

# Concurrent aggregate queries

Gal Sela 

Technion, Israel

Erez Petrank 

Technion, Israel

---

## Abstract

Concurrent data structures serve as fundamental building blocks for concurrent computing. Many concurrent counterparts have been designed for basic sequential mechanisms; however, one notable omission is a concurrent tree that supports aggregate queries. Aggregate queries essentially compile succinct information about a range of data items, for example, calculating the average salary of employees in their 30s. Such queries play an essential role in various applications and are commonly taught in undergraduate data structures courses. In this paper, we formalize a type of aggregate queries that can be efficiently supported by concurrent trees and present a design for implementing these queries on concurrent trees. We bring two algorithms implementing this design, where one optimizes for tree update time, while the other optimizes for aggregate query time. We analyze their correctness and complexity, demonstrating the trade-offs between query time and update time.

**2012 ACM Subject Classification** Computing methodologies → Shared memory algorithms; Computing methodologies → Concurrent algorithms; Theory of computation → Data structures design and analysis

**Keywords and phrases** Concurrent Algorithms; Concurrent Data Structures; Aggregate queries; Range queries; Binary Search Tree; Linearizability

**Digital Object Identifier** 10.4230/LIPIcs...

## 1 Introduction

Concurrent programs rely on concurrent data structures as a foundational component. Considerable effort has been dedicated to constructing efficient concurrent data structures that allow using data structures in a concurrent setting. However, not all sequential functionalities have been extended to the concurrent setting. In this paper we look at such a functionality whose concurrent version has not been addressed: efficient aggregate queries. An aggregate query is a query whose answer aggregates into a succinct value information about a range of elements with consecutive keys in the data structure. For instance, a data structure holding employee records sorted by age may be queried regarding the average salary of employees in a certain age range. Efficient aggregate queries in this spirit were not designed for concurrent data structures.

It is important to build efficient concurrent algorithms for aggregate queries, as sequential aggregate queries are used in various applications, and a concurrent extension may scale their execution on a multi-core machine. For instance, order-statistic trees [12], which support the  $\text{select}(i)$  and  $\text{rank}(key)$  aggregate queries (returning the element with the  $i$ -th smallest key, and the position of  $key$ , respectively), are used in Python libraries for sorted containers [34, 19] to efficiently support the basic operations of accessing  $\text{collection}[i]$  and querying  $\text{collection.index}(key)$  respectively. In C++, Boost library for example offers an order-statistic tree [25]. Furthermore, various text editors use augmented tree structures similar to order-statistic trees to efficiently access and manipulate text at a certain location indicated by the editor's cursor. One example is the *rope* data structure [8], which has various implementations (e.g., [35, 27, 1]), and underlies the buffer implementation of some text editors like Xi [24] and Lapce [2].

Naively, one could answer an aggregate query on a sequential data structure by traversing the relevant elements. The concurrent counterpart would be taking a linearizable snapshot of the data structure and traversing it. Previous works on range queries accomplished that [36, 30, 29, 4], but the traversal in this approach costs time linear in the number of elements in the queried range, which is highly inefficient for aggregate queries that may be answered using some metadata without traversing all the relevant elements. There has been work on implementing specific concurrent aggregate queries more efficiently: [32] proposed a way to efficiently support a *size* query returning the total number of elements in a concurrent data structure. They hold appropriate central metadata regarding the data structure’s size. This metadata is updated once on behalf of each effectual operation (we call an operation that modifies the data structure, like a successful *insert* or *delete*, an *effectual operation*). *size* queries take a snapshot of the metadata without accessing the data structure’s elements themselves. However, their mechanism is not extensible to our case, where we want to efficiently answer a query about a specific range, given as an input to the query. To this end, each effectual operation will update multiple metadata pieces across multiple nodes, and it should carry out all the updates (of its target element in the data structure and the multiple metadata values) seemingly-atomically so that queries will get a coherent view of the data structure’s state.

More specifically, we focus our attention on aggregate queries on trees. We will look at external binary search trees (where external means they hold the elements in the leaves) though our work could be extended to other trees as well. For efficiently answering aggregate queries on sequential trees, one could place in each tree node suitable metadata that is a function of the elements in the leaves of the node’s subtree. For instance, an order-statistic tree augments a binary search tree with a *size* field expressing the number of elements in the node’s subtree. The metadata function should be chosen to be one that effectual operations could maintain during their root-to-target-leaf traversal for not harming their asymptotic time complexity, and also one that aggregate queries could use to get an answer via root-to-leaf traversals (instead of naively traversing the relevant elements), thus executing in time linear in the traversed path length instead of at the number of elements in the query’s range. We formalize the type of addressed aggregate functions and queries in Section 2.

Extending such augmented trees to support concurrency is not simple: while each effectual operation affects multiple locations (its target-leaf area and metadata fields in the nodes along its root-to-leaf path), a query should somehow obtain a consistent view of the nodes it traverses, including their metadata fields. If insertions and deletions simply update the relevant metadata fields one by one, a query might obtain an inconsistent view of the metadata, in which ongoing effectual operations might be reflected only in part of the obtained fields. The query should obtain a linearizable snapshot of the query path including its metadata, where each effectual operation is either fully absent or fully reflected in all relevant snapshotted fields.

To get a snapshot while viewing the multi-point updates per insertion or deletion as atomic, one could naively use a lock, or employ transactions—for atomically updating the target-leaf area and the metadata along the root-to-leaf path by each successful insertion or deletion, as well as for reading the root-to-leaf path by queries. But that would be inefficient since all effectual operations as well as concurrent queries would be serialized one after another because they all synchronize on the metadata in the root node. We wish to take an alternative more fine-grained approach, where we let queries traverse the tree without taking a lock, but make sure to fill in the missing updates in the otherwise potentially partial picture a query obtains. We employ two mechanisms to achieve this goal: multi-versioning, and

announcements of ongoing effectual operations. The first enables queries to ignore effectual operations that are considered to occur after them, and the second enables them to take into account all effects of effectual operations that are considered to occur before them.

In more detail, each effectual operation and each aggregate query get a timestamp. An aggregate query with timestamp  $ts$  is linearized (i.e., considered to occur) after all effectual operations with timestamp  $\leq ts$  and before all effectual operations with timestamp  $> ts$ . By multi-versioning we mean that effectual operations which update values in a timestamp later than an ongoing query's timestamp are responsible to leave old versions of the updated objects. These old versions enable queries to obtain a snapshot of the relevant parts of the data structure (including metadata values) without taking into account new updates that are considered to happen after them: each query grabs a timestamp and then builds its view of the query path by reading object versions tagged with this timestamp (to be precise, with the biggest timestamp that is  $\leq$  this timestamp). Indeed, such a snapshot will not reflect updates with a timestamp greater than the query's timestamp; however, the snapshot might contain partial updates for some still-ongoing insertions or deletions considered to precede the query according to their timestamp. Therefore, insertions and deletions globally announce themselves, and queries read these announcements to fill in missing details about them by themselves, and form the desired full view of their traversed path.

Different approaches could be taken toward the operations announcement and the aggregate metadata representation, optimizing for the time complexity of either effectual operations or aggregate queries. We bring two algorithms implementing our design: **FastUpdateTree** optimizing for tree update time, in fact incurring zero additional asymptotic time on the original tree operations, and **FastQueryTree** optimizing for aggregate query time, reducing its time to be linear in its traversal length times a factor dependent on the number of concurrent operations (in comparison to time linear in the number of elements in the queried range in the naive implementation). Applications in which it is important that the original tree operations run fast should use **FastUpdateTree**, while applications that require fast aggregate queries should use **FastQueryTree**.

To reduce the contention on effectual operations incurred by our design, **FastUpdateTree** lets them work mostly on single-writer fields written only by the thread that performs the operation. Both the object in which threads announce their effectual operations, and the aggregate metadata field in each tree node, are arrays with a single-writer cell per thread. This demonstrates a time-space trade-off: effectual operations can execute faster by paying in more space. The trade-off between the original tree operations' time and aggregate query time is also apparent here, as aggregate queries will have to invest more work to calculate aggregate values based on the per-thread aggregate metadata array.

**FastUpdateTree** demonstrates that effectual operations do not have to serialize (order themselves one in respect to another) and they in fact do not need to be aware of other effectual operations at all, which is interesting to note as it might initially seem like they must be ordered for the metadata to be updated of relevant effectual operations by their order. On the other hand, in **FastQueryTree**, for aggregate queries to run faster, they are not responsible for combining per-thread metadata into a unified aggregate value, and instead effectual operations serialize in order to know which effectual operations precede them and help update metadata on behalf them. The aggregate metadata is made of a single field (and not an array), updated on behalf of relevant effectual operations by their order. To serialize themselves, effectual operations announce themselves by enqueueing their announcement to a global queue.

As parallelization of aggregate queries is notably absent in the literature, it is perhaps

unsurprising that this challenge has attracted attention from other researchers. In an independent and very interesting complementary research, the problem of concurrent aggregate queries is also being explored [23], where a different approach is presented. In [23] all operations are first serialized on a queue of operations at the root. Each operation then helps all preceding operations in the queue to advance to the appropriate child before proceeding to the next node in its own traversal. Queues are used in each node of the tree to hold all operations that still need to be executed on that node's subtree. Reminiscent of hand-over-hand synchronization, this approach provides a virtual snapshot for the queries, as each operation views the tree with all previous operations have already been executed and none of the subsequent operations have. Their alternative approach complements ours, providing a more comprehensive view of possible efficient solutions for aggregate queries. Their solution optimizes aggregate query performance at the expense of the other operations, while our approach also includes the **FastUpdateTree** solution, which offers optimal complexity for the original tree operations.

**Contribution.** We present a formalization of aggregate queries that may be efficiently supported on concurrent trees, and their related aggregate metadata functions. We then present a design for supporting such queries on concurrent trees, with two different implementations, presenting a trade-off between the time complexity of the original tree operations and the newly-added aggregate queries.

**Organization.** The formalization of aggregate queries appears in Section 2. Some terminology required to discuss our algorithms is brought in Section 3, followed by our design presentation in Section 4 and the analysis of the algorithms in Section 5. Related work is covered in Section 6. We conclude with a discussion of future directions in Section 7.

## 2 Aggregate metadata and aggregate queries

We look at aggregate queries on binary trees, using metadata placed in each node aggregating information about its subtree. The basic idea is to use this metadata to answer queries efficiently without traversing all the elements in the query's range, while making sure not to substantially harm the asymptotic time complexity of insertions and deletions, which should be able to maintain the metadata throughout their root-to-leaf traversal, as the only affected metadata should lie along the path to their target key.

### 2.1 Aggregate functions used for metadata

The aggregate metadata we will add to tree nodes is the value of an aggregate function  $f$  applied to the set of  $(key, value)$  elements in the leaves of the node's subtree. An aggregate function is a function  $f : \mathcal{P}(A) \setminus \{\emptyset\} \rightarrow B$ , where  $A, B$  are non-empty sets. Our aggregate functions' domain would be all the non-empty subsets of the set of possible  $(key, value)$  elements in the tree's leaves (denoted by  $A$ ). We note that aggregate functions are usually referred to as applied to multisets, since database rows may include repetitions. However, since we treat the domain as pairs of  $(key, value)$  and the tree's keys as unique, we define aggregate functions to operate on sets. The codomain of aggregate functions, denoted here by  $B$ , could be for example  $\mathbb{R}$ ,  $\mathbb{Z}_2$ , the set of possible tree keys, the set of possible tree values, etc., or a product of several such sets (so that  $B$ 's elements are tuples). Next we bring a definition that will give us the first useful property we need of aggregate functions:

► **Definition 1** (additive aggregate function). *We say that an aggregate function  $f : \mathcal{P}(A) \setminus \{\emptyset\} \rightarrow B$  is additive if there exists a binary operation  $\oplus : B \times B \rightarrow B$  such that for every*

disjoint  $X_1, X_2 \in A$ ,

$$f(X_1 \uplus X_2) = f(X_1) \oplus f(X_2) \quad (1)$$

To better understand how an additive aggregate function generally looks, we bring the following equivalent definition:

► **Definition 2** (additive aggregate function – alternative). *An aggregate function  $f : \mathcal{P}(A) \setminus \{\emptyset\} \rightarrow B$  is additive if there exists a binary operation  $\oplus : B \times B \rightarrow B$  such that  $(B, \oplus)$  is a commutative semigroup (namely,  $\oplus$  is associative and commutative) and  $f$  satisfies:*

$$f(X) = \bigoplus_{a \in X} f(\{a\})$$

(where  $\oplus$ , though binary, may be extended to be applied to any number of elements from  $A$  thanks to the operation's associativity).

In fact, we could build  $f$  on top of any “base” function  $F$ : for any commutative semigroup  $(B, \oplus)$  and any  $F : A \rightarrow B$ ,  $f(X) := \bigoplus_{a \in X} F(a)$  is an additive aggregate function.

Examples of useful additive aggregate functions include **size** (sometimes denoted by **count**), for which  $B := \mathbb{Z}$  and  $\oplus := +$  (simple addition), and **sum** over the keys or over the values with  $\oplus$  taken to be addition over the appropriate domain, e.g.  $\mathbb{R}$  if the keys or values are taken from  $\mathbb{R}$ . The sum of squared values  $f(X) = \sum_{(key, value) \in X} value^2$  may be useful for sample variance calculation. Product is another associative commutative operation that could be taken as  $\oplus$  to produce the multiplication of the keys or values.

Requiring the metadata in tree nodes to be a value of an additive aggregate function over the set of  $(key, value)$  elements in the leaves of the node's subtree ensures that the metadata in each node may be directly updated upon an insertion to its subtree:

► **Property 3.** *Upon an insertion of a  $(key, value)$  element into a certain subtree, the new updated value of the metadata in the subtree's root may be calculated by  $old \oplus f(\{(key, value)\})$  where  $old$  is the old metadata value.*

**min** and **max** are also additive aggregate functions, where  $\oplus$  is taken to be **min** or **max** respectively, but our work will not support using them as metadata in nodes since they do not satisfy the following property:

► **Definition 4** (subtractive aggregate function). *We say that an aggregate function  $f : \mathcal{P}(A) \setminus \{\emptyset\} \rightarrow B$  is subtractive if it is additive, and there exists a subtractive binary operation  $\ominus : B \times B \rightarrow B$  such that for every disjoint  $X_1, X_2 \in A$ ,*

$$f(X_2) = f(X_1 \uplus X_2) \ominus f(X_1) \quad (2)$$

The following definition is equivalent to Definition 4:

► **Definition 5** (subtractive aggregate function – alternative). *An aggregate function  $f : \mathcal{P}(A) \setminus \{\emptyset\} \rightarrow B$  is subtractive if it is additive with an operation  $\oplus : B \times B \rightarrow B$ , and  $(B, \oplus)$  is a group (namely,  $\oplus$  has an identity element and every element of  $B$  has an inverse element).*

This alternative definition helps to materialize the  $\ominus$  operator from Definition 4: as each  $b_2 \in B$  has an inverse element  $-b_2$ , we define  $\ominus$  as follows (and this satisfies the requirement of Equation (2)):

$$b_1 \ominus b_2 := b_1 \oplus -b_2$$

All additive aggregate function examples brought above, except for `min` and `max`, are subtractive. For all of them  $\ominus$  should be subtraction, other than for product for which it should be division.

The metadata field we will add to the tree nodes will hold the value of an aggregate function which is not only additive but rather also subtractive. This way, the following property will hold and enable to update the metadata accordingly:

► **Property 6.** *Upon a deletion of a  $(key, value)$  element from a certain subtree, the new updated value of the metadata in the subtree's root may be calculated by  $old \ominus f(\{(key, value)\})$  where  $old$  is the old metadata value.*

On deletion we conceptually invert the node's metadata value to its state without the deleted item, which we could not have done for non-subtractive aggregate functions like `min`. Being able to directly update the metadata in a certain node to reflect a deletion in its subtree, without re-calculating the metadata node by node from the location of the deleted leaf upwards, is especially important for our algorithms, where the deletion initiator is not the only who needs to calculate its effect on the metadata in the deleted leaf's ancestors. Other operations whose root-to-leaf path intersects the deletion's path might need to do so as well in ancestors mutual with this deletion, and they should not traverse all the way from the deleted leaf to the relevant ancestor which might be costly.

## 2.2 Aggregate queries

An aggregate query on a data structure returns a result based on multiple data elements of the data structure. We look specifically at trees, whose augmentation with appropriate metadata in all nodes may enable aggregate queries to execute efficiently through root-to-leaf traversals. Metadata obtained during the traversals may be used both to navigating through the tree, choosing the appropriate path to traverse, and for calculating the query's result. Some queries require multiple root-to-leaf traversals for computing their answer. These traversals may be independent of each other, which means they could be executed concurrently, followed by a central calculation of the query's answer using their results. But there are also queries that require a serial execution of traversals, which is the case when each traversal depends on the result of the previous traversal. Accordingly, we next define a *simple* aggregate query, which executes only independent traversals; a general aggregate query is a chain of one or more simple aggregate queries composed one on another: the user's input is the input of the first simple query in the chain, the output of the  $i$ -th simple query in the chain is the input of the  $(i + 1)$ -st query, and the output of the last simple query is the output of the whole query. Appendix A brings examples of supported aggregate queries, and gives intuition to what kind of queries require each part of the construction: a basic traversal, multiple traversals, and chained traversals.

A simple aggregate query performs one or more independent root-to-leaf traversals to gather the information required to answer the query, and then computes the answer using the traversals' results. The traversals may be executed concurrently as they are independent of each other. All traversals have a fixed structure that appears in the template in Figure 1 detailed below; the only difference between them is in the `shouldDescendRight` method called for each node to determine to which child the traversal should proceed, and hence the leaf they eventually arrive at. A traversal outputs a tuple  $(aggValue, leaf)$ , where  $aggValue$  is the value of the metadata subtractive aggregate function on the set of  $(key, value)$  pairs of all leaves in the key range  $(-\infty, k)$  with  $k$  being the key in the leaf ending the traversal, and  $leaf$  is this leaf's object. A simple aggregate query takes an input (e.g., a key or an index)



```

1 aggregateTraversalTemplate<shouldDescendRight>():
2 init:
3   node = tree.root
4   skippedNodesAggValue = identityElement(B,⊕f)
5 traverse:
6   while node is not a leaf:
7     aggValueUpToCurrentKey = skippedNodesAggValue ⊕f node.left.aggValue
8     if shouldDescendRight(aggValueUpToCurrentKey, node.key):
9       node = node.right
10      skippedNodesAggValue = aggValueUpToCurrentKey
11   else:
12     node = node.left
13 return (skippedNodesAggValue, node)

```

■ **Figure 1** Template for a basic aggregate query

and needs to return the required output, namely, the query's answer. Its definition is made of two components: as many `shouldDescendRight` methods as the traversals it needs (each of them may integrally use the query's input), and a `computeAnswer` method that takes the list of the traversals' outputs and computes the query's answer.

Next, we present the traversal algorithm ran during aggregate queries execution. Throughout the paper, we denote the nodes' metadata type by  $B$  and the metadata subtractive aggregate function (whose codomain is  $B$ ) by  $f$ . We further denote  $f$ 's corresponding operators (from Definitions 1 and 4) by  $\oplus_f$  and  $\ominus_f$ , and the identity element of the group  $(B, \oplus_f)$  by `identityElement(B,⊕f)`. The general traversal template, shown in Figure 1, takes as parameter a `shouldDescendRight` method (defined by the aggregate query). It performs a root-to-leaf traversal on the tree. At any point in the traversal, `skippedNodesAggValue` holds the value of  $f$  on the set of  $(key, value)$  pairs of all leaves found in subtrees that the traversal has jumped over so far (namely, descended to the right while they were in the left subtree). At the end of the traversal, this will be the set corresponding to all keys preceding the key of the leaf the traversal has reached. For each traversed node, the aggregate value `skippedNodesAggValue` computed so far is combined using  $\oplus_f$  with the aggregate value of the current left subtree, found in the metadata of the left child, to form `aggValueUpToCurrentKey`—which represents the value of  $f$  on the set of  $(key, value)$  pairs of all leaves with  $key \leq$  the key of the current node in the traversal (Line 7). The computation of this value is made possible using one simple  $\oplus$  operation thanks to using an additive aggregate function on the subtree's leaves as the node's metadata. Then the query-specific method `shouldDescendRight` is called to determine to which child the traversal should proceed (Line 8). It takes `aggValueUpToCurrentKey` and the current node's key. (For example, a traversal that aggregates  $f$  for all keys up to a certain key  $k$  should search for  $k$ , and the query would accordingly define a `shouldDescendRight` method that returns true iff  $k \geq$  the current node's key—we assume a binary search tree where keys equal to or greater than a node's key are stored in its right subtree. An example to `shouldDescendRight` that requires `aggValueUpToCurrentKey` to make its decision is a `select` query, for which  $f$  counts the number of elements, and whose traversal proceeds to the right child if the required leaf's index  $>$  `aggValueUpToCurrentKey`.) In case of descending to the right, `skippedNodesAggValue` is updated to take into account the leaves of the current left subtree (Line 10). The traversal stops when it reaches a leaf node, and returns `skippedNodesAggValue` and the leaf object.

### 3 Terminology

We will look at binary search trees implementing dictionaries. A *dictionary* (synonymously *map* or *key-value map*) is a collection of distinct keys with associated values, supplying the following interface operations: an `insert( $k, v$ )` operation which inserts the key  $k$  with the associated value  $v$  if the key does not exist or else returns a failure; a `delete( $k$ )` operation which deletes  $k$  and its value if  $k$  exists and returns the value or else returns a failure; and a `contains( $k$ )` operation which returns  $k$ 's value if  $k$  exists else returns NOT\_FOUND. We call `insert` and `delete` that return failure *failing*, otherwise they are *successful*. We call successful `insert` and `delete` *effectual operations*. We will assume binary search trees where keys smaller than a node's key are stored in its left subtree and keys equal to or greater than a node's key are stored in its right subtree. We will look specifically at external trees, i.e., their items are found in the leaves.

We assume the basic asynchronous shared memory model [17], in which a fixed set of threads communicate through memory access operations. An execution on a concurrent data structure is considered *linearizable* [18] if each method call appears to take effect at once, between its invocation and its response events, at a point in time denoted its *linearization point*, in a way that satisfies the sequential specification of the objects. A concurrent data-structure is *linearizable* if all its executions are linearizable. The two tree algorithms we present, `FastUpdateTree` and `FastQueryTree`, are linearizable.

### 4 The design

An overview of our design for augmenting a concurrent binary search tree with subtractive aggregate metadata to support aggregate queries appears in Section 4.1. We describe the two different approaches towards implementing this design in Section 4.2. We then delve into details—of the tree on which we demonstrate our methodology in Section 4.3, the common backbone of the two algorithms in Section 4.4, the unique details of each algorithm in Appendices C and D and optimizations in Appendix G.

#### 4.1 Design overview

We wish to extend a basic tree with support for efficient aggregate queries. For that we add to tree nodes aggregate metadata that will enable to answer them efficiently. The aggregate metadata in each tree node equals to the value of a subtractive aggregate function  $f$  applied to the set of  $(key, value)$  elements in the leaves of the node's subtree.

We need to correctly answer aggregate queries even if they are concurrent with operations that update the tree. The challenge is to overcome the fact that effectual operations carry out multiple modifications of the tree, and let aggregate queries observe a consistent view of the parts they traverse in the data structure, as if each concurrent effectual operation has completely taken place or did not start at all. Each effectual operation and each aggregate query obtain a timestamp. Every query should observe all modifications related to effectual operations with timestamps  $\leq$  its timestamp, and not see modifications related to effectual operations with a greater timestamp.

For that, on the one hand we need a query to consider all modifications by effectual operations, which run concurrently with the query and have a timestamp  $\leq$  its timestamp, even if some of these modifications have not yet occurred. To this end, ongoing effectual operations announce themselves by adding an *Update* object with their details to a global *CurrUpdates* object, to enable concurrent queries to complete the missing details by themselves.



Among other details, the *Update* object contains a *timestamp* field indicating the operation's timestamp, and a *done* flag indicating whether the operation is done both updating affected aggregate fields and applying itself to the tree. (All the fields are detailed in Appendix B.1.)

On the other hand, we also need to prevent effectual operations, which run concurrently with a query and have a greater timestamp than the query's timestamp, from overriding data the query is about to use with new data. To this end, we employ versioning for modifiable fields in the tree's nodes: effectual operations leave old versions of the data for the queries to inspect, and write the new values in new versions they create for the relevant fields. More specifically, we use timestamped version lists for both the child pointers and the added aggregate metadata field in the tree nodes. These versioned fields are made of a linked list of values tagged with descending timestamps. Reading them may be performed with or without an input timestamp, while writing to them must be done with an input timestamp, as detailed next.

A *versioned read* takes a timestamp  $ts$ , and traverses the list until reaching a version with timestamp  $\leq ts$ , whose value it returns. A *standard read* returns the value in the first (most recent) version in the list. A *standard timestamped read* returns the value in the first version in the list and its timestamp. A write to a versioned field (unprotected write, which should be performed while it is guaranteed that no other thread concurrently tries to write to the field) takes a value and a timestamp  $ts$ , and links a new version to the head of the linked list of versions with the new value and timestamp  $ts$ . A thread-safe write to a versioned field (namely, while other threads might concurrently try to write to it) takes a value and a timestamp  $newTs$ , in addition to  $lastTs$ —the timestamp expected to be the most recent one in the list; it obtains the current first version, and if its timestamp is  $lastTs$  it tries to link before it (to the head of the linked list) a new version with the new value and timestamp  $newTs$  using a compare-and-swap (CAS). The full pseudocode for versioned fields appears in Appendix B.2. Which kind of read or write is used in what scenarios will become clear in the following sections.

## 4.2 The two algorithms

The two proposed algorithms share the same backbone, but they handle differently the way operations obtain a timestamp and announce themselves in *CurrUpdates* in case of effectual operations, as well as the aggregate metadata representation. These could be biased in favor of the time complexity of either effectual operations or aggregate queries. We design two algorithms to handle the timestamps and aggregate values—**FastUpdateTree** that preserves the asymptotic time complexity of operations in the base tree algorithm, incurring no additional asymptotic time cost on them; and **FastQueryTree** which offers a better time complexity for aggregate queries.

In the design of the tree algorithm's extension for supporting aggregate queries, effectual operations have to perform several additional steps in which they might potentially contend with operations of other threads: globally announce and unannounce themselves and update the metadata fields affected by the operation. **FastUpdateTree** aims to reduce the contention on effectual operations incurred by our extension, and thus lets effectual operations work mostly on single-writer fields written only by the thread that performs the operation. This manifests in both the announcement mechanism and the aggregate metadata representation:

In **FastUpdateTree**, the *CurrUpdates* object—in which effectual operations announce themselves—is an array with a cell per thread to point to its *Update* object. When effectual operations announce themselves in this array, they do not order themselves in respect to each other, and there is no variable they serialize on (like obtaining a unique timestamp).

Aggregate queries are the ones to grab a timestamp while incrementing a global *Timestamp* field using a fetch-and-increment; effectual operations only need to obtain a timestamp bigger than the last query's timestamp, for writing their updates of versioned fields in a newer version, not overriding data the query needs. For that, an effectual operation first announces itself with an unset timestamp, and then it obtains the global timestamp value and sets it in the announcement's timestamp field using a CAS. A concurrent aggregate query might be ahead of it, obtaining the global timestamp value and CASing it into the announcement's timestamp, which is what aggregate queries do for all effectual operations with an unset timestamp they encounter in their first traversal in *CurrUpdates*.

As for the aggregate metadata field in each **FastUpdateTree** node, it is also an array with a cell per thread, where each cell is a versioned field (namely, holds a linked list of versions with different timestamps) containing metadata regarding operations by the associated thread on the node's subtree. Aggregate queries can correctly calculate the total aggregate value from the per-thread values using  $\oplus_f$  (the aggregate function's binary operation), thanks to  $\oplus_f$  being commutative and associative—which is the case as we allow only an additive aggregate function  $f$  (whose  $\oplus_f$  is commutative and associative by Definition 2).

**FastQueryTree** on the other hand favors the performance of aggregate queries, hence does not let them gather values from a per-thread metadata array; instead, it allocates a single versioned metadata field in each tree node. To update such a field to reflect an effectual operation, it needs to know which effectual operations are ordered before it, in order to update the metadata to reflect all relevant operations that occurred so far. To this end, all effectual operations serialize by enqueueing their *Update* object to a queue, and while doing so they also get a unique timestamp so that the timestamps induce a total order on all effectual operations (specifically, they enqueue an *Update* object with a timestamp greater by 1 than the timestamp of the preceding *Update* object in the queue). Namely, *CurrUpdates* is a queue containing *Update* objects with consecutive timestamps. Aggregate queries obtain a timestamp (that determines which effectual operations they take into account) by simply reading the timestamp in the *Update* object in the current last node of the queue (namely, the most recent *Update*), which is considered the current global timestamp. Equipped with this timestamp, they know which version of each aggregate metadata field to obtain and which announced effectual operations they should consider.

In Section 4.4 we elaborate on the common backbone of both algorithms, and the full details of the different points between the two algorithms are deferred to Appendices C and D.

### 4.3 The base tree

The proposed methodology for augmenting a concurrent binary tree with subtractive aggregate metadata to support aggregate queries, could be applied to different concurrent trees. We will focus on a specific concurrent tree to demonstrate the methodology—the linearizable binary search tree of [13, 14]. This is an external full binary tree (the elements are in the leaves; each internal node has two children). Three sentinel nodes are always part of the tree: a root node with the key  $-\infty$ , a leaf node (which is the root's left child) with the key  $-\infty$ , and a leaf with the key  $\infty$ . In particular, the root never changes. Each internal node has a memory-word-sized field containing two special locks, each used to protect the link to another one of its children. It is possible to acquire one of the locks without affecting the other, or permanently acquire them both in a single atomic operation (the latter has a different signature than when each separate lock is acquired, and is used to permanently lock a parent of a leaf that is about to be removed, since as part of the delete operation the

parent will also be unlinked and its edges will never be modified again). We will refer to acquiring a node's lock associated with a certain child as locking the edge to the child, and permanently acquiring both locks as a permanent lock of the node.

A `contains( $k$ )` operation is simple and operates like in a sequential algorithm: it searches for  $k$  until reaching a leaf node, and returns an answer based on the leaf's key. An `insert( $k$ )` operation wishes to insert a new leaf— $N$ —with the key  $k$ . It first searches for  $k$  in the tree. If it finds it, it returns failure. Otherwise, it reaches a leaf  $L$  with a key  $\neq k$ . It locks the edge from  $P$ — $L$ 's parent—to  $L$  (or restarts if the attempt to lock failed), modifies this edge to point at a new internal node pointing to  $L$  and  $N$  as its children (and having the right child's key as its key), and then unlocks the edge. A `delete( $k$ )` operation starts with searching for  $k$  in the tree. If it does not find it, it returns failure. Otherwise, it reaches a leaf  $L$  with the key  $k$ ; let  $S$  be its sibling,  $P$ —its parent, and  $G$ —its grandparent. The `delete` operation locks the edge from  $G$  to  $P$  and then permanently locks  $P$  (or restarts if the attempt to lock any of them failed), modifies the edge from  $G$  to  $P$  to point at  $S$  (which unlinks both  $L$  and  $P$  from the tree), and then unlocks the edge.

#### 4.4 Design backbone details

We describe the backbone common to our two algorithms. The tree is initialized with three sentinel nodes like in the base tree, and their aggregate value is set to `identityElement(B, ⊕f)`. Next we describe the general scheme of each operation on the tree. Effectual operations (successful `insert` and `delete`) acquire the necessary locks, and then before applying the operation to the tree—they globally announce themselves including obtaining a timestamp, and update affected aggregate metadata. Failing `insert` and `delete` and `contains` operate as in the base algorithm, but then in the end verify that no ongoing operation has already announced itself and logically deleted the node they found / inserted a node with the key they have not found. Aggregate queries use the aggregate metadata throughout their traversal like sequential aggregate queries, but they also grab a timestamp in the beginning, and obtain versions of child pointers and of aggregate metadata according to this timestamp and according to announced effectual operations. Details follow.

##### 4.4.1 insert and delete operations

An `insert` and a `delete` of a key  $k$  are performed as follows:

1. Run the base tree algorithm until it is about to return **failure** or until (including) it **acquires lock/s**.

In the first case, let  $L$  be the leaf reached by the traversal,  $P$  be the parent from which it was reached, and *direction* be left or right according to which child of  $P$   $L$  is. Return failure only in the following cases:

- a. If it is an insertion (namely,  $k$  was found in  $L$ ), call `isDeleted( $L$ ,  $P$ )` (see Section 4.4.4) and return failure if it returns false.
- b. If it is a deletion (namely,  $k$  was not found), call `getValuelIfInserted( $L$ ,  $P$ , direction,  $k$ )` (see Section 4.4.4) and return failure if it returns `NOT_FOUND`.

Otherwise, start Step 1 over.

2. **Announce:** Add an *Update* object to *CurrUpdates*, including **obtaining** *ts*.
3. **Update aggregate values:**
  - a. Gather *CurrUpdates* with timestamp  $\leq ts$  into *currUpdates*.
  - b. Traverse from the root to the leaf, and for each *node* (excluding the leaf):

## XX:12 Concurrent aggregate queries

- i. Update *node.aggValue* by versioned writes, using *currUpdates* (while traversing *currUpdates*, ignore done operations and eliminate them from *currUpdates*).
  - ii. Obtain next traversal's node (left or right child) using a versioned read with *ts*.
  - iii. Eliminate out-of-range effectual operations (namely, effectual operations on keys  $< node.key$  if proceeding to the right, or with keys  $\geq node.key$  if proceeding to the left) from *currUpdates*.
4. **Apply** the operation to the target leaf area.
5. **Unannounce**: set *Update.done* and remove the *Update* from *CurrUpdates*.
6. **Finalize** deletion: modify grandparent's pointer.
7. **Release** the lock.

In more detail, an *insert( $k, v$ )* or a *delete( $k$ )* starts by running the base tree algorithm, where all reads of versioned fields are standard reads (i.e., return the value in the first version in the list) (Step 1 above). It executes until one of the following occurs: The first alternative is that the base algorithm is right about to return failure. Before doing so, it must make sure that it is still correct to return failure in our extension of the algorithm—if the operation is an insertion and  $k$  was found, it needs to make sure  $k$  is not already considered deleted by an ongoing deletion, and if it is a deletion and  $k$  was not found, it must make sure  $k$  is not already considered inserted by an ongoing insertion (see Section 4.4.4 for details). Otherwise it restarts, starting Step 1 from the top.

The other alternative is that the necessary locks are acquired, right before the relevant tree's link is modified in the base algorithm. At this point, the operation is guaranteed to succeed but is not yet linearized (which intuitively means it is not yet considered as occurred, and concurrent operations will not consider it). It will be linearized only when completing the next stage—globally announcing itself by adding an *Update* object with its details to *CurrUpdates*, including obtaining a timestamp *ts* (Step 2). In addition to the *ts* and *done* fields that were earlier mentioned, an *Update* object also includes a *leaf* field with the node that is inserted or deleted, as well as information about the tree's edge that is about to be modified: its source node (*edgeSource*), its designated new target node (*edgeTarget*) and its direction (*edgeDirection*). The operation proceeds to update the aggregate metadata in the nodes in its root-to-leaf path. When updating, it also takes into account ongoing effectual operations with timestamp equal to or less than *ts*. For that, it gathers them from *CurrUpdates* into a local copy *currUpdates* (Step 3a). This local copy is maintained throughout the traversal: done effectual operations (indicated by a *done* flag set to true) are eliminated from it, as well as effectual operations with keys which are out of the range covered by the current subtree. The update is done using versioned writes (Step 3(b)i). To traverse its root-to-leaf path, it chooses each time whether to continue to the right or left child by simply searching for  $k$ , and then uses a versioned read with timestamp *ts* to obtain the appropriate child (Step 3(b)ii).

The next step is to apply the operation to the target leaf area (Step 4): In case of an insertion, the child pointer from the parent to the target leaf (namely, the leaf at which the traversal arrived) is modified using a versioned write with *ts* to point at a new internal node which has the new node and the target leaf as its children. In case of a deletion, we add a step that was not part of the base algorithm—to apply the deletion, the target leaf is marked as deleted by setting a *marked* flag which we place in the tree leaves. A deletion is finalized by pointing the target leaf's grandparent at the leaf's sibling using a versioned write with *ts* only later—in Step 6 (which is for deletions only). After the operation is applied as detailed above, it sets its *Update* object's *done* field to true and removes the object from *CurrUpdates* (Step 5). Lastly, it releases the lock as in the base algorithm—of the target leaf's original

parent in case of an insertion or the target leaf's original grandparent in case of a deletion (Step 7). The reason for the special deletion scheme (using a mark in deletion, and ordering its steps to be marking the leaf, then setting *done* in the *Update* object and removing it from *CurrUpdates*, then modifying the grandparent's pointer) is as follows: Without the marking step, a deletion announcement would have been removed from *CurrUpdates* while its leaf is still physically in the tree, and so *contains* and a failing *insert* might consider the node as in the tree. Instead, they check the *marked* field in their *isDeleted* call to correctly determine if the node is deleted, and thanks to the *marked* field they do not miss a linearized deletion even if it has not yet physically unlinked the node from the tree. On the other hand we cannot physically unlink the node prior to unannouncing the deletion (setting its announcement's *done* field and removing the announcement from *CurrUpdates*), since if we did that then an operation *op* that calculates aggregate metadata using *CurrUpdates* (either an aggregate query, or an effectual operation that helps update aggregate metadata on behalf of other effectual operations) might gather from *CurrUpdates* an announcement of a deletion whose parent is already unlinked and mistakenly consider it in later subtrees, as *op* does not eliminate the deletion from its *currUpdates* because it does not narrow the range according to the already-unlinked parent (which is solved by using the *done* field to correctly eliminate the deletion from *currUpdates*).

#### 4.4.2 Aggregate queries

An aggregate query starts by obtaining a timestamp *ts*, and then continues like the sequential counterpart, running the traversals and *computeAnswer* calls defined for the query, but with the following modifications to the template in Figure 1 used to run the traversals (for which the template will take *ts* as an input):

1. As part of *init*, gather *CurrUpdates* with timestamp  $\leq ts$  into a local copy *currUpdates*, and if it is the first traversal ran for this query—also guarantee (in a way that will be detailed in Appendices C and D) that no effectual operations other than the gathered ones will later obtain a timestamp  $\leq ts$ .
2. After running *shouldDescendRight*, eliminate out-of-range effectual operations (namely, effectual operations with key  $< node.key$  if proceeding to the right, or with key  $\geq node.key$  if proceeding to the left) from *currUpdates*.
3. To obtain *node.left*, *node.right* and *aggValue* of the obtained left child, access these fields through versioned reads with *ts* and plug in the effect of the relevant effectual operations from *currUpdates* (while traversing *currUpdates*, ignore operations with *done*==*true* and eliminate them from *currUpdates*). The way their effect is plugged in will be detailed in Appendices C and D.

#### 4.4.3 contains operation

A *contains(k)* operation searches for *k* until reaching a leaf *L* as in the base tree algorithm, while additionally maintaining a *prev* variable holding the previous traversed node and setting a *direction* variable to left or right according to which child of *prev* *L* is. While searching, all reads of versioned *left* and *right* fields are standard reads (i.e., return the value in the first version in the list). If *L.key* == *k*, it calls *isDeleted(L, prev)* (see its description in Section 4.4.4) and if *isDeleted* returns false it returns *L*'s value else it returns NOT\_FOUND; otherwise (*L.key*  $\neq k$ )—it returns *getValueIfInserted(L, prev, direction, k)* (see its description in Section 4.4.4).

#### 4.4.4 isDeleted and getValuelInserted auxiliary methods

The `isDeleted` and `getValuelInserted` auxiliary methods are used to help guarantee the correctness of non-effectual operations, by verifying whether the key they found to be inserted or deleted is not already considered deleted or inserted respectively. These methods' objective is to give an answer that is true at some point during the calling operation's interval, for a correct linearization. All their reads of versioned fields are standard reads (i.e., return the value in the first version in the list).

The method `isDeleted` is called by `contains` and `insert` after they find the searched-for key in the tree, before they return its associated value or failure respectively, to verify that this key was not considered as already deleted by a deletion that has already taken a timestamp (thus considered linearized) but has not yet modified the tree structure. The method `getValuelInserted` is called by `contains` and `delete` after they do not find the searched-for key, before they return a negative answer, to verify that this key was not considered as already inserted by an insertion that has already taken a timestamp (thus considered linearized) but has not yet linked the new node to the tree. These methods' implementation presentation is deferred to Appendix B.3.

## 5 Analysis

### 5.1 Correctness

Our algorithms are linearizable. Effectual operations and aggregate queries are linearized by timestamp order, where aggregate queries are linearized after effectual operations with the same timestamp, and effectual operations with the same timestamp in `FastUpdateTree` are ordered according to when the timestamp that is set in their announcement is obtained. The linearization of original tree operations that do not modify the tree (`contains` and failing `insert` and `delete`) is more delicate, and is detailed in Appendix E, with a linearizability proof.

### 5.2 Time complexity

For each of the two presented algorithms, we analyze the time complexity of the new aggregate query operation, as well as the addition to the time complexity of the tree operations, incurred by our extension of the base algorithm. In our analysis we denote the number of threads in the system by  $t$ , the number of effectual operations / aggregate queries whose execution interval overlaps with that of the analyzed operation by  $\text{concUpdates}$  /  $\text{concQueries}$ , and the number of threads running effectual operations whose execution interval overlaps with that of the analyzed operation by  $\text{concUpdatingThreads}$  (this, unlike  $\text{concUpdates}$ , is bounded by  $t$ ). For an effectual operation, we denote by  $\text{effectualDepth}$  the number of nodes traversed in its last traversal in Step 1 in Section 4.4.1 (the base algorithm might carry out multiple traversals due to restarts). For an aggregate query, we denote by  $\text{queryDepth}$  the number of nodes traversed in its longest traversal (which is equal to the depth in the query's timestamp of the leaf reached in this traversal).

As shown in Appendices F and G, when embedding our algorithms with the optimizations mentioned there, `FastUpdateTree` incurs no additional asymptotic time cost on any of the base tree's operations, and performs aggregate queries in  $O(\text{queryDepth} \cdot t \cdot \min\{\text{concUpdates}, \text{concQueries}\})$  time. `FastQueryTree` performs aggregate queries in  $O(\text{queryDepth} \cdot \min\{\text{concUpdates}, \text{concQueries}\})$  time, and incurs  $O(\text{concUpdatingThreads})$  additional time on `contains` and failing `insert` and `delete`, and  $O(\text{effectualDepth} \cdot \text{concUpdatingThreads})$  on effectual operations.



## 6 Related work

Aggregation has been studied in the database community, where aggregate queries like COUNT, SUM and AVG are used to calculate an aggregate value over several database rows. Our term definitions in Section 2 are in the spirit of definitions from works on databases and consolidate them into a unified view on aggregate queries useful for addressing aggregate queries on concurrent trees. Definition 1 resembles the definition of distributive aggregate function in [16], Definition 2 resembles the definition of commutative-semigroup aggregation function in [11], and Definition 4 resembles the definition of distributive additive aggregate function in [20].

Various works have researched concurrent range queries that scan the keys in the range [30, 29, 4, 9, 5, 6, 38, 10, 15, 36, 33], an approach which is too costly for implementing efficient concurrent aggregate queries that return more succinct information about a key range. A certain kind of concurrent aggregate queries has been addressed in [32], which implements specifically a size query on sets and dictionaries. We study general concurrent aggregate queries, similarly to the independent work of [23]. Interestingly, [23] does not support a failure option for the insert and delete operations. Such failures may require traversing the tree twice, which poses additional challenges. In contrast, this work offers an improved space complexity over the current work, as they do not employ multi-versioning. Another advantage of [23] is that they also offer support for balanced trees.

Multi-versioning, where multiple versions of data structure objects are preserved when it is modified to enable queries to access old versions, has been employed in previous works which use multi-versioning to offer support for range queries (e.g., [15, 36, 33, 21]). Our multi-versioning based design for concurrent trees with aggregate queries may be used to integrate also support for non-aggregate range queries, similarly to those works. Using multi-versioning allows to reduce contention and interference between effectual operations and operations that do not modify the data structure: effectual operations leave versions for queries and do not need to help them further, and queries read old versions without interfering with newer updates. For example, the `contains` operation in our design does not need to coordinate with concurrent operations or help them throughout its traversal (it might only need to look at the announced effectual operations once when it completes its traversal). Another advantage of using multi-versioning in our design is that it enables to support the more general form of aggregate queries, composing several simple aggregate queries when several serial traversals are required (e.g., for querying the median key of a given input key range). Thanks to using multi-versioning, we may support such composite queries by executing multiple simple aggregate queries over the tree and obtaining object versions bearing the same timestamp in all of them. Using multi-versioning requires employing appropriate garbage collection to reclaim unnecessary versions, which is an orthogonal problem that could be addressed using techniques from e.g. [7, 33, 37].

## 7 Discussion

We addressed the problem of designing a concurrent efficient implementation for aggregate queries on trees. We formalized aggregate queries that could be efficiently supported on concurrent trees, presented a design that augments a concurrent binary search tree with such aggregate queries, and suggested two algorithms that implement this design, demonstrating a trade-off between aggregate query time and tree update operations. It would be interesting to further investigate if there is a better alternative in this time trade-off between effectual operations and aggregate queries, as well as in the time-space trade-off, or whether it

is impossible to achieve better solutions such as better aggregate query time complexity while still maintaining optimal time complexity for the original tree operations like our `FastUpdateTree` algorithm. Another compelling research direction is a generalization of our design to additional trees, including lock-free, balanced and internal trees.

## A Aggregate query examples

For illustration, we will take as a running example a tree of donations where a node's key is a donation amount (as an integer) and a node's value is a list of donors that gave a donation of this amount. For our metadata, we take  $B$  to be  $\mathbb{Z}$ , and define  $f(X) := \sum_{(key, value) \in X} key \cdot value.size$  (where  $key$  is a donation amount and  $value$  is a list of donors), namely,  $f$  returns the sum of subtree donations.  $\oplus_f$  and  $\ominus_f$  are simply addition and subtraction on  $\mathbb{Z}$ , and  $identityElement_{(B, \oplus_f)} = 0$ .

The simplest aggregate queries are ones concerned with a key range of the form  $(-\infty, k)$ , such as `select` and `rank`, which may be answered using a simple aggregate query that requires a single traversal. To illustrate a simple aggregate query that requires a single traversal on our running example, we inquire about the accumulated donation of all small donations up to a certain amount. For that we shall define a `sumUpTo` query whose input would be the upper donation amount bound. The `sumUpTo` query uses a single traversal to obtain the accumulated donation amount of all donations up to the given bound. Its `shouldDescendRight` returns true iff  $input \geq key$ . In this case the value of `aggValueUpToCurrentKey` is not used, but there are queries which need it to decide on the traversal's direction. For instance, `select` proceeds to the right child if  $input$  (which is the required leaf's index)  $> \text{aggValueUpToCurrentKey}$  (which counts the leaves with  $key \leq key$ , a count that includes the leaves of the current node's left subtree). The `computeAnswer` method for `sumUpTo` simply returns the aggregate value received from the traversal.

Thanks to  $f$  being subtractive, aggregate queries concerned with a key range of the form  $(lowKey, upKey)$ , such as `sum(lowKey, upKey)`, may be computed using a simple aggregate query with two independent traversals, and then having `computeAnswer` compute  $aggValueForUpKey \ominus_f aggValueForLowKey$  to get the aggregate value on the desired key range. This could be extended to unions of such key ranges by having `computeAnswer` apply  $\oplus_f$  over the results of the sub-ranges. We note that `computeAnswer` has an additional role other than combining the traversals' aggregate values into an aggregate value on the desired range: certain queries, such as average and sample variance queries, require additional calculations which `computeAnswer` takes care of. For example, to compute the average donation amount for a certain key range, we should hold different metadata: a  $\langle sum, size \rangle$  tuple, and change  $f$  to compute both sum and size. Traversals would return a tuple of  $\langle sum, size \rangle$  values as their outputted aggregate value. `computeAnswer` would compute the aggregate sum and size of the input key range by combining the traversals' outputted aggregate values using the  $\oplus_f$  and  $\ominus_f$  operators, and then divide the aggregate sum by the aggregate size to get the average.

The most general aggregate queries we define are composite queries, that require serial execution of a chain of two or more queries one after another, as each query in the chain needs an input value which is the result of a previous query in the chain. For example, we may run a `medianKeyInRange(lowKey, upKey)` query that would return the median key (or any other percentile as a generalization) of a given input key range. To implement such a query we may first run a simple aggregate query made of two independent `rank` traversals: `rank(upKey)` and `rank(lowKey)`. Let  $rankUp$  and  $rankLow$  be the aggregate values they return. Define

```

14 class InternalNode:
15     key
16     aggValue
17     left
18     right
19     leftRightLock
20 class Leaf:
21     key
22     value
23     marked

```

■ **Figure 2** Fields of the node classes

```

24 class Update:
25     timestamp
26     leaf
27     edgeSource
28     edgeTarget
29     edgeDirection
30     done
31     operationKind

```

■ **Figure 3** Fields of the Update class

this first simple aggregate query's `computeAnswer` to return  $rankLow + (rankUp - rankLow)/2$  (which is the index of the required median key). Then the second simple aggregate query should run a `select` query on the output of the first one.

## B Implementation details

### B.1 Node classes and the Update class

The fields of our node classes appear in Figure 2. An internal node does not contain a value as we handle an external tree (in which the items are placed in the leaves). A leaf class does not include an `aggValue` field since its aggregate value is constant and may be directly computed from its key and value as  $f(\{(key, value)\})$ .

Our Update class fields appear in Figure 3. `edgeSource` contains the node whose `edgeDirection` (left or right) child should be modified, namely, the grandparent node of the leaf to be deleted in case of a deletion, or the parent (which will become the grandparent after the insertion is complete) in case of an insertion. `edgeTarget` has the new child: the deleted node's sibling for a deletion, or the new internal node in case of an insertion. `leaf` is either the deleted node or the inserted new node. `operationKind` is either `delete` or `insert`.

### B.2 Versioned fields

Both child pointers and aggregate value fields in the tree nodes in our algorithms are implemented using the `VersionedField` class, whose pseudocode appears in Figure 4. For child pointers the template parameter  $T$ , which indicates the value type of the field, is a pointer to a tree node. For aggregate values,  $T$  is the aggregate type  $B$ .

```

32 class Version<T>:
33     T value
34     int timestamp
35     Version<T> next
36 class VersionedField<T>:
37     Version<T> versionListHead;
38     read():
39         return versionListHead.value
40     standardTimestampedRead():
41         version = versionListHead
42         return (version.value, version.timestamp)
43     versionedRead(ts):
44         version = versionListHead
45         while version.timestamp > ts:
46             version = version.next
47         return version.value
48     write(v, ts):
49         versionListHead = new Version<T>(v, ts, versionListHead)
50     writeIfTimestamp(lastTs, newValue, newTs):
51         firstVersion = versionListHead
52         if firstVersion.timestamp == lastTs:
53             versionListHead.CAS(firstVersion, new Version<T>(newValue, newTs, firstVersion))

```

■ **Figure 4** The VersionedField class

### B.3 isDeleted and getValuelInserted auxiliary methods

The usage of the `isDeleted` and `getValuelInserted` auxiliary methods is described in Section 4.4.4. Next we bring their implementation.

The method `isDeleted` (Figure 5) checks if the node from which the caller reached *leaf* is not permanently locked (which should be the case for an internal node that is under deletion), and if that is the case, it returns false to indicate the key was not deleted at the time the calling operation obtained *leaf* (Lines 55–56). Otherwise, an announcement of a deletion with the searched-for key is looked for in *CurrUpdates*. If such an announcement is found, the caller makes sure that it has a set timestamp (as detailed in Section 4.2) and then returns true (Lines 57–59). Else, the method returns a value according to the *marked* flag of *leaf* (Line 60).

The method `getValuelInserted` (Figure 6) checks if the edge to the found leaf is locked (separately from the other edge from *parent*, namely, this is not a permanent lock of *parent*), and if so, *CurrUpdates* is searched for an insertion announcement with *searchedKey*. If such an announcement is found, the caller makes sure that it has a set timestamp (as detailed in Section 4.2) and then returns its value (Lines 62–64). In any other case (the edge from which

```

54 isDeleted(leaf, parent):
55     if parent is not permanently locked:
56         return false
57     if a deletion del with the key leaf.key is found in CurrUpdates:
58         Guarantee del.timestamp is set
59         return true
60     return leaf.marked

```

■ **Figure 5** Pseudocode for the `isDeleted` method

```

61 getValuelfInserted(leaf, parent, leafDirection, searchedKey):
62 if the edge from parent in direction leafDirection is locked and an insertion ins with the key
    searchedKey is found in CurrUpdates:
63     Guarantee ins.timestamp is set
64     return ins.leaf.value
65 curr = the child pointer from parent in direction leafDirection
66 while curr is not a leaf:
67     if searchedKey < curr.key:
68         curr = curr.left
69     else:
70         curr = curr.right
71 if curr.key == searchedKey:
72     return curr.value
73 return NOT_FOUND

```

■ **Figure 6** Pseudocode for the `getValuelfInserted` method

the caller reached *leaf* is not locked or there is no insertion announcement with *searchedKey* in *CurrUpdates*, a traversal (similar to the search performed by `contains` in the base tree algorithm) is resumed from the current child of *parent* in direction *leafDirection* (Lines 65–73).

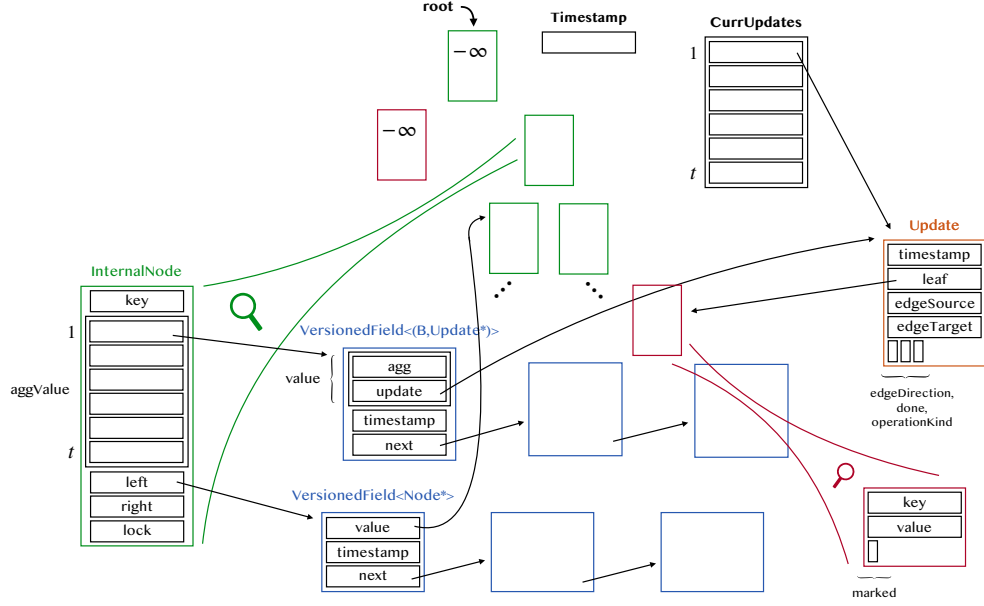
## C FastUpdateTree details

We fill in the missing details in the operations' implementation that were not covered so far. Throughout the description, we denote the id of an executing thread by *tid*. The tree holds several global members: the root node, an integer *Timestamp* field which holds the current timestamp, and the *CurrUpdates* array with a per-thread cell to be pointed at its ongoing effectual operation. See Figure 7 for a visualization of the tree parts.

We start with describing the announcement mechanism and the way operations get their timestamp. An effectual operation *op* announces itself by creating an *Update* object (Figure 3) with its details (and a special *NOT\_SET* value at the timestamp field) and pointing *CurrUpdates[*tid*]* at it. It then performs *CurrUpdates[*tid*].timestamp.CAS(NOT\_SET, Timestamp)* to set the timestamp of *op* to the current global timestamp (if it has not yet been set by a concurrent aggregate query, hence the usage of CAS which stands for compare-and-swap). This completes Step 2 in Section 4.4.1. To unannounce itself (Step 5 in Section 4.4.1) it simply sets *CurrUpdates[*tid*]* to NULL.

The operation could not have announced the *Update* object with the timestamp field already set, because had it done so, it might have announced the operation with a too old timestamp, preceding the timestamps of aggregate queries that have already executed without considering *op*. This mechanism—of first announcing, then obtaining the global timestamp value, then setting it in the announcement's timestamp field—ensures that effectual operations will not be announced with old timestamps. This together with the help by aggregate queries to set the timestamp if they observe it before it is set (as detailed next), enables aggregate queries to prevent effectual operations that they have not considered as preceding them from taking a timestamp smaller than theirs.

An aggregate query operation first of all grabs a timestamp *ts* (see Section 4.4.2), which it accomplishes by executing a fetch-and-increment on the *Timestamp* field (the fetch-and-increment atomically reads the old value—which is returned and will be the query's timestamp—and increments it by 1). In each of the operation's traversals, the operation goes through the array *CurrUpdates* and gathers the objects pointed from it that



■ **Figure 7** FastUpdateTree representation

have  $\text{timestamp} \leq ts$  into *currUpdates*, as mentioned in Step 1 in the traversals' template modifications detailed in Section 4.4.2. In all the traversals this query runs but the first one, *Update* objects with a *NOT\_SET* timestamp value are considered as future ones and are disregarded. But the first traversal is responsible to guarantee that no effectual operations other than the ones it gathers will obtain a  $\text{timestamp} \leq ts$ . To this end, while it goes through *CurrUpdates* to gather preceding effectual operations, for any object *CurrUpdates*[*i*] it encounters whose timestamp value is *NOT\_SET*, it carries out the following: It first executes *CurrUpdates*[*i*].*timestamp.CAS*(*NOT\_SET*, *Timestamp*). After this CAS, whether it was the CAS of this operation that succeeded, or it failed because a concurrent operation succeeded, *CurrUpdates*[*i*].*timestamp* is guaranteed to be set. Therefore it reads its value again, and if it is  $\leq ts$  it gathers it into *currUpdates*.

contains as well as failing insert and delete operations (namely, ones that return failure) might also need to verify that a timestamp field of an *Update* object they hold is set (in Lines 58 and 63 in the *isDeleted* and *getValuelfInserted* methods respectively). They do it similarly, by applying *CAS*(*NOT\_SET*, *Timestamp*) to the *timestamp* field of the *Update* object if its value is *NOT\_SET*.

Effectual operations announce themselves in *FastUpdateTree* without serializing on some global variable and without being aware of which effectual operations precede them; each of them performs work only on behalf of itself (unlike in *FastQueryTree* where, as will be shown in Appendix D, they need to help preceding ones). Therefore, Steps 3a and 3(b)iii in Section 4.4.1 are void in *FastUpdateTree*, and in Step 3(b)i there is no usage of *currUpdates*.

Moving on to the aggregate metadata representation, in *FastUpdateTree* an *aggValue* field in a tree node is an array with a versioned field cell per thread. The value in this versioned field is made of two variables: *agg*—the aggregate value, and *update*—each version value will also hold a pointer to the *Update* object announcing the operation that created this version. An effectual operation, with *update* being a pointer to its *Update* object, updates the



aggregate value for each node in its traversal (Step 3(b)i in Section 4.4.1) as follows. It obtains the current aggregate value for this thread (which represents the value of  $f$  on all elements currently in this node's subtree that were inserted by the current thread, before considering the current operation):  $currValue = node.aggValue[tid].read().agg$ , calculates the new aggregate value:  $newValue = currValue \oplus_f f(\{(update.leaf.key, update.leaf.value)\})$  if this is an insertion, or the same but with  $\ominus_f$  instead of  $\oplus_f$  if this is a deletion (in accordance with Properties 3 and 6); and finally performs a versioned write:  $node.aggValue[tid].write((newValue, update), update.timestamp)$ .

An aggregate query needs to calculate the aggregate metadata value of left children along its traversal (see Step 3 in Section 4.4.2), which it does using its current  $currUpdates$  as follows. Let  $left$  be the obtained left child of a node whose key is  $k$ . The query first obtains for each thread its aggregate value in  $left$  in the query's timestamp  $ts$  (namely, the value of  $f$  over elements in  $left$ 's subtree inserted by this thread in timestamps  $\leq ts$ ), and then computes the total aggregate value (over all elements in the subtree) from all the per-thread values. In detail, for each thread id  $tid$ , it performs a versioned read  $value_{tid} = left.aggValue[tid].read(ts)$ , and takes  $aggValue_{tid} = value_{tid}.agg$ , unless  $currUpdates[tid] \neq NULL$  and  $currUpdates[tid].leaf.key < k$  and  $value_{tid}.update \neq currUpdates[tid]$  (which means an ongoing effectual operation by thread  $tid$  on a leaf in  $left$ 's subtree has not yet updated its aggregate value in  $left$  so its effect should be computed using its announcement) in which case it takes  $aggValue_{tid} = value_{tid}.agg \oplus_f f(\{(currUpdates[tid].leaf.key, currUpdates[tid].leaf.value)\})$ , or the same but with  $\ominus_f$  instead of  $\oplus_f$  if  $currUpdates[tid].operationKind$  is `delete`. It then computes the total aggregate value by  $\oplus_f aggValue_{tid}$  over all  $tids$ .

To complete the algorithm's details, we explain how  $node.left$  and  $node.right$  are obtained during an aggregate query's traversal (see Step 3 in Section 4.4.2). To this end, an aggregate query with timestamp  $ts$  performs the following (we describe how to obtain  $node.left$ , the description for  $right$  is similar): It checks if there exists an announcement  $update$  in  $currUpdates$  with  $update.sourceEdge == node$  and  $edgeDirection == left$ . (As an optimization, searching  $currUpdates$  could be done only if the appropriate lock is not acquired.) If so, it takes  $update.edgeTarget$  as the desired value. Otherwise, it takes the result of a versioned read of  $node.left$  with  $ts$ .

## D FastQueryTree details

As for `FastQueryTree`, its global members are the root node and the `CurrUpdates` queue. Effectual operations enqueue themselves to the queue to announce themselves (Step 2 in Section 4.4.1), and the queue's enqueue operation takes care of setting the operation's timestamp to be the timestamp of the last operation in the queue +1. The wait-free queue of [22], which is based on [28], may be used as a basis for the `CurrUpdates` queue. It could be easily extended to contain consecutive timestamps in the nodes (by setting the timestamp field to the value in the current tail node +1 and trying to enqueue this node using a CAS). It could also be extended to enable the removal of a node that is not in the head of the queue, as should be done to unannounce an effectual operation (Step 5 in Section 4.4.1).

Aggregate queries obtain a timestamp by reading the timestamp in the `Update` object in the current last node of the queue (which is the last one to be added to the queue). To gather announcements with timestamp  $\leq ts$  from `CurrUpdates` into a local  $currUpdates$  (by either an effectual operation with timestamp  $ts$  in Step 3a in Section 4.4.1, or an aggregate query with timestamp  $ts$  as mentioned in Step 1 in the traversals' template modifications detailed in Section 4.4.2), the operation goes through `CurrUpdates`'s nodes and copies their

announcements' pointers starting from the first (oldest) node until reaching a node with timestamp  $> ts$ , whose announcement is excluded from the gathered announcements. In a similar fashion, when `contains` and failing `insert` and `delete` operations traverse *CurrUpdates* (in Lines 57 and 62 in the `isDeleted` and `getValueIfInserted` methods respectively), they in fact first obtain the current global timestamp by reading the timestamp in the *Update* object in the current last node of the queue, and then go through *CurrUpdates*'s nodes and copy their announcements' pointers starting from the first node until reaching a node with timestamp greater than the timestamp they obtained (otherwise, they might continue going through *CurrUpdates* forever as more and more concurrent operations enqueue their announcements).

Unlike in *FastUpdateTree*, in *FastQueryTree* an aggregate query with timestamp  $ts$  needs to do nothing to guarantee that effectual operations will not later obtain a timestamp  $\leq ts$ , as after it obtains the timestamp  $ts$ , any later operation will be allocated a bigger timestamp. Hence, guaranteeing that in the first query's traversal as mentioned in Step 1 in Section 4.4.2 is void in *FastQueryTree*. Also, for `contains` and failing `insert` and `delete` operations, guaranteeing a timestamp in an *Update* object they obtained from *CurrUpdates* is set (in Lines 58 and 63 in the `isDeleted` and `getValueIfInserted` methods respectively) is void in *FastQueryTree*, since nodes are enqueued to the *CurrUpdates* queue with a set timestamp.

We move on from the timestamp and announcement handling in *FastQueryTree* to its aggregate value access scheme. To update the aggregate value of a node to reflect an effectual operation with timestamp  $ts$  (Step 3(b)i in Section 4.4.1), it needs to add a version with timestamp  $ts$  and the appropriate aggregate value to the linked list of versions in *node.aggValue*. However, it must first update the aggregate value on behalf of ongoing effectual operations with smaller timestamps. To update the aggregate metadata on behalf of those effectual operations that have not yet updated it and on behalf of itself, it performs the following until going through *currUpdates* (which is ordered by timestamp like the *CurrUpdates* queue, and includes the current operation itself in the last announcement) to its end: executing a standard timestamped read of *node.aggValue* to obtain its (*lastValue*, *lastTs*); continuing to go through *currUpdates* (from the last point reached at the previous iteration) until reaching an object *update* with a timestamp greater than *lastTs*; calculating the new aggregate value taking *update* into account by computing  $newValue = lastValue \oplus_f f(\{(update.leaf.key, update.leaf.value)\})$  if *update.operationKind* is `insert`, or the same but with  $\ominus_f$  instead of  $\oplus_f$  if it is `delete`; and trying to perform a thread-safe write of *newValue* with timestamp *update.timestamp* given the current version list's timestamp is still *lastTs* (namely, running *node.aggValue.writeIfTimestamp(lastTs, newValue, update.timestamp)*, for which there is no need to check for failure as it fails only if another thread performed the same write).

An aggregate query with timestamp  $ts$  needs to calculate the aggregate metadata value of left children along its traversal (see Step 3 in Section 4.4.2), which it does using its current *currUpdates* as follows. Let *left* be the obtained left child of a node whose key is  $k$ . The query first executes a standard timestamped read of *left.aggValue* to obtain its current (*value*, *currTs*). If *currTs*  $== ts$  it returns *value*; if *currTs*  $> ts$ , it executes a versioned read of *left.aggValue* with  $ts$  and returns the returned value. Otherwise, it needs to plug in the effect of relevant ongoing operations on top of *value*. For that, it goes through *currUpdates*, and for each object *update*, if *update.leaf.key*  $< k$  and *update.timestamp*  $> ts$  then it calculates the new aggregate value taking *update* into account by computing  $value = value \oplus_f f(\{(update.leaf.key, update.leaf.value)\})$  if *update.operationKind* is `insert`, or the same but with  $\ominus_f$  instead of  $\oplus_f$  if it is `delete`. After exhausting *currUpdates*, *value* has the required aggregate metadata value. As for how an aggregate query obtains *node.left* and

*node.right* during its traversals, it does so the same as in `FastUpdateTree` (see details in the end of Appendix C).

## E Correctness

`FastUpdateTree` and `FastQueryTree` are linearizable. In this section we detail the linearization points of their operations, and use them to prove linearizability.

### E.1 Linearization Points

Effectual operations are ordered by their timestamps. For `FastUpdateTree` we set the linearization point of an effectual operation to be at the moment the timestamp that is eventually set in its *Update* announcement using a CAS is obtained (by either the operation itself or a helping aggregate operation); for `FastQueryTree` we set it at the linearization point of the enqueue of its *Update* object to the *CurrUpdates* queue (which is, by definition of linearization points of the underlying queue [22], when the node with this *Update* object is physically linked to the queue). Aggregate queries are also ordered by timestamp, such that they occur after effectual operations of the same timestamp. We accordingly set their linearization points to be at the fetch-and-increment of the *Timestamp* field in `FastUpdateTree`, and in `FastQueryTree` – when they obtain in their beginning the pointer to the last node of the *CurrUpdates* queue, from which they obtain the global timestamp.

Lastly, contains and failing insert and delete operations are linearized according to their last `isDeleted` or `getValueIfInserted` call (`contains` always has only one such call) as follows. Let *op* be such an operation on a key *k*, *m* be the last method it called (`isDeleted` or `getValueIfInserted`), and *L*—the leaf node *op* passes to *m*. *op* is linearized at a linearization point as follows:

1. If *m* is `isDeleted`, and it returns false: a moment within *op*'s interval which is after the linearization point of the insertion of *L* and before the linearization point of the deletion of *L* (if such a deletion occurs in the execution).
2. If *m* is `isDeleted`, and it returns true: a moment within *op*'s interval which is after the linearization point of the deletion of *L* and before the linearization point of the next insertion of the key *k* (if such an insertion occurs in the execution).
3. If *m* is `getValueIfInserted`, and it returns the value that is in a node *N* (*N* is *ins.leaf* if *m* returns in Line 64, or *curr*'s value when Line 72 is executed if *m* returns in Line 72): a moment within *op*'s interval which is after the linearization point of the insertion of *N* and before the linearization point of the deletion of *N* (if such a deletion occurs in the execution).
4. If *m* is `getValueIfInserted`, and it returns `NOT_FOUND`: a moment within *op*'s interval in which *k* is logically not found in the tree; namely, if an insertion of *k* to the tree has been linearized at any point in the execution by the time *op* returns, then a moment within *op*'s interval in which the last operation on *k* to be linearized was a deletion, otherwise any moment within *op*'s interval is fine.

We should show the last linearization points are well-defined, as we shall do in Lemma 7.

### E.2 Linearizability Proof

We prove that our algorithms are linearizable using the equivalent definition of linearizability that is based on linearization points (see [31, Section 7] and the atomicity definition in [26]).

We need to show that (1) each linearization point occurs within the operation's execution time, and that (2) ordering an execution's operations (with their results) according to their linearization points forms a legal sequential history. (1) is pretty straightforward for the linearization points we defined, although the linearization point definition for `contains` and failing `insert` and `delete` operations does require proving that they are well defined, as we do in Lemma 7. We prove (2) in Claim 8.

► **Lemma 7.** *The linearization points of `contains` and failing `insert` and `delete` operations as defined in Appendix E.1 are well defined. Namely, a moment as described in them indeed occurs within `op`'s interval.*

**Proof.** We use the same notations as in Appendix E.1. Additionally, let  $P$  be the parent node from which `op` reached  $L$  and which it passes to  $m$ .

We start with the case of  $m$  being `isDeleted` which returns false. Let  $t$  be the moment in which `op` obtains  $L$  from its parent. The `insert` operation that inserts  $L$  is already linearized at time  $t$ , because the fact that `op` reaches  $L$  in its traversal means that this `insert` operation must have already physically linked  $L$  to the tree, which it does only after its linearization point. It remains to show that the `delete` operation of  $L$  (if such a deletion occurs in the execution) is not yet linearized at time  $t$ . We look at two cases according to the line at which  $m$  returns. If it returns in Line 56, it observes  $P$  when it is not permanently locked. We observe that in the base tree algorithm, a tree node may be pointed only from its current parent in the tree and permanently-locked nodes that were its parent in the past (before they were unlinked from the tree when the node's then-sibling was deleted). When a node is deleted, the single node pointing to it that was not permanently locked becomes permanently locked as well, namely, all nodes pointing to a deleted node are permanently locked. Back to our proof, this implies that  $L$  could not be deleted when the not-permanently-locked  $P$  pointed to it. Alternatively,  $m$  returns in Line 60. This means that  $m$  observed no announcement of a deletion of  $k$  in `CurrUpdates`, and then observed `L.marked==false`. But from the moment the deletion of  $L$  is linearized, either its announcement is found in `CurrUpdates` or `L.marked==true` (as the deletion's linearization point occurs when the announcement is in `CurrUpdates`, and the announcement is removed from `CurrUpdates` only after `L.marked` is permanently set to true). So  $L$ 's removal was not linearized when `op` obtained  $L$ .

Next, for the case of  $m$  being `isDeleted` which returns true: The linearization point of the deletion of  $L$  obviously happens before the linearization point of the next insertion of the key  $k$  (if such an insertion occurs in the execution). `op` returns true only after the deletion of  $L$  is linearized. In addition, when `op` is invoked,  $L$  is still reachable in the tree (as `op` reaches it), so it is impossible that  $L$  was removed and  $k$  was re-inserted before `op`'s invocation.

In the case of  $m$  being `getValueIfInserted` which returns `N.value`: The linearization point of the insertion of  $N$  obviously happens before the linearization point of the deletion of  $N$  (if such a deletion occurs in the execution). `op` returns only after the insertion of  $N$  is linearized as it either guarantees its linearization in Line 63 or reaches  $N$  through a traversal which means  $N$  has been physically linked to the tree which happens only after its insertion's linearization. On the other direction, we show that `op` is invoked before the deletion of  $N$  is linearized: We observe that two effectual operations on the same key could not be announced in `CurrUpdates` at the same time, due to the parent's lock they hold while being announced. Therefore, in case `op` observes an announcement regarding the insertion of  $N$  then at that moment  $N$ 's deletion could not have yet happened; and in case `op` reached  $N$  through a traversal, then  $N$  has been physically linked to the tree beforehand, and certainly after the invocation of `op` (otherwise `op` would have reached  $N$  during its traversal prior to calling `getValueIfInserted`, which it did not—recall  $m$  was called after not finding  $k$ ), and at that

moment  $N$ 's insertion was ongoing so a deletion of  $N$  could not have been announced yet in *CurrUpdates* so it was not yet linearized.

The last case revolves around  $m$  being `getValuelfInserted` which returns `NOT_FOUND`. Assuming that an insertion of  $k$  has been linearized at any point in the execution by the time  $op$  returns, we need to show that there is a moment during  $op$ 's interval in which the last operation on  $k$  to be linearized was a deletion. Assume, for sake of contradiction, that an insertion of a node  $N$  with the key  $k$  has been linearized prior to  $op$ 's invocation, and a deletion of  $N$  is not linearized until  $op$  returns. Thus, from the moment  $N$  is physically linked to the tree it cannot be physically unlinked during  $op$ 's interval.  $N$  must not yet be linked when  $op$  obtains  $L$ , otherwise  $op$  would reach  $N$  that has the searched-for key  $k$  instead of reaching  $L$  that has another key. This means that at the invocation moment of  $op$ ,  $P$  must be locked for inserting  $k$ . Thus,  $N$  must be linked as a sibling of  $L$ , with a new internal node becoming  $P$ 's child instead of  $L$ . From the moment  $N$  is linked, it remains in  $P$ 's subtree because the insertion and deletion operations of the base tree preserve ancestor-descendant relations for non-deleted nodes (so nodes may be added or removed between  $P$  and  $N$ , but  $N$  remains  $P$ 's descendant the whole time). And it must be physically linked prior to  $m$ 's traversal in Lines 65–73 because before the traversal, when Line 62 is executed, the *Update* object announcing  $N$ 's insertion is already not found in *CurrUpdates* anymore. Hence,  $m$ 's traversal from  $P$  must reach  $N$ , which is a contradiction to returning `NOT_FOUND`. ◀

▷ **Claim 8.** Consider a sequential history formed by ordering an execution's operations (with their results) according to their linearization points defined in Appendix E.1. Then operation results in this history comply with the sequential specification of a dictionary.

**Proof.** The abstract state of the dictionary includes all keys on which the last operation was a successful `insert`, with the value it inserted. We will show that all operations consider this as the state of the dictionary in their linearization point and return a suitable result.

Starting with `contains` and failing `insert` and `delete` operations, from their linearization point definition it immediately follows that their result complies with a dictionary's sequential specification:

1. The linearization point of both `contains( $k$ )` that returns a node's value after calling `isDeleted` and a failing `insert( $k, v$ )` is defined to be after the linearization point of the insertion of the found key and before its deletion, namely, in a moment  $k$  is in the dictionary with the found value.
2. The linearization point of `contains( $k$ )` that returns `NOT_FOUND` after calling `isDeleted` is after the linearization point of the deletion of the node found with  $k$  and before the linearization point of the next insertion of the key  $k$ , namely, in a moment  $k$  is not found in the dictionary with the found value.
3. The linearization point of `contains( $k$ )` that returns a node's value after calling `getValuelfn-inserted` is after the linearization point of the insertion of the newly-inserted node whose value is returned and before the linearization point of its deletion, namely, in a moment  $k$  is in the dictionary with the found value.
4. The linearization point of both `contains( $k$ )` that returns `NOT_FOUND` after calling `getValuelfn-inserted` and a failing `delete( $k$ )` is defined to be when  $k$  is not found in the dictionary.

For a successful `delete( $k$ )`, in its linearization point, a node  $L$  with  $k$  is physically found in the tree, and the deletion operation has permanently locked  $L$ 's parent. So there are no other operations on  $k$  currently announced in *CurrUpdates* (because they need to grab the

lock first), which means any operation on  $k$  that has already linearized has physically applied itself to the tree, and any operation on  $k$  that is not yet linearized has not physically applied itself to the tree. Therefore, as  $k$  is physically in the tree it means that the last effectual operation on  $k$  was an insertion of  $L$ .

For a successful  $\text{insert}(k, v)$ , in its linearization point, a node with  $k$  is not physically found in the tree, and the insertion operation is holding a lock of the edge where the key should be inserted. So there are no other operations on  $k$  currently announced in *CurrUpdates* (because they need to grab the lock in the same location first), which means any operation on  $k$  that has already linearized has physically applied itself to the tree, and any operation on  $k$  that is not yet linearized has not physically applied itself to the tree. Therefore, as  $k$  is physically not in the tree it means that the last effectual operation on  $k$  was a deletion.

It remains to prove that an aggregate query *agg* returns the correct result. For that we need to show that it obtains the logical values of the child pointers and the aggregate metadata at the moment of its linearization. It obtains the value from the relevant tree node together with the effect of relevant effectual operations in *currUpdates*. Let  $N$  be the current node whose child pointer or aggregate value *agg* obtains during its traversal. *agg* plugs in only the effect of effectual operations which precede it and are not yet reflected in the current field value in  $N$  thanks to the following observations regarding *agg*'s *currUpdates* local variable at the current moment: First, all effectual operations in *currUpdates* are linearized before *agg*. That is because as detailed in Step 1 in Section 4.4.2, *agg* gathers ongoing announcements regarding effectual operations with  $ts \leq \text{agg.ts}$ , which are the ones that are linearized before it. Second, the leaves of all effectual operations in *currUpdates* are in  $N$ 's subtree. That is thanks to eliminating out-of-range effectual operations throughout the traversal as detailed in Step 2 in Section 4.4.2, and additionally thanks to ignoring effectual operations with *done*==*true*—so that in case a deleted node's parent was unlinked and was not taken into account while narrowing the range of *currUpdates*, the deleted node will not be mistakenly taken into account when calculating aggregate metadata of later nodes in the traversal. Third, if an operation that is still in *currUpdates* already updated a certain affected metadata or a child pointer, it will not be updated again on behalf of the same operation (thanks to a unique timestamp for effectual operations in *FastQueryTree*, and to the examination of the *update* pointer field in *FastUpdateTree*). We also observe that there are no ongoing effectual operations with  $ts \leq \text{agg.ts}$  that *agg* does not gather, thanks to *agg* guaranteeing that no effectual operations other than the gathered ones will later obtain a timestamp  $\leq \text{agg.ts}$  as mentioned in Step 1 in Section 4.4.2. Lastly, there is no effectual operation preceding *agg* that is not in its *currUpdates* and has not yet updated the tree (both the affected metadata values and the target area), because effectual operations leave *CurrUpdates* only after updating all relevant fields in versions with their timestamp.

◀

## **F** Time complexity

Using the notations brought in Section 5.2, we next analyze the time complexity of each of our two algorithms.

### **F.1 FastUpdateTree complexity**

In *FastUpdateTree*, searching *CurrUpdates* in the *isDeleted* and *getValueIfInserted* methods costs  $O(t)$  time. However, this cost could be eliminated by setting aside some bits in each internal node's left-child lock as well as right-child lock (the locks that are placed together in



the same memory word) for indicating the tid of the locking thread and modifying the locking mechanism of the base algorithm accordingly (to also set the tid by the same CAS that acquires a lock). Then `isDeleted` and `getValueIfInserted` could avoid searching *CurrUpdates* and instead directly check the cell of the specified locking thread. Then we get zero additional asymptotic time cost on `contains` and failing `insert` and `delete`.

As for effectual operations, `FastUpdateTree` incurs  $O(\text{effectualDepth} \cdot \text{concUpdates})$  additional time on them: They pay  $O(1)$  time for announcing and unannouncing themselves (Steps 2 and 5 in Section 4.4.1) as well as for applying the operation (Steps 4 and 6). To update affected aggregate values (Step 3), they go through  $O(\text{effectualDepth})^1$  nodes, and for each, perform Step 3(b)i that costs  $O(1)$  time and Step 3(b)ii whose versioned read costs  $O(\text{concUpdates})$  time. (Steps 3a and 3(b)iii in Section 4.4.1 are void in `FastUpdateTree` where effectual operations do not help other effectual operations and do not use *currUpdates*.) The `FastUpdateTree` algorithm could however be optimized to avoid any additional asymptotic time cost on effectual operations, by keeping record of the traversed nodes during Step 1, so that the recorded nodes will be traversed in Step 3b and the versioned reads in Step 3(b)ii will be avoided.

An aggregate query in `FastUpdateTree` takes  $O(\text{queryDepth} \cdot t \cdot \text{concUpdates})$  time, according to the following calculation. We assume all `shouldDescendRight` and `computeAnswer` methods defined by the query take  $O(1)$  time, otherwise their cost should be counted as well. Obtaining a timestamp for the query takes  $O(1)$  time. Each traversal ran by the query goes through *CurrUpdates* in its initialization (in Step 1 in Section 4.4.2) in  $O(t)$  time. The traversal then performs  $O(\text{queryDepth})$  iterations, where each iteration costs as follows, based on the traversals' template modifications detailed in Section 4.4.2: Going through *currUpdates* in Step 2 takes  $O(t)$  time. As for Step 3—obtaining child nodes takes  $O(t)$  time to go through *currUpdates* (to look for an ongoing update of the child pointer, which could be spared if indicating the locking thread's tid in internal nodes' locks as suggested above, but shaving the  $t$  factor here anyhow does not improve the time cost of an aggregate query as this is not its dominant factor) and additional  $O(\text{concUpdates})$  time to perform a versioned read of the pointer, and computing the aggregate value of the obtained left child takes  $O(t \cdot \text{concUpdates})$  time because for each thread the query performs a versioned read on its cell in the metadata array (and also checks the thread's cell in *concUpdates* in  $O(1)$  time).

## F.2 FastQueryTree complexity

In `FastQueryTree`, searching the *CurrUpdates* queue in the `isDeleted` and `getValueIfInserted` methods costs  $O(\text{concUpdatingThreads})$  time. In fact, each traversal of *CurrUpdates* by any operation takes  $O(\text{concUpdatingThreads})$  time because each thread may have a single ongoing effectual operation at any moment, and as the queue is always traversed from the head up to a pre-obtained timestamp, only operations that were ongoing when the timestamp was obtained may be traversed. This means that `FastQueryTree` incurs  $O(\text{concUpdatingThreads})$  additional time on `contains`. As for `insert` and `delete`, they might call `isDeleted` and `getValueIfInserted`

<sup>1</sup> An effectual operation goes through  $O(\text{effectualDepth})$  nodes during the traversal in Step 3b, because after it reaches a leaf with key  $k$  and locks the edge from its parent in case of `insert` or permanently locks the parent in case of `delete` (in Step 1), searching for  $k$  again from the root (in Step 3b) may result in traversing through less nodes due to concurrent deletions, but not through additional nodes (which are impossible in the current operation's path because internal nodes are inserted only between a leaf and its parent, and the edge from this path's penultimate node to the leaf may not be modified during the operation's execution as it either the edge's lock is acquired or the penultimate node is permanently locked).

multiple times in Step 1 in Section 4.4.1, as they may restart this step multiple times. However, it is easy to eliminate the additional cost from all calls except the first one by having the first call record the *Update* object it found in *CurrUpdates* (or record it did not find an object with the requested key), and the following calls could avoid the search and just use what it recorded instead. This way, *FastQueryTree* will incur  $O(\text{concUpdatingThreads})$  additional time on failing insert and delete as well.

For an effectual operation, other than the  $O(\text{concUpdatingThreads})$  additional time due to possibly calling *isDeleted* and *getValueIfInserted*, it pays the following additional time. Announcing and unannouncing itself (Steps 2 and 5 in Section 4.4.1) by enqueueing and removing respectively from the *CurrUpdates* queue based on the wait-free queue of [22] takes  $O(t)$  time, which could be optimized to  $O(\text{concUpdatingThreads})$  using techniques of [3] as mentioned by [22]. Applying the operation (Steps 4 and 6) costs  $O(1)$  time. To update affected aggregate values (Step 3), it first goes over *CurrUpdates* to gather announcements into *currUpdates* in  $O(\text{concUpdatingThreads})$  time (Step 3a), then goes through  $O(\text{effectualDepth})$  nodes (for the same argument that was made above for *FastUpdateTree*), and for each node: It performs Steps 3(b)i and 3(b)iii in  $O(\text{concUpdatingThreads})$  time due to going over *CurrUpdates*. In addition, it runs Step 3(b)ii in  $O(\text{concUpdates})$  time due to the versioned read, but this step could be eliminated together with its cost, by keeping record of the traversed nodes during Step 1, so that the recorded nodes will be traversed in Step 3b.

An aggregate query in *FastQueryTree* takes  $O(\text{queryDepth} \cdot \text{concUpdates})$  time, according to the following calculation (assuming *shouldDescendRight* and *computeAnswer* methods defined by the query take  $O(1)$  time). Obtaining a timestamp for the query takes  $O(1)$  time. Each traversal ran by the query goes through *CurrUpdates* in its initialization (in Step 1 in Section 4.4.2) in  $O(\text{concUpdatingThreads})$  time. The traversal then performs  $O(\text{queryDepth})$  iterations, where each iteration costs  $O(\text{concUpdates})$  because this is the cost of Steps 2 and 3 in the traversals' template modifications detailed in Section 4.4.2, due to going through *currUpdates* in  $O(\text{concUpdatingThreads})$  time which is bounded by  $O(\text{concUpdates})$ , and performing versioned reads in  $O(\text{concUpdates})$  time.

In Appendix G we suggest further optimizations that reduce the *concUpdates* factor in the time complexity of aggregate queries down to  $\min\{\text{concUpdates}, \text{concQueries}\}$ .

## **G** Algorithm optimizations

In addition to the optimizations mentioned in Appendix F, we suggest several more ways to optimize our algorithms.

Aggregate queries perform versioned reads of both child pointers and aggregate metadata, and each such read costs  $O(\text{concUpdates})$  time. To reduce the *concUpdates* factor in aggregate queries time complexity down to  $\min\{\text{concUpdates}, \text{concQueries}\}$ , we may apply the following optimizations.

In *FastUpdateTree*, instead of writing to a versioned field by adding a new version to the head of its linked list of versions, we may update the current version in case of a write with the same timestamp, to spare the creation of redundant versions that no aggregate query is going to need. The problem that might be then created is that aggregate queries that run concurrently with the effectual operation might not know whether the operation is already reflected in the current tree data or not and they should plug in its effect on their own. For left and right child pointers, aggregate queries may simply solve the problem by taking into account an announced effectual operation's new edge target if it has an edge source they are currently examining, since that is for sure the correct value. But for the aggregate metadata

field, we need to make some changes to enable an aggregate query to correctly determine if an announced operation has already updated the examined metadata or not: We add an *updateNum* array to the tree, with a cell per thread holding the last effectual operation id of the thread (a field read and written only by the thread, incremented by the thread on each effectual operation it runs); *updateNum* and *valueToSet* fields to the *Update* object; and *updateNum* and *lastUpdateNum* fields to the *VersionedField* object of the aggregate metadata (while having its value field contain only the aggregate value and not an update pointer as in the initial design, which solves also the problem of keep holding a pointer to the *Update* object after it is no longer necessary, thus delaying its reclamation). We have an effectual operation update the metadata as follows (where *update* is its *Update* object, *version* is the *VersionedField* object of the aggregate metadata it currently updates, and *newValue* is the new value to be written to the version—reflecting the current value combined with the effect of the current effectual operation):

```

74 update.valueToSet = newValue
75 version.updateNum = update.updateNum
76 version.value = update.valueToSet
77 version.lastUpdateNum = version.updateNum

```

We have aggregate queries run the following to obtain the correct metadata value (where *update* is the *Update* object obtained from its *currUpdates* as it has the same key as the node whose aggregate value is calculated and *version* is the *VersionedField* object of the node for which the aggregate metadata is calculated):

```

78 versionValue = version.value
79 if version.updateNum < update.updateNum:
80     return < versionValue with the effect of update plugged in >
81 updateValue = update.valueToSet
82 if version.lastUpdateNum < update.updateNum: return updateValue
83 return version.value

```

In *FastQueryTree*, we suggest to apply two changes in order to be able to avoid creating versions not required by any ongoing aggregate query, thus reducing both the time complexity of aggregate queries and the tree's space consumption. First, aggregate queries will announce themselves (specifically, just their timestamp) in a global *QueryTimestamps* wait-free queue, similar to the *CurrUpdates* queue holding the effectual operations' announcements. An effectual update will gather all announced query timestamps smaller than its own timestamp (similarly to the way ongoing effectual operations' announcements are gathered from *CurrUpdates*). While updating aggregate values throughout its update traversal (on behalf of both ongoing effectual operations with smaller timestamp and itself), it will not create a version for each effectual operation, but rather only one version for each batch of effectual operations with timestamps *leq* that of an ongoing aggregate query. This change will add  $O(\text{concUpdatingQueries})$  to the complexity of the aggregate query, where *concUpdatingQueries* denotes the number of threads running aggregate queries whose execution interval overlaps with that of the analyzed operation (and is bounded by both *concQueries* and *t*). The second change is writing in existing versions instead of in new ones, even though this is a write with a newer timestamp, as long as no ongoing aggregate query has timestamp  $\geq$  the existing one and  $<$  the new one (according to the announced query timestamps), because there is no need to keep versions with timestamps that no queries will look at. This could be done using double-width CAS or a CAS of an object with the two fields - the value and the timestamp, in order for aggregate queries to obtain the value associated with the correct timestamp.

Additionally, to make aggregate queries run faster (though without effect on their asymptotic time complexity), they may obtain left and right pointers throughout their traversals using versioned reads only without plugging in the effect of effectual operations in *currUpdates*, which will save searching *currUpdates*. This is fine for all nodes except for at the target area, where current effectual operations must be taken into account because they might have not yet linked a node that the query should reach, or have not yet unlinked a node it should not reach. Thus, an aggregate query should plug in the effect of operations in *currUpdates* to the target area, either by retroactively going back up to the leaf's grandparent after reaching the leaf and re-calculating the child pointers, or by checking in advance on each node during the traversal whether the next-next child is a leaf and if so search *currUpdates* for relevant operations.

---

## References

---

- 1 C++ SGI rope, 2008. URL: <https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.3/a00223.html>.
- 2 Lapce—Lightning-fast And Powerful Code Editor, 2024. URL: <https://github.com/lapce/lapce>.
- 3 Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast. In *STOC*, 1995. doi:10.1145/225058.225271.
- 4 Maya Arbel-Raviv and Trevor Brown. Harnessing epoch-based reclamation for efficient range queries. In *PPoPP*, 2018. doi:10.1145/3178487.3178489.
- 5 Hillel Avni, Nir Shavit, and Adi Suissa. Leaplist: lessons learned in designing tm-supported range queries. In *PODC*, 2013. doi:10.1145/2484239.2484254.
- 6 Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. Kiwi: A key-value map for scalable real-time analytics. In *PPoPP*, 2017. doi:10.1145/3018743.3018761.
- 7 Naama Ben-David and E Guy. Space and time bounded multiversion garbage collection. In *DISC*, 2021. doi:10.4230/LIPIcs.DISC.2021.12.
- 8 Hans-J Boehm, Russ Atkinson, and Michael Plass. Ropes: An alternative to strings. *Software: Practice and Experience*, 25(12), 1995. doi:10.1002/SPE.4380251203.
- 9 Trevor Brown and Hillel Avni. Range queries in non-blocking k-ary search trees. In *OPODIS*, 2012. doi:10.1007/978-3-642-35476-2\_3.
- 10 Bapi Chatterjee. Lock-free linearizable 1-dimensional range queries. In *ICDCN*, 2017. doi:10.1145/3007748.3007771.
- 11 Sara Cohen, Werner Nutt, and Yehoshua Sagiv. Rewriting queries with arbitrary aggregation functions using views. *TODS*, 31(2), 2006. doi:10.1145/1138394.1138400.
- 12 Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.
- 13 Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *ASPLOS*, 2015. doi:10.1145/2694344.2694359.
- 14 Tudor Alexandru David, Rachid Guerraoui, Tong Che, and Vasileios Trigonakis. Designing ASCY-compliant concurrent search data structures. Technical report, EPFL, 2014.
- 15 Panagioti Fatourou, Elias Papavasileiou, and Eric Ruppert. Persistent non-blocking binary search trees supporting wait-free range queries. In *SPAA*, 2019. doi:10.1145/3323165.3323197.
- 16 Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Mining and Knowledge Discovery*, 1(1), 1997. doi:10.1023/A:1009726021843.
- 17 Maurice Herlihy. Wait-free synchronization. *TOPLAS*, 13(1), 1991. doi:10.1145/114005.102808.
- 18 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *TOPLAS*, 12(3), 1990. doi:10.1145/78969.78972.

- 19 Grant Jenks. Python sorted containers, 2019. URL: <https://grantjenks.com/docs/sortedcontainers>.
- 20 Marcus Jürgens. *Index structures for data warehouses*. Springer, 2002. doi:10.1007/3-540-45935-9.
- 21 Tadeusz Kobus, Maciej Kokociński, and Paweł T Wojciechowski. Jiffy: A lock-free skip list with batch updates and snapshots. In *PPoPP*, 2022. doi:10.1145/3503221.3508437.
- 22 Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueueers and dequeuers. In *PPoPP*, 2011. doi:10.1145/1941553.1941585.
- 23 Ilya Kokorin, Dan Alistarh, and Vitaly Aksenov. Wait-free trees with asymptotically-efficient range queries. *arXiv preprint*, 2023. arXiv:2310.05293.
- 24 Raph Levien. Xi-editor, 2022. URL: <https://xi-editor.io>.
- 25 Joaquín M López-Muñoz. Boost.MultiIndex Ranked indices reference, 2015. URL: [https://www.boost.org/doc/libs/1\\_75\\_0/libs/multi\\_index/doc/reference/rnk\\_indices.html](https://www.boost.org/doc/libs/1_75_0/libs/multi_index/doc/reference/rnk_indices.html).
- 26 Nancy A Lynch. *Distributed algorithms*. Elsevier, 1996.
- 27 Riccardo Mazzarini. Crop crate, 2024. URL: <https://docs.rs/crop/0.3.0/crop/index.html>.
- 28 Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, 1996. doi:10.1145/248052.248106.
- 29 Jacob Nelson-Slivon, Ahmed Hassan, and Roberto Palmieri. Bundling linked data structures for linearizable range queries. In *PPoPP*, 2022. doi:10.1145/3503221.3508412.
- 30 Erez Petrank and Shahar Timnat. Lock-free data-structure iterators. In *DISC*, 2013. doi:10.1007/978-3-642-41527-2\_16.
- 31 Gal Sela, Maurice Herlihy, and Erez Petrank. Linearizability: A typo. *arXiv preprint*, 2021. arXiv:2105.06737.
- 32 Gal Sela and Erez Petrank. Concurrent size. *PACMPL*, 6(OOPSLA2), 2022. doi:10.1145/3563300.
- 33 Gali Sheffi, Pedro Ramalhete, and Erez Petrank. EEMARQ: Efficient lock-free range queries with memory reclamation. In *OPODIS*, 2022. doi:10.4230/LIPICS.OPODIS.2022.5.
- 34 Daniel Stutzbach. blist: an asymptotically faster list-like type for Python, 2010. URL: <http://stutzbachenterprises.com/blist>.
- 35 Nathan Vegdahl. Rokey crate, 2023. URL: <https://crates.io/crates/rokey/0.6.3>.
- 36 Yuanhao Wei, Naama Ben-David, Guy E Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. Constant-time snapshots with applications to concurrent data structures. In *PPoPP*, 2021. doi:10.1145/3437801.3441602.
- 37 Yuanhao Wei, Guy E Blelloch, Panagiota Fatourou, and Eric Ruppert. Practically and theoretically efficient garbage collection for multiversioning. In *PPoPP*, 2023. doi:10.1145/3572848.3577508.
- 38 Kjell Winblad, Konstantinos Sagonas, and Bengt Jonsson. Lock-free contention adapting search trees. In *SPAA*, 2018. doi:10.1145/3210377.3210413.