Brief Announcement: Concurrent Aggregate Queries

Gal Sela
Technion, Haifa, Israel

Erez Petrank
Technion, Haifa, Israel

Abstract

Concurrent data structures serve as fundamental building blocks for concurrent computing. Many concurrent counterparts have been designed for basic sequential algorithms; however, one notable omission is a concurrent tree that supports aggregate queries. Aggregate queries essentially compile succinct information about a range of data items. 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 lock-based trees. We present two algorithms implementing this design, where one optimizes for tree update time, while the other optimizes for aggregate query 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.DISC.2024.53

Related Version *Full Version*: `https://arxiv.org/abs/2405.07434` [13]

Funding This work was supported by the Israel Science Foundation Grant No. 1102/21.

1 Introduction

Concurrent programs rely on concurrent data structures as a foundational component. Considerable effort has been dedicated to developing efficient concurrent data structures. 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 summarizes a range of elements with consecutive keys in the data structure into a succinct value. 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.

It is desirable 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 [2], which support the `select`(i) and `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 [14, 8] to efficiently support the basic operations of accessing $collection[i]$ and querying $collection.index(key)$ respectively.

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

[15, 11, 10, 1], 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: [7] proposed a concurrent array supporting a query that aggregates the values of all cells (and not of an input key range). They addressed only simple static arrays without supporting element insertion or deletion, whereas we address dynamic data sets. [12] proposed a way to efficiently support a `size` query returning the total number of elements in a concurrent set or dictionary. They utilize central metadata regarding the data structure's size to allow aggregate queries to return an answer without accessing the data structure's elements themselves. We, however, aim to efficiently answer queries about an arbitrary key range provided as input to the query. For this, one central metadata that aggregates information about all the keys is insufficient.

We focus our attention on aggregate queries on trees. We 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 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 (we call an operation that modifies the data structure, like a successful `insert` or `delete`, an *effectual operation*) 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 addressed aggregate functions and queries in Section 2.

The challenge in designing a linearizable [6] extension for such augmented trees stems from the fact that each effectual operation affects multiple locations (its target-leaf area and metadata fields in the nodes along its root-to-leaf path), and yet aggregate queries should obtain 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. We employ two mechanisms to achieve that: 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. These enable support for non-aggregate range queries as well. We demonstrate our design on the lock-based binary search tree of [3, 4], and present two algorithms that implement this design, highlighting the trade-offs between query time and update time. In the full version of the paper [13], we present all details of our design and algorithms, along with a correctness and complexity analysis.

Two independent concurrent works propose other solutions to the problem of concurrent aggregate queries. [9] suggests that all operations add themselves to a queue of operations in each tree node they traverse, and then help all preceding operations in the queue to advance to the appropriate child before proceeding to the next node in their own traversal. This way, a query sees a virtual snapshot of the tree, as if all previous operations have already been executed and none of the subsequent operations have. [9] do not support a failure option for the `insert` and `delete` operations, which may require traversing the tree twice, posing additional challenges. They also optimize only for aggregate query performance while our approach also includes an algorithm that optimizes the performance of the original tree operations. They offer an improved space complexity over the current work as they do

not employ multi-versioning, but multi-versioning does come with substantial benefits. It allows to reduce contention, as demonstrated by our `contains` operation that does not help concurrent operations throughout its traversal and by concurrent aggregate queries that do not help each other. Multi-versioning also enables to support aggregate queries that require several serial traversals (e.g., for querying the median key of a given input key range).

Another concurrent work [5] supports aggregate queries using an alternative multi-versioning mechanism. Every insert and delete operation builds from bottom up a tree of immutable version objects mirroring the tree itself, and links each version object to the corresponding tree node. Queries obtain a tree of versions from the root's version pointer, and operate on this tree. Running queries on an immutable copy of the tree allows for simpler concurrency. However, it comes with the cost of excessive allocations: each modification of a leaf by an insert operation or a delete operation requires creating new version objects for all ancestors of that leaf.

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 ensuring that the asymptotic time complexity of insertions and deletions is not substantially harmed. These operations should be able to maintain the metadata during their root-to-leaf traversal, as the only affected metadata should lie along their path to the target key.

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 $f : \mathcal{P}(A) \setminus \{\phi\} \to B$, 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), and the nodes' metadata type is denoted by B. Next we present a definition that will provide a useful property for aggregate functions:

▶ **Definition 1** (additive aggregate function). *An aggregate function $f : \mathcal{P}(A) \setminus \{\phi\} \to B$ is* additive *if there exists a binary operation $\oplus_f : B \times B \to B$ such that $(B, \oplus_f)$ is a commutative semigroup (namely, $\oplus_f$ is associative and commutative) and for every $X \in \mathcal{P}(A) \setminus \{\phi\}$, $f(X) = \oplus_{f\, a \in X} f(\{a\})$.*

We require 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. This ensures that the metadata in each node may be directly updated upon an insertion of a $(key, value)$ element to its subtree, to be $old \oplus_f f(\{(key, value)\})$ where old is the old metadata value. To be able to similarly update the nodes' metadata during a root-to-leaf traversal upon deletion, we require that the metadata function satisfy the following property:

▶ **Definition 2** (subtractive aggregate function). *We say that an aggregate function $f : \mathcal{P}(A) \setminus \{\phi\} \to B$ is* subtractive *if it is additive, and there exists a subtractive binary operation $\ominus_f : B \times B \to B$ such that for every disjoint $X_1, X_2 \in \mathcal{P}(A) \setminus \{\phi\}$, $f(X_2) = f(X_1 \cup X_2) \ominus_f f(X_1)$.*

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 deleted leaf upwards, is required by our algorithms. This is because not only the deletion initiator needs to calculate its effect on the metadata in the deleted leaf's ancestors. Concurrent operations linearized

after the deletion 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.

An aggregate query on an augmented tree returns a result based on multiple data elements of the data structure by executing merely root-to-leaf traversals. Metadata obtained during the traversals may be used both to navigate through the tree and to calculate 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 define a *simple* aggregate query, which executes only independent traversals. In contrast, a general aggregate query is a chain of one or more simple aggregate queries composed with one another: the user's input is the input to the first simple query in the chain, the output of the i-th simple query in the chain is the input to the $(i+1)$-st query, and the output of the last simple query is the output of the whole query.

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 perform a root-to-leaf traversal on the tree as follows. They maintain the prefix sum f using $\oplus_f$ 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). For each traversed node, this value and the aggregate value of the current left subtree are added using $\oplus_f$ to yield the prefix sum on all leaves with keys smaller than the current key. The computation of this value is made possible using one simple $\oplus_f$ operation thanks to using an additive aggregate function on the subtree's leaves as the node's metadata. Then a query-specific method `shouldDescendRight` is called to determine to which child the traversal should proceed. It takes as inputs the prefix sum up to the current key and the current node's key. In case of descending to the right, the prefix sum is updated to include the leaves of the current left subtree. The traversal stops when it reaches a leaf node, and returns this leaf and the prefix sum which is now the value of the metadata subtractive aggregate function on the set of $(key, value)$ pairs of all leaves with keys smaller than the reached key. A simple aggregate query is defined by as many `shouldDescendRight` methods as the traversals it needs and a method that takes the list of the traversals' outputs and computes the query's answer.

3 Our design for concurrent aggregate queries

We look at binary search trees implementing a dictionary, which is a collection of distinct keys with associated values, supporting an insertion of an input key with the associated input value if the key does not exist or else returns a failure; a deletion which deletes an input key and its value if the key exists and returns the value or else returns a failure; and a `contains` operation which returns the input key's value if it exists else returns NOT_FOUND. We look specifically at external trees, i.e., trees whose items are found in the leaves.

To make sure aggregate queries obtain a consistent view of the parts they traverse in the data structure, even though each concurrent effectual operation modifies multiple locations, each effectual operation and each aggregate query obtains a timestamp. We will ensure that every query observes all modifications related to effectual operations with timestamps $\leq$

its timestamp, and does not see modifications related to effectual operations with a greater timestamp. Our algorithms are linearizable, and the linearization order of effectual operations and aggregate queries respects the timestamp order, where aggregate queries are linearized after effectual operations with the same timestamp.

For a query to consider all modifications by concurrent effectual operations with timestamp $\leq$ its timestamp, ongoing effectual operations announce themselves by adding an announcement object with their details (including their timestamp) to a global announcements object. Queries read these announcements to fill in missing details about them by themselves, and form the desired full view of their traversed path.

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, we employ versioning in the spirit of the multi-versioning of [15] for modifiable fields in the tree's nodes. Effectual operations leave old versions of the data for concurrent 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 consist of a linked list of values tagged with descending timestamps. 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).

We apply our design to the lock-based binary search tree of [3, 4]. 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 in a root-to-leaf traversal. 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 grab a timestamp and gather the announced effectual operations, and then traverse the tree based on the aggregate metadata similarly to a sequential aggregate query, but while obtaining versions of child pointers and of aggregate metadata according to the obtained timestamp and announced effectual operations.

4 The two algorithms implementing our design

Different approaches could be taken toward the implementation of our design, specifically the operations announcement and the aggregate metadata representation, optimizing for the time complexity of either effectual operations or aggregate queries. We present two algorithms implementing our design: `FastUpdateTree` optimizes for tree update time, in fact incurring zero additional asymptotic time on the original tree operations when they do not face concurrent effectual operations on the same key. `FastQueryTree` offers a better worst-time complexity for aggregate queries, which is a function of their traversal length and the number of concurrent operations (and does not depend on the number of elements in the queried range as in the naive implementation).

In our design, 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. To reduce contention on effectual operations, `FastUpdateTree` lets them work mostly on single-writer fields written only by the thread that performs the operation. The object in which threads announce their effectual operations and the aggregate metadata field in each tree node are both arrays with a single-writer cell per thread. This demonstrates a time-space trade-off, as effectual operations can execute faster by paying in more space.

When effectual operations announce themselves in the announcements array in `FastUpdateTree`, they do not order themselves with 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 newer versions, not overriding data the query needs. For that, an effectual operation first announces itself with an unset timestamp and then obtains the global timestamp value and sets it in the announcement's timestamp field. It sets it using a `CAS` because a concurrent aggregate query might have already set it: aggregate queries obtain the global timestamp value and `CAS` it into the announcement's timestamp of each effectual operation with an unset timestamp they encounter in their traversal of the announcements array.

As for the aggregate metadata array in each `FastUpdateTree` node, each of its cells is a versioned field (namely, holds a linked list of versions with timestamps) containing metadata regarding operations on the node's subtree by the thread associated with this cell. Aggregate queries can calculate the total aggregate value from the per-thread values using $\oplus_f$ (the aggregate function's binary operation).

`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, an effectual operation 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 announcement object to a global queue, with a timestamp greater by 1 than the timestamp of the preceding announcement in the queue. The timestamps induce a total order on all effectual operations. Aggregate queries obtain a timestamp (that determines which effectual operations they take into account) from the end of the announcements queue. Equipped with this timestamp, they know which version of each aggregate metadata field to obtain and which announced effectual operations they should consider.

References

1 Maya Arbel-Raviv and Trevor Brown. Harnessing epoch-based reclamation for efficient range queries. In *PPoPP*, 2018. doi:10.1145/3178487.3178489.

2 Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

3 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.

4 Tudor Alexandru David, Rachid Guerraoui, Tong Che, and Vasileios Trigonakis. Designing ASCY-compliant concurrent search data structures. Technical report, EPFL, 2014.

5 Panagiota Fatourou and Eric Ruppert. Lock-free augmented trees. In *DISC*, 2024.

6 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *TOPLAS*, 12(3), 1990. doi:10.1145/78969.78972.

7 Prasad Jayanti. f-arrays: Implementation and applications. In *PODC*, 2002. doi:10.1145/571825.571875.

8 Grant Jenks. Python sorted containers, 2019. URL: https://grantjenks.com/docs/sortedcontainers.

9 Ilya Kokorin, Dan Alistarh, and Vitaly Aksenov. Wait-free trees with asymptotically-efficient range queries. In *IPDPS*, 2024. doi:10.1109/IPDPS57955.2024.00023.

10 Jacob Nelson-Slivon, Ahmed Hassan, and Roberto Palmieri. Bundling linked data structures for linearizable range queries. In *PPoPP*, 2022. doi:10.1145/3503221.3508412.

11 Erez Petrank and Shahar Timnat. Lock-free data-structure iterators. In *DISC*, 2013. doi:10.1007/978-3-642-41527-2_16.

12 Gal Sela and Erez Petrank. Concurrent size. *PACMPL*, 6(OOPSLA2), 2022. doi:10.1145/3563300.

13 Gal Sela and Erez Petrank. Concurrent aggregate queries. *arXiv preprint*, 2024. doi:10.48550/arXiv.2405.07434.

14 Daniel Stutzbach. blist: an asymptotically faster list-like type for Python, 2010. URL: http://stutzbachenterprises.com/blist.

15 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.