Wait-free Trees with Asymptotically-Efficient Range Queries

Ilya Kokorin
VK.com, ITMO University
Saint-Petersburg, Russia
kokorin.ilya.1998@gmail.com

Victor Yudov
ITMO University
Saint-Petersburg, Russia
me@spcfox.ru

Vitaly Aksenov
City, University of London
London, UK
aksenov.vitaly@gmail.com

Dan Alistarh
IST Austria
Vienna, Austria
dan.alistarh@ist.ac.at

***Abstract*—Tree data structures, such as red-black trees, quad trees, treaps, or tries, are fundamental tools in computer science. A classical problem in concurrency is to obtain expressive, efficient, and scalable versions of practical tree data structures. We are interested in concurrent trees supporting *range queries*, i.e., queries that involve multiple consecutive data items. Existing implementations with this capability can list keys in a specific range, but do not support *aggregate range queries*: for instance, if we want to calculate the number of keys in a range, the only choice is to retrieve a whole list and return its size. This is suboptimal: in the sequential setting, one can augment a balanced search tree with counters and, consequently, perform these aggregate requests in logarithmic rather than linear time.**

In this paper, we propose a generic approach to implement a broad class of range queries on concurrent trees in a way that is wait-free, asymptotically efficient, and practically scalable. The key idea is a new mechanism for maintaining metadata concurrently at tree nodes, which can be seen as a wait-free variant of hand-over-hand locking (which we call *hand-over-hand helping*). We did a preliminary implementation of the wait-free binary search tree and preliminary experiments have indicated the soundness of our approach.

***Index Terms*—Data structures, Concurrent programming, Range queries**

I. Introduction

Tree data structures are ubiquitous in computer science, due to their high expressive power and practical versatility. For instance, in databases, index trees allow searching for an indexed key faster than traversing through all the elements. Typically, such index is implemented as B-tree [10], [16], [20], although alternate implementations are possible, such as the red-black tree [21], or the splay tree [32]. Moreover, one could use quad trees [17] to store and retrieve a collection of points in a plane, or tries [11] for fast prefix matching in strings.

In this paper, we are interested in *concurrent* implementations of fundamental tree data structures that combine theoretical and practical efficiency, with *expressivity* in terms of the class of queries they support efficiently. Specifically, we are interested in trees supporting the following types of operations. We call a query, retrieving or modifying a single data item, a *scalar query*; and a query, involving multiple consecutive (by value) data items, a *range query*. For example, a search tree can provide the following scalar queries:

The paper is partially sponsored by JetBrains and Huawei.

- `insert(key)` — if `key` does not exist in the tree, inserts it to the tree, otherwise, leaves the tree unmodified;
- `remove(key)` — if `key` exists in the tree, removes it from the tree, otherwise, leaves the tree unmodified;
- `contains(key)` — returns `true` if the tree contains `key`, `false`, otherwise.

Also, a search tree can provide the following range queries:

- `collect(min, max)` — returns all the keys from the `[min; max]` interval from the set;
- `count(min, max)` — returns the number of keys from the `[min; max]` interval from the set.

In addition, we would like to support aggregate range queries: for example, in a search tree storing key-value pairs, the range query `range_add(min, max, delta)` adds `delta` to all the values corresponding to the keys in a given range `[min, max]`, whereas the range query `range_sum(min, max)` calculates the sum of all values corresponding to the keys in a given range `[min, max]`.

In this work, in addition to extensively investigated `collect` query (see e.g., [8], [13]) we require the index to perform aggregate range queries (e.g., `count`) in an asymptotically optimal way (i.e., their asymptotic cost should not exceed the logarithmic cost in a sequential setting). For example, we can use such aggregate range queries to find the number of requests to the system in the specified time range from the specified users.

Currently, all existing concurrent trees answer the aggregate range queries in time proportional to the number of elements in the range, i.e., for a count query it works as `count(min, max) = collect(min, max).length()`. This is clearly suboptimal: in the sequential setting, augmented search trees can perform such queries in $O($`height`$)$ (where `height` is the height of the tree) which can be exponentially faster for balanced trees.

Now, we overview how to sequentially perform `count` query in $O($`height`$)$ time for a binary search tree. Note that other aggregate range queries can be implemented similarly. For each node, we store the number of keys in its subtree. Then, we start traversing the tree from the root downwards. When we are in the node `v` we check three cases. If the range of keys in the subtree of `v` lies inside the required range — we add the stored size of the subtree of `v` to the answer. If it intersects with the required range — we go recursively to children, returning the sum of results for `v.left` and `v.right`. Finally, if it does not intersect with the required range — we stop the call and return zero (we unroll this recursion in our sequential and concurrent implementations). We provide a more detailed description of the sequential algorithm, as well as the proof of the $O(height)$ time complexity, in Technical Report [**?**].

In this paper, we present a scalable approach that can make any tree data structure support wait-free operations, including asymptotically efficient aggregate range queries with logarithmic amortized time. The main idea is that the execution of an operation `Op` by a process `P` at node `v` begins by inserting the descriptor of `Op` into the root queue `root.Queue`, and obtaining a timestamp. Then, the process `P` helps to perform all pending operations in the queue, applies itself, and proceeds recursively at the children nodes, applying the same pattern. Thus, the process traverses the tree downwards, from the root to the appropriate lower nodes, at which the operation (e.g., an insertion of a new data item, or a removal of an existing one) should be performed. This method can be seen as a *wait-free* version of the classic *hand-over-hand locking* technique [25], where instead of blocking we ask processors to perform work that is preceding them in the queue. We name this method as *hand-over-hand helping*. In the following, we describe this construction in detail, using a binary search tree as a running example. Finally, we provide an implementation of such a tree, supporting `insert`, `delete`, `contains`, and `count` operations. We discuss its overheads and preliminary experiments which indicate the soundness of our approach. However, despite the presence of such experiments, we consider our main contribution a theoretical approach to implementing asymptotically efficient wait-free range queries on concurrent trees rather than experimental evaluation.

Roadmap. In Section II we overview our approach. In Section III, we present the preliminary results that show the soundness of our approach. And we conclude in Section IV.

A. Related work

Lock-based solutions. The easiest and the most obvious way to implement a concurrent data structure is to protect a sequential data structure with a *lock* to guarantee mutual exclusion [27]. Such construction is not lock-free (it is not even obstruction-free) and suffers from starvation. Moreover, since a lock allows only one process to work with the data structure at a time, the resulting construction does not scale and its throughput remains low.

There exist more complicated lock-based approaches, e.g., an approach with hand-over-hand validation and locks used during the update phase [12]). One can see them as an inspiration to our project. However, these approaches lack strong progress guarantees, such as lock-freedom.

Linear-time solutions. Several papers [8], [9], [13], [19], [35] address the issue of executing lock-free (and even wait-free, but with lock-free scalar queries) range queries on concurrent trees. However, the aforementioned papers address only the `collect(min, max)` query, returning the list of keys, located within a range `[min; max]`. All other range queries are proposed to be implemented on top of the `collect` query. For example, as we said before, the `count` query can be implemented as `count(min, max) = collect(min, max).length()`.

This approach suffers from a major drawback: the `collect` query is executed in time proportional to the number of keys in the range. Thus, for wide ranges, such query takes $O(N)$ time where N is the size of the tree: the number of keys in the range is almost equal to the size of the tree. This implementation is not asymptotically efficient: e.g., the `count` query can be executed in $O(\log N)$ time in a sequential environment using balanced search trees.

Therefore, despite being lock-free, these methods do not guarantee time efficiency, and thus cannot be used.

Persistent data structures. There exists a solution for efficient aggregate range queries based on persistent data structures [3]. Each read-only operation (e.g., `contains` or `count`) takes the current version of the data structure and operates on it. Each update operation (e.g., `insert` or `remove`) creates a new version of the data structure without modifying the existing one and then tries to replace the old version with the new one using a Compare-And-Swap [1] (or CAS, for simplicity). If the CAS succeeds the operation finishes, otherwise the operation restarts from the very beginning. This approach is called Lock-free Universal Construction [25] and can be applied to any sequential persistent tree. As an interesting observation, this approach scales even on write-only workloads [5]. However, there are at least three drawbacks: 1) we cannot provide wait-free guarantees — an operation can restart infinitely often if it is not lucky enough; 2) for an update range query, the majority of work is spent needlessly — an unsuccessful CAS makes us retry the whole operation from the start; 3) it stops scaling early, e.g., on 10 threads.

Parallel augmented persistent trees. Sun, Ferizovic, and Belloch [34] presented a persistent augmented tree that can serve a batch of operations in parallel using *fork-join* parallelism. The paper does not propose a method of executing concurrent operations on augmented data structures. However, we can use various combining techniques [7], [23], [33] to form large batches of operations from individual concurrent updates. The main problem with this approach is that the combining techniques increase individual operation latency and, thus, are not acceptable in settings, where low operation latency is required.

II. Overview of the approach

A. Timestamps invariant

The main problem with the sequential algorithm for an aggregate range query presented in the introduction is that it will be incorrect if running as is in a concurrent environment. Indeed, each update operation (e.g. `insert` or `remove`) should modify not only the tree structure, but the *augmentation values* on the path. Augmentation values are values required for aggregate range queries to calculate the results in the

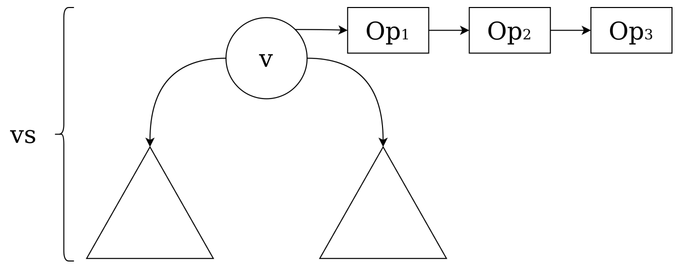

Fig. 1: Node `v` has an operations queue with descriptors of three operations: $\texttt{Op}_1$, $\texttt{Op}_2$ and $\texttt{Op}_3$. These three operations should be applied to `vs` in the order of descriptors in the queue: first $\texttt{Op}_1$, then $\texttt{Op}_2$, and, finally, $\texttt{Op}_3$

required complexity: for example, for `count` queries in each node we have to store the size of that node subtree as an augmentation value. By that, the augmentation values may become inconsistent with the tree structure.

Therefore, the main purpose of our concurrent solution is to get rid of such situations by ensuring that all operations are executed in a particular order. We enforce a particular execution order by maintaining an operation queue in each node.

Consider an arbitrary node `v` and its subtree `vs`. At `v` we maintain an *operations queue*, that contains descriptors of operations to be applied to `vs` (Fig. 1). These operations can, for example, insert a key to `vs` or remove a key from `vs`. We maintain the following invariant: operations should be applied to `vs` in the order, their descriptors were added to `v` queue.

Note that the aforementioned invariant can be applied to the root node too: indeed, since the whole tree is just the subtree of the root operations should be applied to the tree in the order their descriptors were added to the operations queue in the root. Thus, the order, in which operation descriptors are added to the queue in the root, is exactly the *linearization order*.

Thus, we may use the operations queue at the root to allocate timestamps for operations. A timestamp allocation mechanism should provide the following guarantee: if a descriptor of operation `A` was added to the root queue before a descriptor of operation `B`, then `timestamp(A)` $<$ `timestamp(B)` should hold. We explain how to achieve it in Section II-D. We store the timestamp of an operation in the corresponding descriptor, i.e., `descriptor.Timestamp` field.

As was stated before, operations should be applied to the tree in the order, their descriptors were added to the root descriptor queue. Therefore, one can wonder: how can we achieve parallelism, while linearizing all operations via the root queue? Note that there is no parallelism only in the queue in the root. Lower by the tree, two operations (even the modifying ones, e.g., two `inserts`) may be executed in parallel if they are executed on different subtrees, since on lower tree levels their descriptors will be placed to different operation queues (Fig. 2).

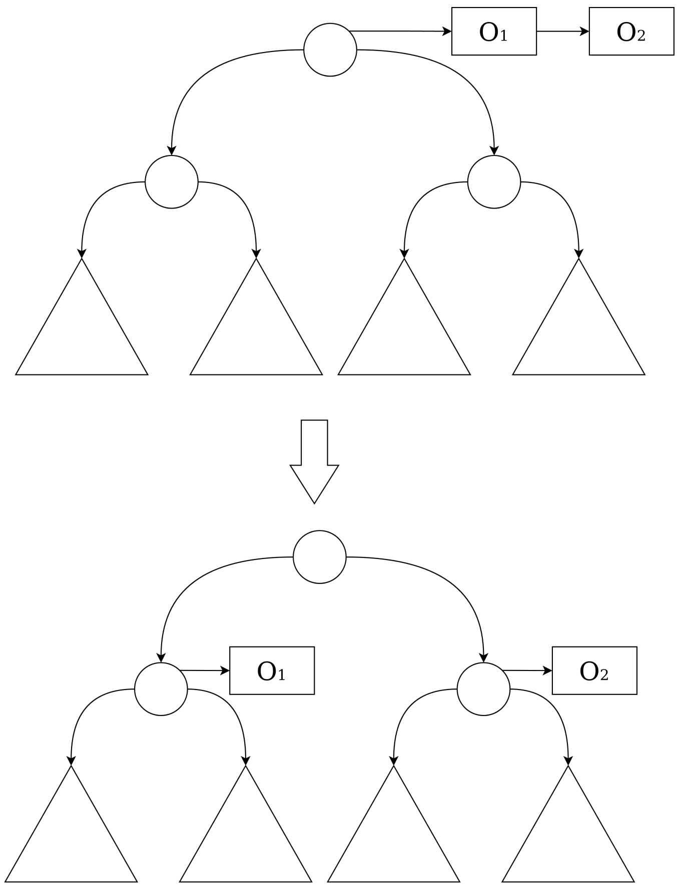

Fig. 2: Two operations can be executed in arbitrary order (even in parallel) if they operate on different subtrees, since on lower tree levels they are placed to different queues

B. Operation execution: overview

For simplicity of the overview, we consider only unbalanced trees for now. If we want to make our tree balanced, we can adapt the subtree rebuilding approach (we provide a detailed description in Section II-E). The study of other concurrent balancing strategies we leave for the future work.

The execution of an operation `Op` by a process `P` (we call such process `P` the *initiator* process) begins with inserting the descriptor of `Op` into the root queue and obtaining `Op` timestamp. In Section II-D, we describe, how the root queue with timestamp allocation may be implemented.

After that, the initiator process starts traversing the tree downwards, from the root to the appropriate lower nodes, at which the operation (e.g., an insertion of a new data item, or a removal of an existing one) should be performed.

In each visited node `v` some additional actions should be performed in order to execute `Op` properly. For example, during the `count` query the size of `v`'s subtree can be added to the result, and during `insert` or `remove` operations pointers to `v`'s children and `v`'s subtree size can be changed. We call the process of performing these necessary actions — an *execution of operation* `Op` *in node* `v`.

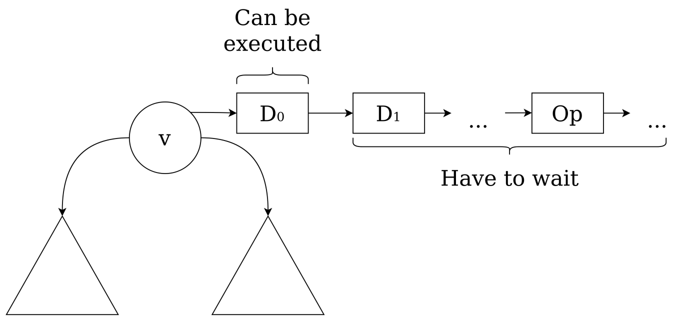

Fig. 3: Process P has to wait before executing Op in node v, since only the operation D_0, corresponding to the descriptor at the head of v queue, can be executed right now in v.

As stated in the previous subsection, operations should be applied to v's subtree in the order their descriptors appear in v's operations queue. Thus, if the descriptor of Op is not located at the head of v's queue the initiator process P has to wait before executing Op in node v (Fig. 3). The execution of Op in node v cannot begin until execution of all the preceding operations in node v is finished.

To make the algorithm wait-free we use the *helping* mechanism (e.g., [22], [29]): instead of merely waiting for the Op descriptor to move to the head of v queue, P helps executing in node v the operation from the head of the queue — D_0 in the example above. Thus, even if the initiator process of D_0 is suspended, the system still makes progress.

As discussed later, while helping to execute operations $D_0, D_1, \ldots$ in node v the process P removes descriptors of these operations from the head of v's queue and inserts them to queues of appropriate v's children. Thus, while helping other processes execute their initiated operations in v, P moves Op descriptor closer to the head of v queue. Once P helped all preceding operations to finish their execution in node v, it can finally execute its operation Op in v (note that some other process may help executing Op in v, just like P previously helped executing D_0 in v).

The process of executing an operation Op in a node v consists of the following actions:

1. Determine the set of child nodes C, in which Op execution should continue.
   For example, an execution of the count query on a binary search tree may continue in either single child or both children, as explained in Section I.
2. For each child c from the set C:
   2.1. Modify the state of c (e.g., a size of c's subtree), if necessary;
   2.2. Try to insert Op descriptor to the end of c's operations queue, thus allowing Op to continue its execution at lower levels of the tree.
3. Remove Op descriptor from the head of v's queue.

Note, that during the execution of operation Op in node v the said operation only modifies states of v's children, not v itself. Thus, no operation can ever modify the root state, since the root is not a child of some other node. We overcome that issue by the introduction of the *fictive root*. This fictive root does not contain any state and has only one child — the real tree root. The only purpose of the fictive root is to allow operations to modify the state of the real root. The state of the real root can be modified by operation Op while Op is being executed in the fictive root, since the real root is the child of the fictive root.

In Section II-C, we describe how an operation Op should be executed in a node v.

Since now we force processes to help each other, operation Op, initiated by process P, in any node v can be executed by some other helper process. Thus, we need to provide a mechanism for the process P by which it distinguishes between the two following situations:

- Operation Op has not yet been executed in node v. Thus, the descriptor of Op is still located somewhere in v queue. In that case, P needs to continue executing operations from the head of v queue in node v.
- Operation Op has already been executed in node v. In that case, P can proceed to execute Op in lower nodes of v's subtree.

We use timestamps to distinguish between these two situations. We describe that usage of timestamps with formulating and proving *timestamps increasing property*.

Theorem 1. *In each queue, operation timestamps form a strictly increasing sequence. More formally, if at any moment we traverse any queue Q from the head to the tail and obtain $t_1, t_2, \ldots t_n$ — a sequence of timestamps of descriptors, located in Q, then $t_1 < t_2 < \ldots < t_n$ will hold.*

We prove that theorem in Technical Report [**?**].

As follows from that property, the initiator process P can easily learn, whether its operation Op has been executed in node v by using the simple algorithm:

- if the queue is empty — we conclude that Op has been executed in v;
- if the queue is not empty, we compare the timestamp of the descriptor in the head of v queue with the timestamp of Op: if v.Queue.Head.Timestamp > Op.Timestamp, we conclude that Op has been executed in v, otherwise, we conclude that Op has not been executed in v yet.

Therefore, we can implement the algorithm of executing all operations from v's queue up to Op.Timestamp (Listing 1):

```
fun execute_until_timestamp(Op, v):
    while true:
        // obtains the first descriptor in FIFO order
        head_descriptor := v.Queue.peek()
        if head_descriptor = nil:
            return
        if head_descriptor.Timestamp > Op.Timestamp:
            return
        // execute_in_node changes states of v children
        // pushes head_descriptor to child queues,
        // removes head_descriptor from v queue
        execute_in_node(head_descriptor, v)
```

Listing 1: The algorithm to execute all operations, up to the specified timestamp `Op.Timestamp`, from `v` queue

Suppose the initiator process `P` is traversing the tree to execute operation `Op` and `P` just finished executing `Op` in node `v`. How can `P` choose the next node in the traversal? It is not necessary to always continue the traversal in one of `v`'s children, since `Op` can be now finished in `v`'s subtree by other helper processes. To address this issue, in each operation descriptor we store a queue with nodes `Op.Traverse` — the queue of nodes that must be visited during the execution of `Op`. The `Traverse` queue is maintained and used in the following way:

- When any process (no difference initiator or helper) starts executing `Op` in node `v`, it adds to the tail of `Op.Traverse` all children of node `v` in which the execution should continue;
- When the initiator process finishes the procedure `execute_until_timestamp(Op.Timestamp, v)`, it removes `v` from the head of `Op.Traverse` queue. Note, that only the initiator process can remove nodes from `Op.Traverse` queue;
- After the initiator process has removed the current node `v` from the head of `Op.Traverse`, it checks `Op.Traverse`: if it is empty, the operation is completed and the initiator returns the query result to the caller; otherwise (if `Op.Traverse` is not empty), the initiator continues the traverse by taking the next node from the head of `Op.Traverse`.

Note, that this queue maintenance scheme allows a node `v` to be inserted into `Op.Traverse` multiple times, since multiple helper processes may be executing `Op` in `v`'s parent in parallel. However, as will be explained in Section II-C `v`'s state will still be modified exactly once, no matter how many times it is processed. The traverse algorithm can be implemented as in Listing 2.

```
fun execute_operation(op):
    Tree.Root.Queue.push_acquire_timestamp(op)
    op.Traverse = {Tree.Root}
    while true:
        v := op.Traverse.peek()
        if v = nil: // op is finished
            return
        execute_until_timestamp(op.Timestamp, v)
        op.Traverse.pop()
```

Listing 2: The algorithm for traversing the tree by the initiator process

Now we have to design a method, that will allow the initiator process to learn the operation result when the operation is completed. The problem here is that the operation result might consist of multiple parts (e.g., `count` result consists of a sum of multiple subtree sizes), and these parts (e.g., subtree sizes) may be computed by different processes, since force processes to help each other.

To allow operation result to be assembled from these parts, in each operation descriptor we store a concurrent map `Op.Processed`, filled with nodes, in which the execution of `Op` has been finished. The size of this map is expected to be small for aggregate range queries (e.g., $O(\log N)$), so, we can implement them in any way we want: a wait-free queue that stores all the required nodes (maybe multiple times, which we filter out at the end of the operation) or with a Wait-free Universal Construction [24], and, finally, we can use a wait-free map.

The `Op.Processed` uses tree nodes as its keys. To allow this, we augment each tree node `v` with an identifier, stored in the `v.Id` field. Each node receives its identifier at the creation moment and the node identifier does not change throughout the node lifetime. The node identifiers must be unique. We can achieve that property using Universally Unique Identifier (UUID) [4] generation procedure or by incrementing `fetch-and-add` [2] counter.

Values of the `Op.Processed` map store parts of the result: for example, for the `count` query we store in the `Op.Processed` the node identifiers with the sizes of their subtrees that should be added to the result of the query.

Before removing `Op` descriptor from the head of `v`'s queue we try to add `v.Id` along with a value `x`, corresponding to the part of the answer for the node `v`, into the `Op.Processed` map. If key `v.Id` already exists in the `Processed` map, we left the `Op.Processed` map unmodified, without changing the value, associated with `v.Id`.

We never modify the value, associated with node `v`, since stalled processes can calculate the value incorrectly. Indeed, consider the following scenario:

1. Descriptor `D`, corresponding to a `count` operation with timestamp `42`, is located at the head of `v`'s queue;
2. Process `P` reads `D` from the head of `v`'s queue;
3. Process `P` is suspended by the OS;
4. Process `R` reads `D` from the head of `v`'s queue;
5. Process `R` determines that the size of `v`'s left subtree should be added to the result;
6. Process `R` reads the size of `v`'s left subtree (say, it equals to 5) and adds key-value pair $\langle$ `v.Id, 5` $\rangle$ to the `Processed` map;
7. A new key is inserted to `v` left subtree by `insert` operation with timestamp `43`, making `v` left subtree size equal to 6;
8. Process `P` is resumed by the OS;
9. Process `P` reads the size of `v`'s left subtree (now it equals to 6) and tries to add key-value pair $\langle$ `v.Id, 6` $\rangle$ to the `Processed` map.

On step (9) we should not modify the value, corresponding to the node `v`, since the value `6` reflects the modification, performed by the `insert` operation with timestamp `43`. The `count` operation has timestamp `42`, thus, the `count` result should not include the key, inserted by `insert` operation with timestamp `43`.

When the operation execution is finished (i.e., `Op.Traverse` is empty) we traverse the `Processed` map, forming the query result from partial results associated with visited nodes. Note, that it is safe to traverse the

Processed map — indeed, now the Processed map cannot be modified concurrently, since the query execution is finished.

C. Detailed description of an execution in a node

In Section II-B, we explained how the execution of the operation works in general. Now, we go into details of the execution in the node.

The process of executing an operation Op in a node v consists of the following actions:

- Determine the set of child nodes C, in which Op execution should continue.
- For each child c from the set C:
  1. Insert c into Op.Traverse queue;
  2. Modify the state of c (e.g., a size of c's subtree), if necessary;
  3. Insert Op descriptor to the operations queue of c, thus allowing Op to continue its execution at lower levels of the tree.
- Try to add v.Id along with a value x, corresponding to the part of the answer for the node v, into Op.Processed map.
- Try to remove Op descriptor from the head of v's queue if it is still there.

The removal of Op descriptor from the head of v's queue should be done after the insertion of Op descriptor to child queues and modification of child states are finished. Otherwise, the execution of later operations in v may start before the execution of Op in v is finished, which may break the main invariant (Section II-A). Inserting the descriptor to child queues, modifying child states, and removing the descriptor from the parent queue should happen exactly once, no matter how many processes are working on the descriptor concurrently.

Exactly-once insertion to and removal from queues is handled by our implementation of concurrent queues (see Section II-D). Queues provide two procedures:

- push_if inserts the descriptor to the tail of the queue only if it has not been inserted yet, otherwise, the queue is left unmodified.
- pop_if removes the descriptor from the head of the queue only if it has not been removed yet, otherwise, the queue is left unmodified.

The main problem in the execution of an operation Op in a node v is the proper work with the children states: we should be able to work with each state atomically and we should modify each state exactly once, no matter how many processes are executing Op in v.

The atomicity problem comes from the fact that the state may consist of multiple fields. To solve this problem, we do not store the state directly inside the node — instead we store the immutable state in the heap and the node stores the pointer S_Ptr to it.

The state, located in the heap, is considered immutable and is never modified. To modify the node state, we simply do the following:

1. create the structure, corresponding to the modified state, with an arbitrary set of fields changed;
2. place the modified state somewhere in the heap;
3. change the node.S_Ptr, so that it points to the new state.

To read the state atomically, we simply read the S_Ptr register. After that, we can safely access any field from the state structure, pointed at by the fetched pointer, without worrying that the state structure is being modified concurrently by another process. Since the structure is immutable, it can never be modified by another process.

Now, we return to the second problem of modifying the state exactly once. In the state we store one additional field: Ts_Mod — timestamp of the operation, that was the last to modify the state. Thus, if the operation Op is willing to modify the state of node v, we should first read the current v's state and acquire the last modification timestamp.

- If Ts_Mod $\geq$ Op.Timestamp we conclude that v's state has been already modified by Op. In that case, we simply do not try to modify v's state according to Op anymore.
- Otherwise, we create a new state (with Ts_Mod = Op.Timestamp) and try to change the state pointer using CAS(&v.S_Ptr, cur_state, new_state). We then go to the next step, no matter what was the CAS result. If the CAS returned true — we have successfully modified the state, otherwise (if the CAS returned false), some other process has already modified the state according to Op.

Thus, the state is modified with each executed operation exactly once. Indeed, even if some stalled process will try to modify node v with an already applied operation Op the node state will not be changed, since the last modification timestamp is greater than or equal to Op.Timestamp. Therefore, the algorithm can be implemented in the following way (Listing 3):

```
fun execute_in_node(op, v):
    C := /* set of v children in which
        execution of op should continue */
    for c in C:
        cur_state := v.State_Ptr
        op.Traverse.push(c)
        if cur_state.Ts_Mod < op.Timestamp:
            new_state := op.get_modified_state(cur_state)
            new_state.Ts_Mod := op.Timestamp
            CAS(&v.State_Ptr, cur_state, new_state)
        c.Queue.push_if(op)
    node_key := v.Id
    node_value := /* part of the result
        corresponding to v */
    op.try_insert(node_key, node_value)
    v.Queue.pop_if(op)
```

Listing 3: Algorithm for executing operation op in node v

D. Implementation of an operations queue

Queue structure. For our purpose, we can use any practical queue algorithms as a basis for our descriptors queues, e.g., fetch-and-add queue [36] or practical wait-free queue [26]: the final implementation remains almost the same.

However, for simplicity of the presentation, we use Michael-Scott queue [29]. This queue is lock-free which makes the whole algorithm lock-free. But if we make the root queue to be wait-free — all other queues based on Michael-Scott queue will automatically have the same progress guarantee due to the way how we work with the descriptors. For more information about the wait-freedom see Section II-F.

In each node of the queue we store the descriptor in field `Data` and the pointer to the next node in field `Next`. Also, we have two pointers: `Tail`, that points to the last node of the queue, and `Head`, that points to the node *before* the first node of the queue. Note that the node at `Head` pointer does not store any data, residing in the queue. This node is considered dummy and only the node at `Head.Next` pointer contains the first real descriptor in the queue.

Queue in the root.

As discussed in Section II-A, the operation queue in the root node should provide timestamp allocation mechanism, with the following guarantees: if the descriptor of operation `A` was added to the root queue before the descriptor of the operation `B`, then `timestamp(A) < timestamp(B)` should hold.

As explained in "Queue structure" paragraph, we can use a slight modification of Michael-Scott queue [29] to implement the timestamp allocation mechanism for the root queue. Each time we need to add a new descriptor to the root queue, we 1) create a new node with the descriptor; 2) take the timestamp of the tail; 3) set the new timestamp in our descriptor as the incremented timestamp of the tail; 4) try to move the queue tail to the new node using CAS; 5) if the CAS is successful we stop, otherwise, we repeat from step (2).

This queue algorithm is not wait-free, however in Section II-F, we show how to implement such queue in a wait-free manner.

`push_if` implementation. As discussed in Section II-C, non-root queues should provide `push_if` operation that inserts a descriptor into the queue if it was not inserted yet (otherwise, the queue should be left unmodified). The procedure is based on the Michael-Scott queue insertion algorithm [29]: we check the timestamp of the tail, if it is higher then the descriptor has been inserted and we leave the queue unmodified, otherwise, we try to move the queue tail to the new node using CAS.

`pop_if` implementation. As discussed in Section II-C, the operation queue in any node should provide `pop_if` operation, that tries to remove descriptor with the specified timestamp `TS` from the head of the queue. If descriptor `D` with timestamp `TS` is still located at the head of the queue, it is removed. Otherwise, the queue is left unmodified — in this case, we assume that `D` was removed by some other process. We assume that at some moment `D` was located at the head of the queue (it may still be located at the head of the queue or it may be already removed), i.e., we never try to remove a descriptor from the middle of the queue. We can do this using Michael-Scott queue [29].

E. Balancing strategy

Until now, we considered unbalanced trees which may have $height \in \Omega(\log N)$. Since most of the queries (e.g., `insert`, `remove`, `contains` ,and `count`) are executed on a tree in $\Theta(height)$ time, using unbalanced trees may result in these queries being executed in non-optimal $\omega(\log N)$ time. Therefore, we must design an algorithm to keep the tree balanced. One possible balancing strategy is based on a subtree rebuilding and is similar to the balancing strategy proposed in [6], [15], [28], [31]. The idea of this approach can be formulated the following way: when the number of modifications in a particular subtree exceeds a threshold, we rebuild that subtree making it perfectly balanced.

For each tree node we maintain `Mod_Cnt` in the node state — the number of modifications in the subtree of this node. Moreover, for each node we store an immutable number `Init_Sz` — the initial size of its subtree, i.e., the number of data items in that node subtree at the moment of node creation (node can be created when a new data item is inserted to the tree or when the subtree, where the node is located, is rebuilt). We rebuild the node subtree when `Mod_Cnt > K ·  Init_Sz`, where `K` is a predefined constant. This approach makes the rebuilding to take $O(1)$ amortized time and, thus, the rebuilding does not affect amortized total cost (according e.g., to [28]).

We check whether the subtree of `v` needs rebuilding (and perform the rebuilding itself) before inserting an operation descriptor to `v`'s queue and changing `v`'s state. Therefore, we can perform `v`'s subtree rebuilding only during execution of some operation in `v`'s parent.

Consider node `v`, its parent `pv` and operation `Op`, that is being executed in `pv` and that should continue its execution in `v`'s subtree (and, therefore, its descriptor should be inserted to `v`'s queue). Before inserting `Op` to `v`'s queue and changing `v` state, we check whether `Mod_Cnt` in `v` will exceed the threshold after applying `Op` to `v`'s subtree: if so, `v` subtree must be rebuilt.

Note, that the subtree of `v` can contain unfinished operations: their descriptors still reside in the queues in that subtree (Fig. 4).

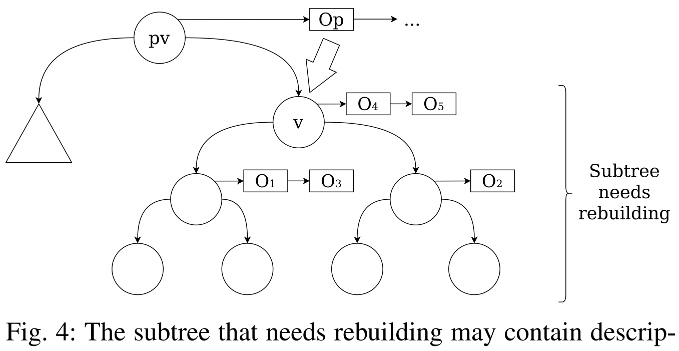

Fig. 4: The subtree that needs rebuilding may contain descriptors of unfinished operations

As the first step, we should finish all these unfinished operations before rebuilding the subtree. To do so, we traverse the subtree and in each node $\mathtt{u} \in \mathtt{subtree(v)}$ execute all operations, residing in `u` queue. After that, we again traverse the subtree of `v`, that no longer contains unfinished operations, and collect all the stored data items (e.g., keys or key-value pairs). Then, we build an ideally balanced subtree, containing all these data items.

Each node of the new subtree should be initialized with `Mod_Cnt = 0` and contain correct `Init_Sz`. We should set `Ts_Mod` of each node in the rebuilt subtree so that `Op` and all later operations (with $\mathtt{timestamp} \geq \mathtt{Op.Timestamp}$) can still modify the new subtree, but all the preceding operations (with `timestamp < Op.Timestamp`) cannot. Thus, we set `Ts_Mod = Op.Timestamp - 1`.

After that, we take `nv` — the root of the new subtree and try to modify the pointer that pointed at `v`, so that it starts to point at `nv`. For example, if `v` was the left child of `pv`, we execute `CAS(&pv.Left, v, nv)`; if `v` was the right child of `pv`, we execute `CAS(&pv.Right, v, nv)`. If the `CAS` returned `true` we conclude that we have successfully finished the rebuilding; if the `CAS` returned `false` we conclude that some other process has completed the rebuilding before us. In either case we resume the execution of `Op` in `pv`: we read `nv` — new root of the subtree, modify `nv`'s state, insert `Op` descriptor to `nv`'s queue (here we re-read root of the subtree because `nv` can be root of the subtree build not by our process, but by some another helder process) and remove `Op` descriptor from `pv` queue.

F. Wait-freedom

We now prove that our solution can be implemented efficiently with wait-free progress guarantee. We recall that *wait-freedom* [24] is a progress guarantee that requires all non-suspended processes to finish their execution within a bounded number of steps.

Theorem 2. *Each operation `Op` in our solution finishes within a bounded number of steps.*

To prove that theorem we recall that the execution of operation `Op` consists of:

1) Inserting `Op` descriptor into the root queue;
2) Propagating `Op` descriptor downwards, from the root to the appropriate lower nodes;
3) Executing `Op` in each node `v` on the target path.

Now, we prove that each of these stages finishes within a bounded number of steps.

Lemma 1. *The insertion of a descriptor into the root queue finishes within a bounded number of steps.*

Proof. Our queue implementation, described in Section II-D is lock-free, but not wait-free, since it is just a version of Michael and Scott queue [29].

The simplest approach is to implement the wait-free root queue using the well-known Wait-free Universal Construction [24], with no implementation caveats.

However, this approach has a very huge overhead. We hope that some practical wait-free queue (e.g., [26], [36]) can emulate our root queue and its timestamps distribution. Unfortunately, a wait-free queue from [36] can support the increasing timestamps using cell identifiers for that, but do not allow a simple wait-free peek function, that reads the head of the queue but does not remove it — this functionality is crucial for our queue in `pop_if`. Luckily for us the wait-free queue from [26] supports wait-free peek function and supports non-decreasing timestamps (or `epochs` in the paper). We can make them strongly increasing using a `fetch-and-add` register.

To distribute the timestamps, we need a version variable and an array of size P that contains the current descriptors. Each descriptor has an empty timestamp variable at the initialization. When performing an operation, process π creates a new descriptor and puts it into the corresponding cell. Then, it gets new version from the version variable using `fetch-and-add` and tries to CAS the current empty timestamp in its descriptor to the obtained version. Not depending on the result of CAS, the descriptor of π has a timestamp. Then, π traverses the array of descriptors and replaces empty timestamps by a newly fetched version. Also, π saves the descriptors with the timestamp smaller than the one in its descriptor. Finally, the process tries to enqueue into the root queue all these descriptors in the sorted order of their timestamps. Thus, the algorithm works in $O(P \log P)$ time. □

Lemma 2. *In each tree node `v` on the `Op` traversal path executing `Op` in `v` finishes in a finite number of steps.*

Proof. Consider an operation queue at node `v` (Fig. 5). Here some operations ($X_1 \ldots X_K$) should be executed before `Op`, while all other operations ($Y_1 \ldots$) will be executed only after execution of `Op` in `v` is fully completed. Thus:

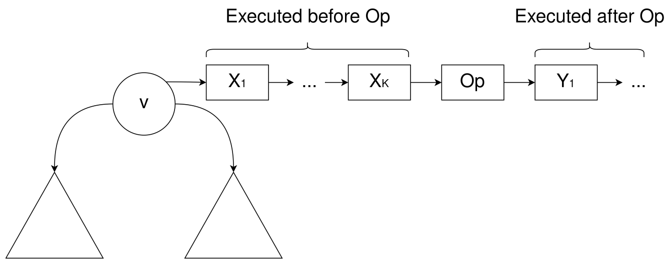

Fig. 5: Operation queue structure at node `v`

- We help to complete only a finite number of operations in a node `v`, since there cannot be more than $|P|$ operations in the queue of `v` before `Op` (where P is the set of the processes executing operations);
- Each operations X_i takes a finite number of steps to complete its execution in a node `v` (see Section II-C for the list of those steps). Note, that in the process of execution operation `Op` in node `v` we never retry any operation (in contrast to lock-free algorithms, e.g., in [29]): for example if the insertion of `Op` descriptor to child node `cv` fails, we conclude that `Op` descriptor has

been inserted to `cv` by another helper process and merely continue the execution of `Op` in `v`;

Therefore, executing `Op` in `v` finishes in a constant number of steps. □

Lemma 3. *Propagating the descriptor downwards, from the root to the appropriate lower nodes finishes within a bounded number of steps*

Proof. Consider some operation Op_2 such that Op_2`.Timestamp > Op.Timestamp`. If both Op_2 and `Op` are willing to change the very same tree node `v`, Op_2 under any conditions will do it after `Op`, since the operations are executed in a strict timestamp order (see Section II-A for details). Thus, Op_2 cannot somehow change the structure of the tree to disrupt `Op`'s traversal. Therefore, `Op` will finish its traversal in a constant amount of steps, since later operations cannot interfere in `Op` traversal. Since none of the later operations can overcome `Op`, we note the following:

- At the moment when `Op` begins execution the size of tree is N and no more than $|P|$ concurrent processes are inserting new nodes in the tree. Thus at the `Op.Timestamp` moment the size of the tree will no exceed $O(N + |P|)$, which is definitely a finite number;
- By Lemma 2 operations takes a finite number of steps to execute in a node.

Thus, the operation takes a finite number of steps to finish its traversal. □

Note, that our rebuilding procedure does not fail the wait-freedom guarantee in the proof above since each rebuilding finishes in a bounded number of steps.

Lemma 4. *The rebuilding procedure finishes in a bounded number of steps*

Proof. Indeed, the rebuilding procedure of a subtree `vs` consists of the following steps:

- Traverse the subtree `vs`, collecting all unfinished operations;
- Help to complete all these unfinished operations;
- Collect all keys from `vs`;
- Build an ideal tree from collected keys.

Note, that only the operations that started before `Op` can be unfinished in `vs` (Fig 6), since we execute operations in the timestamp order.

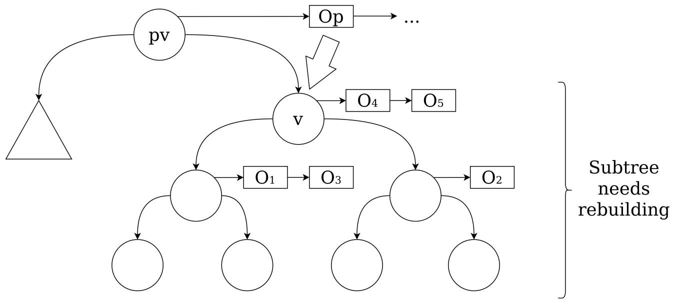

Fig. 6: Unfinished operation $O_1, O_2, \dots O_5$ have timestamp lower than `Op.Timestamp`

Therefore: 1) there is a finite set of unfinished operations in `vs`; 2) a completing of each unfinished operations takes a finite number of steps by Lemma 3; 3) `vs` has a finite size, thus, the collecting all keys from `vs` and the construction of a new ideal subtree also takes a finite amount of steps. Thus, the rebuilding completes in a finite amount of steps. □

G. Time cost analysis

We now estimate the time it takes to execute an operation in our solution.

Theorem 3. *The amortized cost of* `insert`*,* `remove`*,* `contains` *or* `count` *operations on our concurrent binary search tree with rebuilding is* $O((\log N + |P|) \cdot |P|)$ *where* N *is the size of the tree at the start of an operation and* $|P|$ *is the number of processes.*

Proof. In a sequential setting each of these operations takes $O(\log N)$ time since it visits $O(\log N)$ nodes performing $O(1)$ operations in each node. In concurrent setting, up to $|P|$ other processes can be inserting their keys to the tree concurrently with `Op`, thus, at the moment of `Op.Timestamp` the size of the tree will not exceed $N + |P|$, therefore the amortized number of nodes `Op` will traverse is $O(\log N + |P|)$ (since the tree is balanced).

In each node `v` no more than $|P|$ descriptors will be located closer to the head of `v.Queue` than the descriptor of our operation `Op`. Each operation takes $O(1)$ amortized time to execute (the rebuilding takes $O(1)$ amortized time as stated e.g., in [28]), thus, `Op` takes $O(|P|)$ amortized time to finish its execution in each node.

Therefore, amortized `Op` execution cost is $O((\log N + |P|) \cdot |P|)$. Assuming that the number of processes $|P|$ is constant and does not change during the execution we cant state that the execution cost is $O(\log N)$, i.e., logarithmic. □

Theorem 4. *When the workload is uniform (i.e., each argument is chosen from a fixed range uniformly at random)* `insert`*,* `remove`*, and* `contains` *take* $O(\log N + |P|)$ *amortized time on our concurrent binary search tree with rebuilding.*

Proof. Consider the size of the root operation queue. Since there exist up to $|P|$ processes executing operations concurrently, the size of root operation queue is $O(|P|)$.

Let us see, in which nodes these operations will continue their execution. Since each data item is equally likely to be queried, approximately half descriptors continues their execution in `root.Left` node, and the other half continues their execution in `root.Right` node. Therefore, the expected size of operation queue in each node of the second tree level is $O(\frac{|P|}{2})$.

Following the same reasoning, the expected size of operation queue in each node of the third tree level is $O\left(\frac{|P|}{2^2}\right) = O\left(\frac{|P|}{4}\right)$ and the expected size of operation queue in each node of the k-th level of the tree is $O\left(\frac{|P|}{2^{k-1}}\right)$.

Since the tree is balanced, the operation traverses $O(\log N + |P|)$ nodes. The expected amortized number of operations performed in i-th node is $O\left(\max\left(\frac{|P|}{2^{k-1}}, 1\right)\right)$ since the amortized cost of executing a single operation in a node is $O(1)$ (of course, in each node we perform at least $O(1)$ operations).

Therefore, the total expected amortized cost of performing an operation is $O\left(\sum_{k=1}^{\log N+|P|} \max\left(\frac{|P|}{2^{k-1}}, 1\right)\right) = O(\log N + |P|)$. □

III. Preliminary Experiments

Following the algorithm in Section II, we implemented a concurrent balanced BST that supports `insert`, `remove`, `contains`, and `count` queries, using Kotlin. For our implementation, we wanted to choose a JVM language due to the simple memory management while Kotlin is more expressive than Java and allows us to write less boilerplate code.

We now discuss implementation issues and provide a preliminary evaluation.

Firstly, we identified a significant overhead on update operations in the implementation. Consider the following experiment: we fix a range $[1; 2 \cdot 10^6]$, initialize a tree with a random half of the range, and then perform uniformly random `insert`/`delete` operations. In the sequential setting, our tree performs $350k$ random `insert`/`delete` operations per second, while a simple treap executes more than $420k$ such operations per second. We found two reasons behind this overhead: 1) in the process of moving the operation descriptor from the root downwards, we have to allocate a new queue node on each level of the tree, resulting in a large number of allocations; and 2) the number of indirections is large — to execute an operation, a lot of pointers must be traversed from tree nodes to queues, from queue nodes to descriptors, etc.

Nevertheless, our data structure provides unique guarantees: it is wait-free and can execute aggregate range queries asymptotically efficiently. Some concurrent trees do not support range queries at all [12], [18], [30], [31] so there is no point in comparing against them. Other non-blocking trees may support range queries [8], [9], [13], [19], [35], but the algorithm for an aggregate request is not efficient: the operation should collect all the required entries in a list and then run an operation on them. This makes the range query time to cost $O(N)$ for large ranges (e.g., when the operation asks for the full range) while our solution executes aggregate range queries in $o(N)$ time. Therefore, we cannot perform a fair comparison against such trees, as they execute range queries on large ranges much more slowly, even in comparison with our preliminary implementation.

We are aware of a single lock-free tree that supports asymptotically efficient range queries: the lock-free Persistent Tree (PT) [5]. However, this proposal does not scale on write-heavy workloads: as stated in [5] it scales due to hardware cache existence but this scalability is very limited and the performance starts degrading with the increased number of threads. Thus, to show the soundness of our approach, we aimed to show that it scales and outperforms the lock-free Persistent Tree on a write-heavy workload while performing quite well on read-heavy workloads. All the experiments were run on Intel Gold 6240R with 16 cores.

To start with we run a read-only workload: we fix a range $[1, 2 \cdot 10^6]$, initialize a tree with a random half of the range, and then each thread performs uniformly random `contains` operations. The number of `contains` operations per second are presented in the table below:

Threads	ops/sec (our tree)	Scalability (our tree)	ops/sec (PT)	Scalability (PT)
1	882k	1x	701k	1x
4	3352k	3.8x	2747k	3.9x
8	5609k	6.4x	5577k	7.9x
16	10477k	11.8x	11119k	15.9x

This experiment shows that PT scales better on this workload since it executes read-only queries with absolutely no synchronization while our solution has to execute the descriptor propagation logic.

The second experiment is the standard write-heavy workload: we fix a range $[1, 2 \cdot 10^6]$, initialize a tree with a random half of the range, and then perform uniformly random `insert`/`delete` operations. The number of executed operations per second are presented in the table below:

Threads	ops/sec (our tree)	Scalability (our tree)	ops/sec (PT)	Scalability (PT)
1	349k	1x	416k	1x
4	679k	1.9x	1031k	2.4x
8	931k	2.7x	1366k	3.3x
16	1056k	3x	886k	2.1x

As follows from this experiment, PT performance starts degrading when the number of threads reach 16. However, even at 16 threads our solution continues scaling (though not so swift as we were expecting). This allows us to outperform the lock-free Persistent Tree on 16 threads.

The third workload consists of `insert` operation of a random integer (i.e., the range consists of all possible integers) with success probability of almost 1.0 executed on a tree containing initially 10^6 random integers. The number of executed `insert` operations per second are presented in the table below:

Threads	ops/sec (our tree)	Scalability (our tree)	ops/sec (PT)	Scalability (PT)
1	239k	1x	301k	1x
4	424k	1.8x	637k	2.1x
8	573k	2.4x	670k	2.2x
16	651k	2.7x	505k	1.7x

Again, the Persistent Tree performance starts degrading when the number of threads reach 16. However, even at 16 threads our solution continues scaling, allowing us to outperform the lock-free PT.

IV. Conclusion

We presented an approach to obtain concurrent trees with efficient aggregate range queries in a wait-free manner. Our preliminary experiments show the potential of the improvement. We propose a number of avenues for future work. First, we can make the rebuilding collaborative [15], i.e., make different processes work together to rebuild a single subtree. Then, in order to achieve pure $O(\log n)$ complexity, instead amortized, we can use another rebuilding strategy, e.g., the top-down rebuilding from the chromatic tree [14]. Another interesting question is how to decrease the number of memory allocations. Finally, it would be interesting to extend to other tree data structures, e.g., quad-trees or tries.

Acknowledgement

We thank Trevor Brown and Yuanhao Wei for the discussion and anonymous reviewers for helping us to improve the paper. Also, we thank JetBrains and Huawei for their support.

References

[1] compare-and-swap, 2022. URL: https://en.wikipedia.org/wiki/Compare-and-swap.

[2] fetch-and-add, 2022. URL: https://en.wikipedia.org/wiki/Fetch-and-add.

[3] Persistent data structures, 2022. URL: https://en.wikipedia.org/wiki/Persistent_data_structure.

[4] Universally unique identifier, 2022. URL: https://en.wikipedia.org/wiki/Universally_unique_identifier.

[5] Vitaly Aksenov, Trevor Brown, Alexander Fedorov, and Ilya Kokorin. Unexpected scaling in path copying trees. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 438–440, 2023.

[6] Vitaly Aksenov, Ilya Kokorin, and Alena Martsenyuk. Parallel-batched interpolation search tree. In *International Conference on Parallel Computing Technologies*, pages 109–125. Springer, 2023.

[7] Vitaly Aksenov, Petr Kuznetsov, and Anatoly Shalyto. Parallel combining: Benefits of explicit synchronization. In *22nd International Conference on Principles of Distributed Systems*, 2019.

[8] Maya Arbel-Raviv and Trevor Brown. Harnessing epoch-based reclamation for efficient range queries. *ACM SIGPLAN Notices*, 53(1):14–27, 2018.

[9] 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 *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 357–369, 2017.

[10] Rudolf Bayer. R. bayer, e. mccreight organization and maintenance. *Software Pioneers: Contributions to Software Engineering*, 1:245, 2012.

[11] Ferenc Bodon and Lajos Rónyai. Trie: an alternative data structure for data mining algorithms. *Mathematical and Computer Modelling*, 38(7-9):739–751, 2003.

[12] Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. *ACM Sigplan Notices*, 45(5):257–268, 2010.

[13] Trevor Brown and Hillel Avni. Range queries in non-blocking k-ary search trees. In *International Conference On Principles Of Distributed Systems*, pages 31–45. Springer, 2012.

[14] Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 329–342, 2014.

[15] Trevor Brown, Aleksandar Prokopec, and Dan Alistarh. Non-blocking interpolation search trees with doubly-logarithmic running time. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 276–291, 2020.

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

[17] Mark Theodoor De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational geometry: algorithms and applications*. Springer Science & Business Media, 2000.

[18] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 131–140, 2010.

[19] Panagiota Fatourou, Elias Papavasileiou, and Eric Ruppert. Persistent non-blocking binary search trees supporting wait-free range queries. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 275–286, 2019.

[20] Goetz Graefe et al. Modern b-tree techniques. *Foundations and Trends® in Databases*, 3(4):203–402, 2011.

[21] Leo J Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, pages 8–21. IEEE, 1978.

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

[23] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 355–364, 2010.

[24] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[25] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The art of multiprocessor programming*. Newnes, 2020.

[26] Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. *ACM SIGPLAN Notices*, 46(8):223–234, 2011.

[27] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. In *Concurrency: the Works of Leslie Lamport*, pages 171–178. 2019.

[28] Kurt Mehlhorn and Athanasios Tsakalidis. Dynamic interpolation search. *Journal of the ACM (JACM)*, 40(3):621–634, 1993.

[29] Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275, 1996.

[30] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 317–328, 2014.

[31] Aleksandar Prokopec, Trevor Brown, and Dan Alistarh. Analysis and evaluation of non-blocking interpolation search trees. *arXiv preprint arXiv:2001.00413*, 2020.

[32] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

[33] Yihan Sun, Guy E Blelloch, Wan Shen Lim, and Andrew Pavlo. On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes. *Proceedings of the VLDB Endowment*, 13(2), 2019.

[34] Yihan Sun, Daniel Ferizovic, and Guy E Belloch. Pam: parallel augmented maps. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 290–304, 2018.

[35] 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 *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 31–46, 2021.

[36] Chaoran Yang and John Mellor-Crummey. A wait-free queue as fast as fetch-and-add. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–13, 2016.