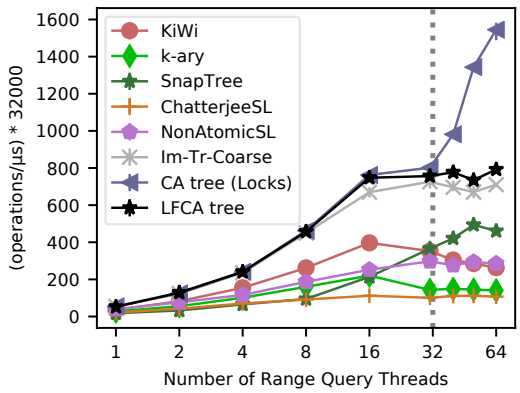
(b) S=10^5, Updates with parallel range queries



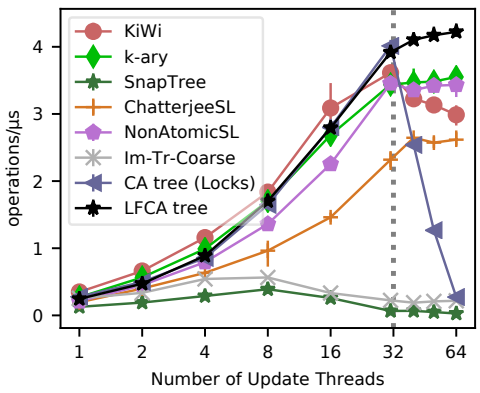
(c) S=10^6, Range queries (size $32k$) with parallel updates



(d) S=10^6, Updates with parallel range queries



(e) S=10^7, Range queries (size $32k$) with parallel updates



(f) S=10^7, Updates with parallel range queries

Fig. 13. On the left, throughput for the range query threads, and on the right, throughput for the threads that are doing only inserts and removes. The range query size is set to $32k$ in all scenarios. The set size S is increasing with the row number.

Table 1. Statistics for the LFCA tree in the scenarios of Fig. 11d (w:20% r:55% q:25%-10).

Threads	1	2	4	8	16	32	64	128
# route nodes	0	76	200	440	890	1.6k	3.1k	3.2k
$\frac{\#\text{ traversed base nodes}}{\#\text{ range queries}}$	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
$\frac{\#\text{ splits}}{\text{milliseconds}}$	0.0	0.04	0.081	0.17	0.33	0.62	1.1	1.1
$\frac{\#\text{ joins}}{\text{milliseconds}}$	0.0	0.033	0.06	0.12	0.24	0.46	0.77	0.78

Table 2. Statistics for the LFCA tree in the scenarios of Fig. 11e (w:20% r:55% q:25%-1000).

Threads	1	2	4	8	16	32	64	128
# route nodes	0	51	130	260	460	720	1.0k	1.0k
$\frac{\#\text{ traversed base nodes}}{\#\text{ range queries}}$	1.0	1.0	1.0	1.0	1.0	1.1	1.1	1.1
$\frac{\#\text{ splits}}{\text{milliseconds}}$	0.0	0.025	0.058	0.12	0.25	0.56	1.1	1.1
$\frac{\#\text{ joins}}{\text{milliseconds}}$	0.0	0.02	0.045	0.091	0.2	0.49	1.0	0.98

Table 3. Statistics for the LFCA tree in the scenarios of Fig. 11f (w:20% r:55% q:25%-100000).

Threads	1	2	4	8	16	32	64	128
# route nodes	0	0	0	11	16	17	22	24
$\frac{\#\text{ traversed base nodes}}{\#\text{ range queries}}$	1.0	1.0	1.0	1.1	1.1	1.3	1.4	1.4
$\frac{\#\text{ splits}}{\text{milliseconds}}$	0.0	0.0	0.0	0.0047	0.013	0.051	0.1	0.11
$\frac{\#\text{ joins}}{\text{milliseconds}}$	0.0	0.0	0.0	0.0036	0.012	0.049	0.1	0.11

achieved, for example, by letting updates register themselves in the irreplaceable base nodes of type `range` that they encounter, so range queries can apply the updates before making their range base nodes replaceable.) This could potentially also decrease the likelihood of starved updates.

In Fig. 13, one can see that the throughput for range queries goes up while the throughput for updates goes down for the lock-based CA tree when one starts to use more threads than hardware threads. (This is shown on the right side of the vertical gray line in the graphs.) This behavior is likely related to the lock implementation (`java.util.concurrent.locks.StampedLock` from the Java standard library) that is used to protect modifications to the base nodes. This highlights one of the advantages that the LFCA tree has over the lock-based CA tree. That is, LFCA tree's performance is not dependent on any lock implementation.

7.4 Some Statistics

We now take a look at the statistics shown in Tables 1 to 3 and Table 4. They show the route node count (measured after the experiments), the average number of traversed base nodes per range query and the number of splits and joins per millisecond for the scenarios that are shown in Figs. 11d to 11f and the experiment in the middle row of Fig. 12, respectively. These statistics indicate that the heuristics works as intended. That is, larger range queries result in fewer route nodes and more threads result in more route nodes. Looking at the number of base nodes traversed per range query, it is also clear that range queries spend a relatively short time traversing shared mutable data, compared to the non-adaptive data structures in the comparison, even for large range queries. This explains how the LFCA tree can perform so much better than the non-adaptive data structures in this comparison.

Table 4. Statistics for the LFCA tree in the scenarios of Figs. 12c and 12d. 16 threads doing range queries and 16 threads doing updates.

Range Size	2	4	8	32	128	512	2k	8k	32k	128k
# route nodes	2.9k	2.7k	2.7k	2.6k	2.5k	2.0k	1.3k	720	380	330
$\frac{\#\text{ traversed base nodes}}{\#\text{ range queries}}$	1.0	1.0	1.0	1.0	1.0	1.3	2.7	5.7	12.0	43.0
$\frac{\#\text{ splits}}{\text{milliseconds}}$	1.1	1.1	1.1	1.2	1.3	1.6	2.5	4.7	11.0	13.0
$\frac{\#\text{ joins}}{\text{milliseconds}}$	0.83	0.8	0.83	0.91	1.1	1.4	2.4	4.6	11.0	13.0

Looking closer, in Table 1, the number of route nodes is increasing with the number of threads which is unsurprising as more threads will increase the conflicts for update operations. However, in Table 3 where the range queries span much more items, the measurements for the numbers of route nodes are quite similar for thread counts 16, 32 and 64. This is likely because the increase in contention due to more threads creates more failing optimistic attempts for range queries (cf. Section 6), which in turn increases the pressure from non-optimistic range queries that span more than one base nodes to not split base nodes. Note that *Im-Tr-Coarse*, which uses coarse-grained synchronization, has similar performance to LFCA tree in the scenario of Table 3 at 64 threads (see Fig. 11f), so more route nodes would likely not cause better performance in this scenario.

7.5 One Last Experiment

The five parts of Fig. 14 show results from a time series experiment that was run in order to illustrate how an LFCA tree adapts its structure when the workload suddenly changes and how this adaptation affects its performance. The top part of the figure shows the number of route nodes, and the bottom part the throughput which is achieved at different time points. At time zero, the LFCA tree contains only one base node with 500k items. The experiment begins with the workload w:20% r:55% q:25%-1000, which is executed using 30 threads for 2.4 seconds. Every 2.4 seconds the maximum range query size changes: besides the initial value (1000) it also takes the values 10, 1000, 10 and 100000 (cf. the values at the top line of Fig. 14). From this time series, we can see that, after each workload change, the rate of change for the number of route nodes gradually decreases until the number of route nodes stabilizes around a certain value. In the parts titled "initial to X-1000" and "X-100000", one can also see a positive change in throughput when the number of route nodes increases/decreases quickly. The change in throughput while the number of route nodes changes quickly is not as big in the three middle parts ("X-10", "X-1000" and "X-10"), which is not strange considering that the synchronization granularity changes relatively less in these parts.

The benchmark set up for the time series experiment is a bit involved in order to obtain numbers that are not disturbed by taking measurements during very short periods of time. It goes as follows. For every time point shown with a dot in the graphs, average measurements from five experiment runs in different JVM instances were collected. Each such run consists of 35 warm up runs and 10 measurement runs (that we take the average measurements from). A warm up or a measurement run does the following after the LFCA tree has been filled with 500k items: It performs a triggering run (skipped in the first workload of the time series) of 2.2 seconds that applies the workload W (where W is the previous workload in the time series), before the actual warm up/measurement run is executed for t seconds (t is always 2 seconds for the warm up runs), after which the number of route nodes and the number of performed operations by the threads are collected. For example, a run to collect measurements for time point 3.4 seconds performs a triggering run with the workload w:20% r:55% q:25%-1000 and then a measurement run with the workload w:20% r:55% q:25%-10 (running for 1 second). The throughput for a time point t_n is calculated as $\frac{o(t_n)-o(t_{n-1})}{(t_n-t_{n-1})}$, where t_{n-1}

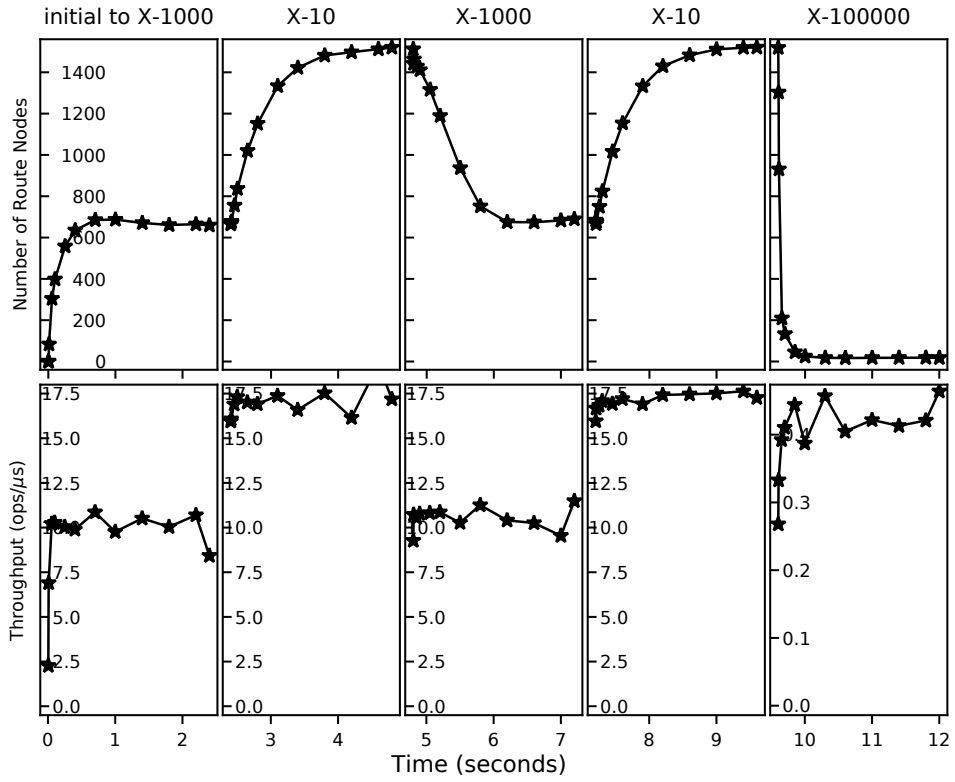


Fig. 14. Time series illustrating sudden changes in the workloads. Number of route nodes in the top and throughput in the bottom. Only the maximum size of the range queries changes between the figures. X = w:20% r:55% q:25%.

denotes the previous time point in the time series and $o(t)$ denotes the average number of performed operations measured for time point t.

8 CONCLUDING REMARKS

We have given a detailed description and correctness arguments for the LFCA tree, the first linearizable lock-free data structure supporting range queries that adapts its structure based on heuristics that take detected contention and information about range queries into account. LFCA trees make use of information gathered at runtime to get a good trade-off between the performance of operations that generally benefit from coarse-grained synchronization and those that generally benefit from fine-grained synchronization. Our experimental evaluation shows that this has real benefits in practice, as the LFCA tree can maintain exceptionally good performance across a wide range of scenarios.

DATA AVAILABILITY STATEMENT

The source code for the LFCA tree as well as the code for the benchmarks are available online [Sagonas and Winblad 2018].

ACKNOWLEDGMENTS

This work was carried out within the Linnaeus centre of excellence UPMARC (Uppsala Programming for Multicore Architectures Research Center), and was partly supported by grants from the Swedish Research Council. We would like to thank the reviewers of our SPAA 2018 paper and for this special issue for valuable comments, and Panagiota Fatourou for pointing out that the lookup operation in our SPAA 2018 paper was only lock-free and not wait-free as we claimed.

REFERENCES

Archita Agarwal, Zhiyu Liu, Eli Rosenthal, and Vikram Saraph. 2017. Linearizable Iterators for Concurrent Sets. https://arxiv.org/abs/1705.08885. (2017). arXiv:arXiv:1705.08885 https://arxiv.org/abs/1705.08885

Maya Arbel-Raviv and Trevor Brown. 2018. Harnessing Epoch-based Reclamation for Efficient Range Queries. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. ACM, New York, NY, USA, 14–27. https://doi.org/10.1145/3178487.3178489

Hillel Avni, Nir Shavit, and Adi Suissa. 2013. Leaplist: Lessons Learned in Designing TM-supported Range Queries. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing (PODC '13)*. ACM, New York, NY, USA, 299–308. https://doi.org/10.1145/2484239.2484254

Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. 2017. 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)*. ACM, New York, NY, USA, 357–369. https://doi.org/10.1145/3018743.3018761

Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10)*. ACM, New York, NY, USA, 257–268. https://doi.org/10.1145/1693453.1693488

Trevor Brown and Hillel Avni. 2012. Range Queries in Non-blocking k-ary Search Trees. In *Principles of Distributed Systems: 16th International Conference, OPODIS 2012. Proceedings*, Roberto Baldoni, Paola Flocchini, and Ravindran Binoy (Eds.). Springer, Berlin, Heidelberg, 31–45. https://doi.org/10.1007/978-3-642-35476-2_3

Trevor Brown and Joanna Helga. 2011. Non-blocking k-ary Search Trees. In *Principles of Distributed Systems: 15th International Conference, OPODIS 2011. Proceedings*, Antonio Fernàndez Anta, Giuseppe Lipari, and Matthieu Roy (Eds.). Springer, Berlin, Heidelberg, 207–221. https://doi.org/10.1007/978-3-642-25873-2_15

Trevor Alexander Brown. 2015. 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)*. ACM, New York, NY, USA, 261–270. https://doi.org/10.1145/2767386.2767436

Bapi Chatterjee. 2017. Lock-free Linearizable 1-Dimensional Range Queries. In *Proceedings of the 18th International Conference on Distributed Computing and Networking (ICDCN '17)*. ACM, New York, NY, USA, Article 9, 10 pages. https://doi.org/10.1145/3007748.3007771

Chao-Hong Chen, Vikraman Choudhury, and Ryan R. Newton. 2017. Adaptive Lock-free Data Structures in Haskell: A General Method for Concurrent Implementation Swapping. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (Haskell 2017)*. ACM, New York, NY, USA, 197–211. https://doi.org/10.1145/3122955.3122973

Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-blocking Binary Search Trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC '10)*. ACM, New York, NY, USA, 131–140. https://doi.org/10.1145/1835698.1835736

Panagiota Fatourou, Elias Papavasileiou, and Eric Ruppert. 2019. Persistent Non-Blocking Binary Search Trees Supporting Wait-Free Range Queries. In *Proceedings of the 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '19)*. ACM, New York, NY, USA, 275–286. https://doi.org/10.1145/3323165.3323197

Phuong Hoai Ha, Marina Papatriantafilou, and Philippas Tsigas. 2007. Self-tuning reactive diffracting trees. *J. Parallel and Distrib. Comput.* 67, 6 (2007), 674–694. https://doi.org/10.1016/j.jpdc.2007.01.011

Maurice Herlihy. 1990. A Methodology for Implementing Highly Concurrent Data Structures. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP '90)*. ACM, New York, NY, USA, 197–206. https://doi.org/10.1145/99163.99185

Maurice Herlihy. 1991. Wait-free Synchronization. *ACM Trans. Program. Lang. Syst.* 13, 1 (Jan. 1991), 124–149. https://doi.org/10.1145/114005.102808

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

Ryan R. Newton, Peter P. Fogg, and Ali Varamesh. 2015. Adaptive Lock-free Maps: Purely-functional to Scalable. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York,

NY, USA, 218–229. https://doi.org/10.1145/2784731.2784734

Chris Okasaki. 1999. *Purely functional data structures*. Cambridge University Press, Cambridge, U.K. https://doi.org/10.1017/CBO9780511530104

Erik Österlund and Welf Löwe. 2014. Concurrent Transformation Components Using Contention Context Sensors. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, USA, 223–234. https://doi.org/10.1145/2642937.2642995

Erez Petrank and Shahar Timnat. 2013. Lock-Free Data-Structure Iterators. In *Proceedings of the 27th International Symposium on Distributed Computing - Volume 8205 (DISC 2013)*. Springer-Verlag New York, Inc., New York, NY, USA, 224–238. https://doi.org/10.1007/978-3-642-41527-2_16

Callum Robertson. 2014. *Implementing Contention-Friendly Range Queries in Non-Blocking Key-Value Stores*. Bachelor Thesis. The University of Sydney.

Konstantinos Sagonas and Kjell Winblad. 2017. A contention adapting approach to concurrent ordered sets. *J. Parallel and Distrib. Comput.* 115 (May 2017), 1–19. https://doi.org/10.1016/j.jpdc.2017.11.007

Konstantinos Sagonas and Kjell Winblad. 2018. Contention Adapting Search Trees. (2018). http://www.it.uu.se/research/group/languages/software/ca_tree.

Raimund Seidel and Cecilia R. Aragon. 1996. Randomized Search Trees. *Algorithmica* 16, 4/5 (01 Oct. 1996), 464–497. https://doi.org/10.1007/BF01940876

Nilufar Shafiei. 2013. Non-blocking Patricia Tries with Replace Operations. In *IEEE 33rd International Conference on Distributed Computing Systems (ICDCS 2013)*. IEEE Computer Society, Los Alamitos, CA, USA, 216–225. https://doi.org/10.1109/ICDCS.2013.43

Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. 2013. F1: A Distributed SQL Database That Scales. *Proceedings of the VLDB Endowment* 6, 11 (Aug. 2013), 1068–1079. https://doi.org/10.14778/2536222.2536232

Kjell Winblad. 2018. Faster Concurrent Range Queries with Contention Adapting Search Trees Using Immutable Data. In *2017 Imperial College Computing Student Workshop (ICCSW 2017) (OpenAccess Series in Informatics (OASIcs))*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 7:1–7:13. https://doi.org/10.4230/OASIcs.ICCSW.2017.7

Kjell Winblad, Konstantinos Sagonas, and Bengt Jonsson. 2018. Lock-free Contention Adapting Search Trees. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA '18)*. ACM, New York, NY, USA, 121–132. https://doi.org/10.1145/3210377.3210413

Yahoo! Developer Network. 2017. Flurry analytics. (2017). https://developer.yahoo.com/flurry/docs/analytics/ Accessed: 2017-07-26.