The Academic College of Tel Aviv–Yaffo

MTASet: A Tree-based Set for Efficient Range Queries in Update-heavy Workloads

By: Daniel Manor

Supervisor: Dr. Moshe Sulamy

Submitted in partial fulfillment of the requirement for an MSc degree in Computer Science

July 2024

Contents

List of Figures

List of Algorithms

Abstract

Numerous concurrent set implementations are optimized and fine-tuned to excel in scenarios characterized by predominant read operations. However, they often perform poorly when confronted with workloads that heavily prioritize updates. Additionally, current leading-edge concurrent sets optimized for update-heavy tasks typically lack efficiency in handling atomic range queries. This study introduces the MTASet, which leverages a concurrent (a,b)-tree implementation. Engineered to accommodate update-heavy workloads and facilitate atomic range queries, MTASet surpasses existing counterparts optimized for tasks in range query operations by up to 2x. Notably, MTASet ensures linearizability.

1 Introduction

Given the inherent challenges of concurrent programming, developers often use various concurrent data structures to build applications and complex systems, such as modern database engines designed for multicore hardware. These structures enable safe utilization in multithreaded environments through sophisticated synchronization algorithms optimized for performance.

The rise of multicore hardware has spurred the development of numerous new concurrent data structure designs, including dictionaries [3, 4, 8, 10, 11, 12, 13, 16] and sets [7, 22, 24]. These innovations consistently enhance performance over existing solutions and introduce features like atomic range scan operations.

Existing concurrent set or dictionary implementations typically excel in scenarios with low contention and predominantly read-oriented workloads, often neglecting the demands of update-intensive environments. Conversely, implementations optimized for update-heavy workloads frequently struggle with efficient range queries. For example, in experiments by Kobus et al. [16], SnapTree [8] performs relatively well on update operations but exhibits poor performance on scan operations. Our research aims to address this issue by enhancing the scalability of range queries within a concurrent set optimized for update-heavy workloads, ensuring robust performance across diverse workload types. Scaling range queries is particularly challenging due to the frequent and concurrent modifications inherent in update-heavy workloads.

In response, we introduce MTASet, a concurrent set with high update throughput that stores keys and their associated values and supports essential operations such as insertion, deletion, and lookup. In addition to high update throughput, MTASet is optimized for atomic range queries, which retrieve values for a specified range of keys.

MTASet uses a tailored multi-versioning approach [5] for atomic range queries, maintaining only the versions required for ongoing scans and managing version numbers through scans rather than updates. This significantly enhances the throughput of range query operations, especially under concurrent update-heavy workloads. Inspired by the KiWi Map [4], MTASet's range query operation demonstrates substantial performance gains in experimental evaluations, outperforming many state-of-the-art data structures in both read-mostly and update-heavy workloads.

MTASet is an (a,b)-tree, a variant of B-trees that allows between a and b keys per

node, where $a \leq \frac{b}{2}$. It is based on a concurrent version of Larsen and Fagerberg's relaxed (a,b)-tree [18], specifically the OCCAB-TREE [26]. MTASet employs fine-grained versioned locks to ensure atomic sub-operations and uses version-based validation in leaf nodes to guarantee correct searches. To manage overhead, MTASet incorporates established techniques, such as avoiding key sorting in leaves and minimizing unnecessary node copying.

The core philosophy of MTASet is to promptly handle client operations while deferring data structure optimizations to an occasional maintenance procedure. This procedure, called rebalance, aims to balance MTASet's (a,b)-tree for faster access and to eliminate obsolete keys through compaction.

In this paper, we study the question of how to scale range queries on a Set that is optimized for update-heavy workloads, ideally without sacrificing update performance.

Evaluation results: The MTASet Java implementation can be found on GitHub [20]. In Section 5, we benchmark its performance under various representative workloads. In most experiments, it significantly surpasses the OCCABTree [26], tailored for update-heavy workloads with atomic range query capabilities [2]. This positions MTASet as a concurrent set optimized for update-heavy tasks, offering efficient, atomic, and wait-free range queries.

The benefits of MTASet are evident in our primary scenario, which includes long scans amid concurrent update operations. MTASet did not outperform competitors [4] optimized solely for range scans but not for updates. However, in updates, MTASet significantly outperformed them, up to three times. In scenarios involving long scans with concurrent puts, MTASet exceeded the performance of the OCCABTree [2, 26] by up to three times while maintaining comparable performance in update operations, thus preserving its update-heavy nature. Notably, MTASet's atomic scans are 1.6 times faster than the non-atomic scans offered by the Java Skiplist written by Doug Lea [19] based on work by Fraser and Harris [14], and MTASet's updates are up to 3.6 times faster than those of the Java Skiplist.

Contributions: The introduction of MTASet significantly improves the range query operation of concurrent sets optimized for update-heavy workloads.

MTASet supports the following operations:

- find(k): Checks if a key-value pair with the key k exists. If it does, the associated value is returned; otherwise, it returns $\perp$.
- Insert(k, v): Verifies if a key-value pair with the key k exists. If it does, it returns the associated value; otherwise, it inserts the key-value pair and returns $\perp$.
- delete(k): Deletes the key-value pair with the key k if it exists and returns the associated value. Otherwise, it returns $\perp$.
- scan(fromKey, toKey): Returns the values of the keys within the range [fromKey, toKey].

2 Background

2.1 Set

A set data structure is a collection that stores unique elements that may not be in any particular order. Sets can be implemented in various ways, including using hash tables or binary search trees, which affects their operations' performance.

2.1.1 Set Operations

- Insertion (add): Adding an element to a set involves checking if the element already exists. If it does not, the element is added. In binary search tree implementations, this operation typically has an average time complexity of $O(\log n)$.
- Deletion (remove): Removing an element involves finding and then deleting the element. This operation in a binary search tree has an average time complexity of $O(\log n)$.
- Membership Test (contains): Checking if an element is present in the set involves searching for it. The time complexity is $O(\log n)$ for binary search trees.

2.2 (a,b)-Tree Data Structure

An (a, b)-tree [6] is a balanced search tree generalizing the B-tree, where each node can have between a and b children, and $2 \leq a \leq \frac{b}{2}$. This tree structure optimizes operations by maintaining logarithmic height with respect to the number of elements, ensuring efficient data retrieval, insertion, and deletion.

2.2.1 Operations on an (a,b)-Tree

- Insertion: To insert a new element, start from the root and find the appropriate leaf node. Add the new element if the node has fewer than b elements. If it has b elements, split it into two nodes and promote the middle element to the parent. This operation has a time complexity of $O(\log n)$.
- Deletion: Deleting an element involves finding and removing it. Adjustments such as borrowing elements from adjacent nodes or merging nodes ensure the tree remains balanced. This operation also has a time complexity of $O(\log n)$.

- Search: Searching for an element involves traversing from the root to the appropriate leaf node. Due to the balanced nature of the tree, this operation has a time complexity of $O(\log n)$.

The balanced structure of the (a, b)-tree ensures logarithmic time complexity for all operations, making it suitable for applications requiring frequent insertions, deletions, and lookups, such as database indexing.

MTASet utilizes an (a,b)-tree data structure, specifically the OCC-ABTREE [26], a concurrent (a,b)-tree optimized for workloads with frequent updates. Unlike the original (a,b)-tree structure, the OCC-ABTREE supports concurrent operations and includes optimizations tailored for update-heavy workloads. Initially, the OCC-ABTREE did not include a built-in range query operation. However, it was noted in [26] that a range query capability could be implemented for the OCC-ABTREE using a specified technique or approach detailed in [2], which we will briefly explain:

In the OCC-ABTREE, leaf nodes are interconnected in a linked list, with each leaf node storing keys along with insertionTime and deletionTime fields indicating when keys were added and removed, respectively. A global variable, TS, is incremented atomically by a range query from time t to t'. During insertion, TS is read and written to the insertionTime field of the new key atomically. Similarly, during deletion, TS is read and written to the deletionTime field of the deleted key, which is then stored in the thread that executes the deletion, list of deleted keys accessible for other threads to read. Special precautions are taken during deletion to prevent race conditions. A range query traverses leaf node lists, collecting keys with insertionTime less than or equal to t. It subsequently checks thread-specific lists for keys deleted after time t, using each key's deletionTime to identify missed deletions during traversal.

MTASet introduces significant improvements, detailed in Section 5, to enhance the range query operation throughput while maintaining the performance advantages of the OCC-ABTREE on update-heavy workloads.

3 Related Work

3.1 Skip Lists

KiWi [4] Key-Value Map that supports linearizable wait-free range scans. It utilizes a multi-versioned architecture and CAS-based operations to ensure lock-free functionality.

LeapList [3] provides linearizable range scans and relies on fine-grained locks and Software Transactional Memory for concurrency control.

Jiffy [16] is a linked-list data structure that supports arbitrary snapshots well as atomic batch updates.

Nitro [17] employs multiversioning to generate snapshots, but creating a new snapshot is not a thread-safe operation, meaning it cannot be performed simultaneously with put/remove operations.

3.2 Trees

OCC-ABtree [26] is a concurrent (a,b)-tree designed for update-heavy workloads. Though it doesn't support range scan, a general range scan method [2] has been proposed to implement it.

SnapTree by Bronson et al. [8] is a lock-based relaxed balance AVL tree. It utilizes a linearizable clone operation to ensure atomic snapshots and range scans.

Minuet [25] is a distributed, in-memory B-tree that supports linearizable snapshots. Although it creates snapshots using a relatively costly copy-on-write method, Minuet enables them to be shared across multiple range scans.

BCCO10 [8] presents a Binary Search Tree (BST) employing optimistic concurrency control for thread synchronization. They introduce a sophisticated handover-hand version number-based validation technique to achieve efficient search operations.

LF-ABtree [9] is a lock-free (a,b)-tree based on the same relaxed (a,b)-tree [18] as MTASet.

3.3 Range query techniques

[2] explained the implementation of range queries in concurrent set data structures using epoch-based memory reclamation. They propose a traversal algorithm that ensures every item within a specified range, present throughout the traversal's lifetime, is visited. [23] presented a technique for implementing linearizable range queries on lock-based linked data structures.

4 MTASet

MTASet contains a permanent entry pointer to a sentinel node. This sentinel node has no keys and only one child pointer, which points to the root node.

A node is $underfull$ if it contains fewer keys than the minimum allowed (denoted as a in the (a,b)-tree), and it is $full$ when the number of keys it contains equals the maximum allowed (denoted as b in the (a,b)-tree).

Below is the pseudocode along with explanations for the data structures used in MTASet and the supported operations.

4.1 Data structures

The MTASet has three types of nodes: leaf nodes, internal nodes and tagged internal nodes.

Each node possesses a lock field, with our lock implementation being MCS locks [21]. In MCS locks, threads awaiting the lock form a queue and spin on a local bit, facilitating efficient scaling across multiple NUMA nodes. In our tree structure, a thread modifies a node only if it holds the corresponding lock. Leaf nodes include an additional field called "version," which tracks the number of modifications made to the leaf and indicates whether it is currently changing. Upon acquiring a leaf's lock, a thread increments the version before initiating modifications. Subsequently, the version is incremented once it has completed its changes before releasing the lock. As a result, the version remains even when the leaf is not being modified and odd when it is being modified. Searches utilize this version field to ascertain whether any modifications occurred while reading the keys of a leaf. Furthermore, nodes contain a marked bit, toggled when a node is unlinked from the tree. This allows updates to determine whether a node is present in the tree. Once marked, nodes are never unmarked.

The search operation returns the PathInfo structure, which provides information about the node at which the search terminated, its parent and grandparent, the index of the node in the parent's pointers array, and the parent index in the grandparent's pointers array.

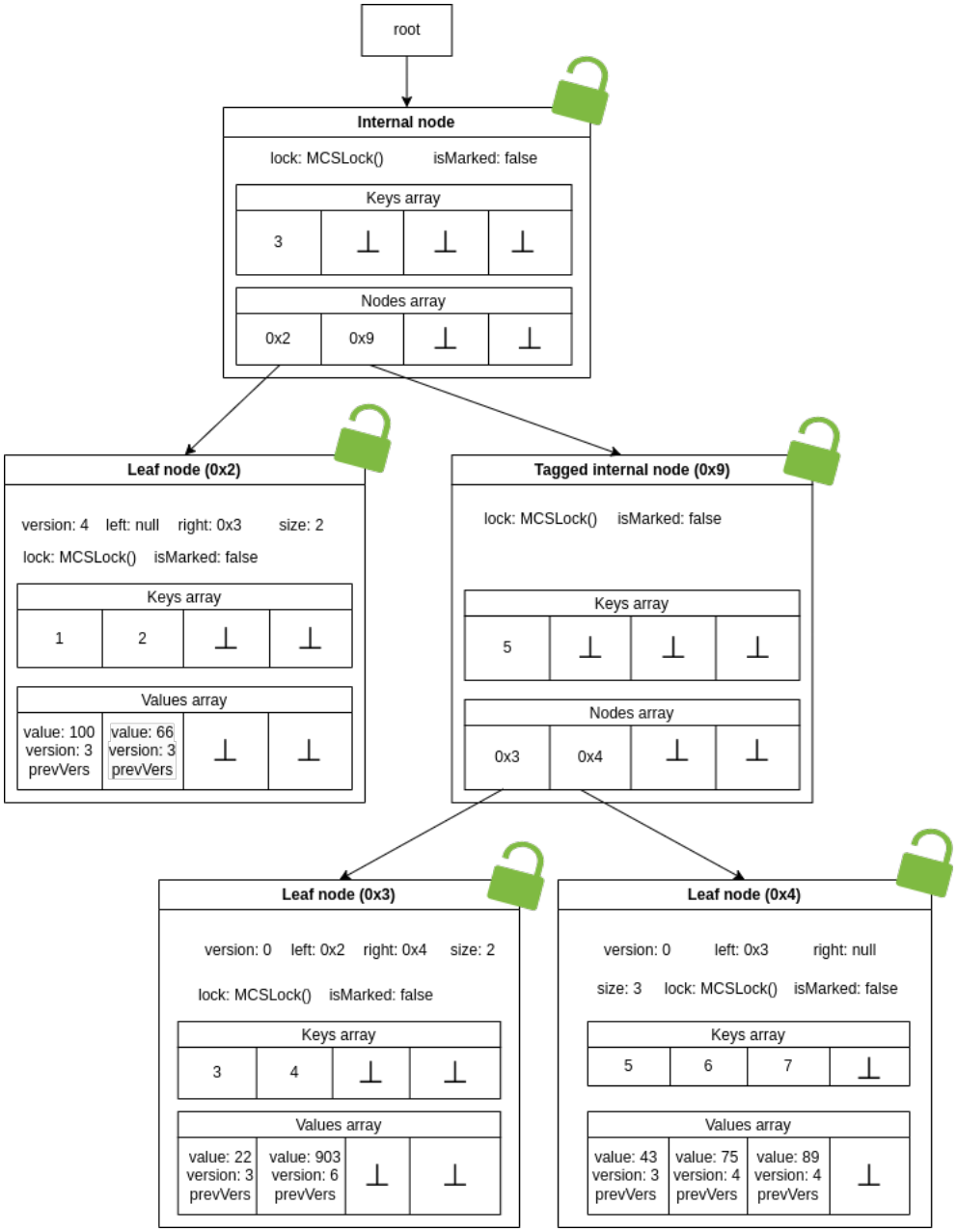


Figure 1: A snapshot of MTASet
An internal node pointing to a tagged internal node and a leaf node. The tagged internal node points to two leaf nodes. No locks are acquired.

4.1.1 Leaf Nodes

Leaf nodes contain keys and values stored within their respective arrays. An entry within the keys array is considered empty when represented as $\perp$ and does not have a corresponding value, as shown in leaf nodes in figure 1. The keys within a leaf node are unordered, and empty slots may exist between them. This characteristic facilitates quicker updates as insertions and deletions do not necessitate the rearrangement of other keys within the node. Adjacent leaf nodes are interconnected through left and right pointers, utilized during traversal scan operations.

Values are versioned, meaning they retain both the value for the latest version and values for past versions. A value could be $\perp$, indicating a logical deletion in the corresponding version, or a non-zero value.

4.1.2 Internal Nodes

Internal nodes have two sorted arrays: one holding k child pointers and the other holding $k-1$ routing keys, which direct searches to the correct leaf. These routing keys remain constant. Adding or removing a key necessitates replacing the entire internal node, which occurs relatively infrequently. On the other hand, child pointers are mutable and subject to change.

4.1.3 Tagged-Internal Nodes

A TaggedInternal node represents a temporary height imbalance within the tree. It exists when a key/value insertion is required into a full node. Upon splitting the node, the two resulting halves are connected by a tagged node. Tagged nodes stand alone and are not involved in any other operations, consistently having precisely two children. They are eventually eliminated from the tree by invoking the fixTagged rebalancing step, described below in Section 4.2.12.

4.1.4 Value cells

Value cells are the objects stored inside each leaf's value array that encapsulate the values corresponding to keys. The most recent value and version are stored in specific variables, while previous values are maintained in a linked list.

4.1.5 Coordination data structure

MTASet utilizes a data structure to coordinate scan and rebalancing operations. This data structure is a global array called the ongoing scans array (OSA), which keeps track of the versions of ongoing scans used by rebalancing for compaction purposes. The global OSA array is employed by the scan (Algorithm 10) and cleanObsoleteKeys (Algorithm 18) operations, and its usage is detailed in the description of each operation in Section 4.2.

4.1.6 Linked-list of Leaf Nodes

Leaf nodes contain left and right pointers, directing to their adjacent nodes from the left and right sides. This setup forms a linked list of leaf nodes with the property: for each leaf 'l' within the linked list, the keys in 'l.right' are strictly greater than those in 'l.' Rebalancing procedures, which involve linking and unlinking leaf nodes due to occasional underfull or full conditions, ensure that at any given time it is possible to reach the right-most leaf node from the left-most leaf node. The Linked List of Leaf Nodes aims to facilitate the scan operation, enabling it to traverse the leaf nodes directly without traversing the entire tree, as these are the only ones containing values.

```
abstract type Node:
  keys    : K[MAX_SIZE]
  lock    : MCSLock
  size    : int
  marked  : bool

type Internal extends Node:
  nodes : Node[MAX_SIZE]

type Leaf extends Node:
  values   : ValueCell[MAX_SIZE]
  version  : int
  left     : Node
  right    : Node

type ValueCell:
  value            : V
  version          : int
  previousVersions : BinarySearchTree<int,V>

type PathInfo:
  grandParent    : Node
  parent         : Node
  parentIndex    : int
  node           : Node
  nodeIndex      : int

Internal entry // sentinel node
int MAX_SIZE
int MIN_SIZE
int GLOBAL_VERSION
```

Figure 2: MTASet Data structures

4.2 Operations

Each operation invokes the Search function (Algorithm 1). It accepts a key k and optionally a reference to a target node tn, returning a PathInfo object. This PathInfo object contains a reference to the node n, which is either intended to store the specified key k or is the node referred to by tn. Additionally, PathInfo includes references to n's parent and grandparent nodes. This function explores the tree from the root, seeking the key. The search determines the appropriate child pointer at each internal node by sequentially traversing the sorted routing keys. Upon reaching a leaf (or the designated target node), it yields a PathInfo object.

The function searchLeaf (Algorithm 2) locates a specified key k within a leaf node l and attempts to retrieve the corresponding value if k exists in l. Drawing inspiration from the classical double-collect snapshot algorithm [1], it executes as follows: Initially, it reads the version of the leaf l. Then, it scans through l's keys array to locate k. Afterward, it re-reads the version of l to verify that no modifications occurred while retrieving the key and its associated value. If concurrent updates are detected, a retry is initiated. If no concurrent updates are detected and k is found, searchLeaf returns $(SUCCESS, \text{value})$. If k is not found, it returns $(FAILURE, \perp)$.

Notably, both the search and searchLeaf functions are engineered to operate without acquiring locks. This methodology enhances concurrency by permitting updates to internal nodes while searches are in progress, thereby optimizing performance in environments characterized by frequent read and write operations.

4.2.1 Find

The find(key) (Algorithm 3) operation straightforwardly calls upon the Search and SearchLeaf functions and returns a value.

Algorithm 1 Search

```
1: function SEARCH(key, targetNode)
2:     gp ←⊥, p ←⊥, pIdx ← 0, n ← entry, nIdx ← 0
3:     while n is not Leaf do
4:         if n = targetNode then
5:             break
6:         end if
7:         gp ← p, p ← n, pIdx ← nIdx, n ← entry, nIdx ← 0
8:         while nIdx < node.size - 1 AND key ≥ node.keys[nIdx] do
9:             nIdx++
10:        end while
11:        n ← n.ptrs[nIdx]
12:    end while
13:    return PathInfo(gp, p, pIdx, n, nIdx)
14: end function
```

Algorithm 2 Search Leaf

```
1: function SEARCHLEAF(leaf, key)
2:     while True do
3:         ver1 ← leaf.ver
4:         if ver1 is odd then
5:             continue
6:         end if
7:         value ←⊥
8:         for i ← 0 to NodeMaxSize - 1 do
9:             if leaf.keys[i] = key then
10:                value ← leaf.values[i].latestValue
11:                break
12:            end if
13:        end for
14:        ver2 ← leaf.ver
15:        if ver1 ≠ ver2 then
16:            continue
17:        end if
18:        if value = ⊥ then
19:            return (FAILURE, ⊥)
20:        else
21:            return (SUCCESS, value)
22:        end if
23:    end while
24: end function
```

Algorithm 3 Find

```
1: procedure FIND(key)
2:     pathInfo ← PathInfo()
3:     searchResult ← SEARCH(key, null, pathInfo)
4:     if searchResult.getReturnCode() ≠ ReturnCode.SUCCESS then
5:         return Result(ReturnCode.FAILURE)
6:     end if
7:     leaf ← pathInfo.n
8:     searchLeafResult ← SEARCHLEAF(leaf, key)
9:     return searchLeafResult
10: end procedure
```

4.2.2 Insert

During the insert(key, value) operation (Algorithm 4), a thread starts by executing a search(key, target) and searchLeaf functions. The operation returns the associated value if the key is found during this search. Otherwise, it proceeds to lock the leaf and tries to insert the key (along with its corresponding value) into an available empty slot within the keys and values array. This process is known as a simple insert. However, if no empty slot is found, and considering that keys may become obsolete due to logical removals, the insert operation then checks for keys that can be physically removed by invoking the cleanObsoleteKeys function (Algorithm 18). If obsolete keys are indeed removed, the new key is inserted, and the fixUnderfull function (Algorithm 15) is called to ensure that the node remains at least as large as the minimum size allowed.

If no obsolete keys are removed, the insert operation locks the leaf's parent and replaces the pointer to the leaf with a pointer to a newly created tagged node. This tagged node points to two new children: one containing the contents of the original leaf and the other containing the newly inserted key-value pair. This scenario is termed a splitting insert. The modification of the pointer, and thus the insertion of the key, occurs atomically. Following this, the insert operation invokes fixTagged (Algorithm 14) to eliminate the tagged node from the tree.

4.2.3 Insert Key

The insertKey (Algorithm 5) function accepts a key, a value, and a leaf node as inputs. It writes the given key and value to the specified leaf node and attempts to atomically

assign a version to the inserted value. If the leaf node is full, the function returns RETRY, signaling to the caller that the insertion failed. If the insertion is successful, it returns the inserted value.

4.2.4 Update Key

The updateKeyInIndex function (Algorithm 6) accepts an index, a value, and a leaf node as inputs. It writes the value to the specified leaf node at the given index and attempts to atomically assign a version to the inserted value. This function is used when the caller knows the exact cell for insertion. It is used when a key is logically deleted, but not physically, and a new key-value pair needs to be inserted.

4.2.5 Can Update Key

The canUpdateKeyInIndex function (Algorithm 7) takes a key, value, and leaf node as inputs and determines whether it is possible to update the corresponding value for the key. Updating the key is possible only if the key is logically deleted or if the key exists and needs to be (logically) deleted. The function returns (true, i) if the key can be updated at index i, (false, i) if it cannot be updated at index i, and (false, $\perp$) if the key does not exist.

4.2.6 Create Tagged Internal Node

The createTaggedInternalNode function (Algorithm 8) takes a key, value, leaf, the index of the leaf in its parent node's array, and the parent node as inputs. It creates a tagged internal node that points to two new leaf nodes, which evenly distribute the given key along with all the keys from the specified leaf node. Additionally, it connects the new leaf nodes to the linked list of leaf nodes.

Algorithm 4 Insert Operation

```
1: function INSERT(key, value)
2:     while True do
3:         path = SEARCH(key, NULL)
4:         retCode, retValue = SEARCHLEAF(path.node, key)
5:         if retCode = SUCCESS and value ≠⊥ or retCode = FAILURE and value = ⊥ then
6:             return retValue
7:         end if
8:         leaf, parent = path.node, path.parent
9:         LOCK(leaf)
10:        if leaf.marked then
11:            Unlock leaf
12:            Continue
13:        end if
14:        canUpdateKeyInIndexResult ← CANUPDATEKEYININDEX(key, value, leaf)
15:        if canUpdateKeyInIndexResult.Result then
16:            updateKeyInIndexResult ← UPDATEKEYININDEX(canUpdateKeyInIndexResult.Index, value, node)
17:            Unlock leaf
18:            return updateKeyInIndexResult
19:        else if canUpdateKeyInIndexResult.Index ≠ ⊥ then
20:            Unlock leaf
21:            return ⊥
22:        end if
23:        currSize ← leaf.size
24:        if currSize < this.maxNodeSize then
25:            result ← INSERTKEY(key, value, leaf)
26:            Unlock leaf
27:            return result
28:        else
29:            removedKeys ← CLEANOBSOLETEKEYS(leaf)
30:            if removedKeys > 0 then
31:                writeResult ← INSERTKEY(key, value, leaf)
32:                UNLOCK(leaf)
33:                FIXUNDERFULL(leaf)
34:                return writeResult
35:            end if
36:            if leaf.left and/or leaf.right parent is not equal to leaf's parent then
37:                lock and check if marked. If marked, Unlock leaf.left and/or leaf.right, leaf, parent and Continue
38:            end if
39:            newTaggedInternal ← CREATETAGGEDINTERNALNODE(key, value, leaf, path.nodeIndex, parent)
40:            Unlock leaf and parent
41:            fixTagged(newTaggedInternal)
42:            return ⊥
43:        end if
44:    end while
45: end function
```

Algorithm 5 Insert key

```
1: function INSERTKEY(key, value, node)
2:     for i ← 0 to this.maxNodeSize − 1 do
3:         if node.keys[i] = ⊥ then
4:             node.version ← node.version + 1
5:             vc ← new ValueCell(key)
6:             vc.putNewValue(value)
7:             node.values[i] ← vc
8:             node.keys[i] ← vc.key
9:             node.values[i].casLatestVersion(0, GlobalVersion.Value.get())
10:            node.size ← node.size + 1
11:            node.version ← node.version + 1
12:            return node.values[i].latestValue
13:        end if
14:    end for
15:    return RETRY
16: end function
```

Algorithm 6 Update key

```
1: function updateKeyInIndex(keyIndex, value, node)
2:     node.version ← node.version + 1
3:     vc ← node.values[keyIndex]
4:     vc.putNewValue(value)
5:     vc.casLatestVersion(0, GlobalVersion.Value.get())
6:     node.version ← node.version + 1
7:     return vc.latestValue
8: end function
```

Algorithm 7 Can Update Key in Index

```
1: function CanUpdateKeyInIndex(key, value, leaf)
2:     for i ← 0 to this.maxNodeSize − 1 do
3:         if leaf.keys[i] = key then
4:             if leaf.values[i].latestValue ≠⊥ and value = ⊥ or leaf.values[i].latestValue
   = ⊥ and value ≠⊥ then
5:                 return (true, i)
6:             else
7:                 return (false, i)
8:             end if
9:         end if
10:    end for
11:    return (false, ⊥)
12: end function
```

Algorithm 8 Create Tagged Internal Node

```
1: function createTaggedInternalNode(key, value, leaf, leafIdxInParent ,parent)
2:     N ← contents of leaf ∪ {key/value}
3:     initialize two leaf nodes, child1 and child2
4:     newTaggedInternal ← TaggedInternal with two children: child1 and child2, who
   evenly shares N
5:     child1.right ← child2
6:     child1.left ← leaf.left
7:     child2.left ← child1
8:     child2.right ← leaf.right
9:     leaf.right.left ← child2
10:    leaf.left.right ← child1
11:    parent.ptrs[leafIdxInParent] ← newTaggedInternal
12:    leaf.marked ← true
13:    return newTaggedInternal
14: end function
```

4.2.7 Delete

Deleting a key (Algorithm 9) involves writing (key, $\perp$) by calling the Insert function. If a key is not found or has already been logically deleted, $\perp$ is returned. In case the key exists, the thread duplicates the current latest value into the key's version history data structure, sets the latest value with $\perp$, and then updates its version using CAS.

Algorithm 9 Delete

```
1: function DELETE(key)
2:     return INSERT(key, ⊥)
3: end function
```

4.2.8 Scan

In the scan(lowKey, highKey) operation (Algorithm 10), a thread initially performs an atomic fetch-and-add operation on the GV (Global Version) global variable to increment its value. The obtained version is then published by writing it to the global, ongoing scan array (OSA). The thread also synchronizes with the rebalancing operation by atomically attempting to write the read global version using CAS (compare-and-swap) (Algorithm 12). Upon invoking the search operation, the thread identifies the node intended to contain lowKey. From this node, using the scanLeaf function (Algorithm 11), it traverses the leaf nodes, reading the values corresponding to keys within the [lowKey, highKey] range. These values meet the criteria of having the latest version equal to or less than the version in the OSA and are not $\perp$. Throughout this traversal, the thread ensures that the collected values are sorted by their keys in ascending order before being copied to the result array. The scan terminates by not proceeding to the next node upon encountering a key whose value exceeds highKey or upon reaching the end of the traversal path. Finally, the scan information is removed from the OSA by writing $\perp$ to the appropriate cell. The operation then returns an array containing the scanned values along with its size.

4.2.9 ScanLeaf

The Scan Leaf function (Algorithm 11) takes a leaf node, a version, and a Scan operation range defined by low and high values as inputs. It scans through the leaf's key array to collect keys within the [low, high] range that have a value with a version less than or equal to the specified version. If there are multiple versions meeting this criterion, it

selects the largest one. The function returns an array containing the collected keys and their corresponding values. Additionally, it provides a flag indicating whether any keys exceeding the high value were encountered during the scan.

Algorithm 10 Scan

```
1:  function Scan(low, high, entry, result)
2:      myVer ← newVersion(low, high)
3:      resultSize ← 0
4:      pathInfo ← search(low, ⊥)
5:      leftNode ← pathInfo.n]
6:      while true do
7:          scanLeafResult ← scanLeaf(leftNode, myVer)
8:          leafKvArray ← scanLeafResult.kvArray
9:          leafKvArraySize ← scanLeafResult.kvArraySize
10:         continueToNextNode ← scanLeafResult.continueToNextNode
11:         sortByKeyAsc(leafKvArray)
12:         for i ← 0 to leafKvArraySize - 1 do
13:             result[resultSize] ← leafKvArray[i].value
14:             resultSize++
15:         end for
16:         if continueToNextNode AND leftNode.right ≠⊥ then
17:             leftNode ← leftNode.right
18:         else
19:             break
20:         end if
21:     end while
22:     publishScan(⊥)
23:     return resultSize
24: end function
```

Algorithm 11 Scan Leaf

```
1:  function scanLeaf(leaf, version, low, high)
2:      kvArray ← new KeyValue[maxNodeSize]
3:      kvArraySize ← 0
4:      continueToNextNode ← true
5:      for i ← 0 to maxNodeSize - 1 do
6:          key ← leftNode.keys[i]
7:          if key = ⊥ then
8:              continue
9:          end if
10:         valueCell ← leftNode.values[i]
11:         if valueCell = ⊥ then
12:             continue
13:         end if
14:         if key ≥ low AND key ≤ high then
15:             value ← valueCell.helpAndGetValueByVersion(myVer)
16:             if value = ⊥ then
17:                 continue
18:             end if
19:             kvArray[kvArraySize] ← (key, value)
20:             kvArraySize++
21:         end if
22:         if key > high then
23:             continueToNextNode ← false
24:         end if
25:     end for
26:     return (kvArray, kvArraySize, continueToNextNode)
27: end function
```

4.2.10 Create New Version

The createNewVersion function (Algorithm 12), utilized by the Scan (Algorithm 10) operation, reads and increments the Global Version. It then creates and stores an object in the OSA at the cell index matching the executing thread ID. This object holds the retrieved version, indicating that a scan, linearized at the read version, is currently running.

Algorithm 12 Create New Version

```
1:  function NewVersion(scanData)
2:      publishScan(scanData)                ▷ insert scanData to shared array (OSA)
3:      myVer ← GV.getAndIncrement()
4:      if CAS(scanData.version,0, myVer) then                ▷ Compare-and-Swap
5:          return myVer
6:      else                                 ▷ version was already set by a different thread
7:          helpedVer ← scanData.version
8:          return helpedVer
9:      end if
10: end function
```

4.2.11 Help and Get Value by Version

The helpAndGetValueByVersion function (Algorithm 13) accepts a ValueCell and a version as inputs. It returns the value with a version equal to or less than the specified version. If it encounters a value without a version, it attempts to assign one by reading from the global version and using a CAS (Compare-And-Swap) operation.

Algorithm 13 HelpAndGetValueByVersion

```
1:  function helpAndGetValueByVersion(valueCell, version)
2:      if valueCell.version = 0 then
3:          CAS(0, valueCell.version, GV)
4:      end if
5:      vv ← valueCell.version
6:      if vv ≤ version then
7:          return valueCell.value
8:      end if
9:      return this.previousVersions.floor(version)
10: end function
```

4.2.12 Rebalancing

FixTagged (Algorithm 14) removes a tagged node from the tree. Initially, it searches for the tagged node, and if it's not found, the function exits, indicating that another thread has removed the node. If fixTagged locates the target node, it attempts to eliminate it by performing a series of steps. Firstly, it creates a copy c of the parent node, merging the key and children of the tagged node into c, and updates the grandparent to point to c. However, if the resulting merged node exceeds the maximum allowed size, fixTagged takes an alternative approach. It generates a new node p with two new children, evenly distributing the contents of the old tagged node and its parent between them. Subsequently, the grandparent is updated to point to p, a tagged node, unless it becomes the new root, in which case it serves as an internal node.

FixUnderfull (Algorithm 15) addresses a node n that falls below the minimum size unless n is the root or entry node. It accomplishes this by evenly distributing keys between n and its sibling s, provided that this action does not result in either of the new nodes becoming underfull (Algorithm 16). Alternatively, if redistribution is not feasible, fixUnderfull merges n with s. In this scenario, the merged node may still be underfull, or the parent node might become underfull if it was already at the minimum size before merging its children. Consequently, fixUnderfull is recursively called on both the merged node and its parent. It is crucial for fixUnderfull that n is underfull, its parent p is not underfull, and none of n, p, and s are tagged, and if n is a leaf node, n's adjacent leaf nodes must be unmarked. If these conditions aren't met, fixUnderfull retries its search (Algorithm 17).

CleanObsoleteKeys (Algorithm 18) removes keys logically deleted from a leaf node, creating space for new keys and reducing the need for rebalancing. Initially, the process involves identifying the smallest version of the current scan, $minScanVersion$, by iterating through the ongoing scan array (OSA) and retrieving the version of each ongoing scan. Subsequently, the function loops over the keys array of the leaf node to identify any deleted keys (where the latest version of the corresponding value is $\perp$). If a deleted key is found, and its latest version is less than or equal to $minScanVersion$, it is physically removed from the node, replacing it with $\perp$.

Algorithm 14 Fix Tagged Node

```
1: function fixTagged(node)
2:     while True do
3:         if node.marked then
4:             return
5:         end if
6:         path ← search(node.searchKey, node)
7:         if path.n ≠ node then
8:             return
9:         end if
10:        Lock path.n, path.p, and path.gp
11:        if path.n, path.p, or path.gp is marked or path.p is TaggedInternal then
12:            Release all locks
13:            Continue
14:        end if
15:        node.marked ← true
16:        path.p.marked ← true
17:        if path.p.size + 1 ≤ MAX_NODE_SIZE then
18:            newNode ← new Internal containing the keys & pointers of node and parent
19:            path.gp.ptrs[path.pIdx] ← newNode
20:            Release all locks
21:        else
22:            newNode ← new subtree of three nodes consisting of a new Internal that points
       to two new internal nodes which evenly share the keys & pointers of node and parent (except
       for the pointer to node)
23:            path.gp.ptrs[path.pIdx] ← newNode
24:            Release all locks
25:            fixTagged(newNode)
26:        end if
27:    end while
28: end function
```

Algorithm 15 Fix Underfull Node

```
1:  function fixUnderfull(node)
2:      if node = entry OR node = entry.ptrs[0] then
3:          return
4:      end if
5:      while true do
6:          path ← search(node.searchKey, node)
7:          if path.n ≠ node then
8:              return
9:          end if
10:         right, left ←⊥
11:         if path.nIdx = 0 then
12:             sIndex ← 1                                          ▷ Sibling is right child
13:             right ← parent.ptrs[sIndex]
14:             left ← path.n
15:         else
16:             sIndex ← path.nIdx - 1
17:             right ← path.n
18:             left ← parent.ptrs[sIndex]
19:         end if
20:         sibling ← parent.ptrs[sIndex]
21:         Lock node, sibling, path.p, path.gp
22:         if node.size ≥ MIN_NODE_SIZE then
23:             return
24:         end if
25:         if parent.size < MIN_NODE_SIZE OR node, sibling, parent, gParent is marked OR node, sibling, parent is TaggedInternal then
26:             Release all locks
27:             Continue
28:         end if
29:         if node is leaf then
30:             if node.left and/or node.right.parent ≠ node.parent then
31:                 Lock and check if marked
32:                 if marked then
33:                     Release all locks
34:                     Continue
35:                 end if
36:             end if
37:         end if
38:         if node.size + sibling.size ≤ 2 × MIN_NODE_SIZE then
39:             newParent ← distributeKeys(node, sibling, parent)
40:             gParent.ptrs[path.pIdx] ← newParent
41:             Mark node, parent, and sibling
42:             Release all locks
43:             return
44:         else
45:             combineKeys(left, right, parent, path.pIdx, gParent)
46:         end if
47:     end while
48: end function
```

Algorithm 16 Distribute keys

```
1: function distributeKeys(node, sibling, parent)
2:     newNode, newSibling ← Distribute keys of node and sibling
3:     newParent ← copy of parent plus pointer to newNode and newSibling, and key between newNode and newSibling
4:     if node and sibling are leafs then
5:         Point newNode and newSibling to each other, using the left and right pointers and connect newNode and newSibling to the leafs linked-list left and right pointers
6:     end if
7:     return newParent
8: end function
```

Algorithm 17 Combine keys

```
1: function combineKeys(left, right, parent, parentIndexInGp, grandParent)
2:     newNode ← Combined keys of right and left nodes
3:     if newNode is leaf then
4:         newNode.left ← left.left
5:         newNode.right ← right.right
6:         right.right.left ← newNode
7:         left.left.right ← newNode
8:     end if
9:     if gParent = entry AND parent.size = 2 then
10:        entry.ptrs[0] ← newNode
11:        Mark left, parent, and right
12:        Release all locks
13:        return
14:    else
15:        newParent ← copy of parent with pointer to newNode instead of left / right
16:        grandParent.ptrs[parentIndexInGp] ← newParent
17:        Mark left, parent, and right
18:        Release all locks
19:        fixUnderfull(newNode)
20:        fixUnderfull(newParent)
21:    end if
22: end function
```

Algorithm 18 Clean Obsolete Keys

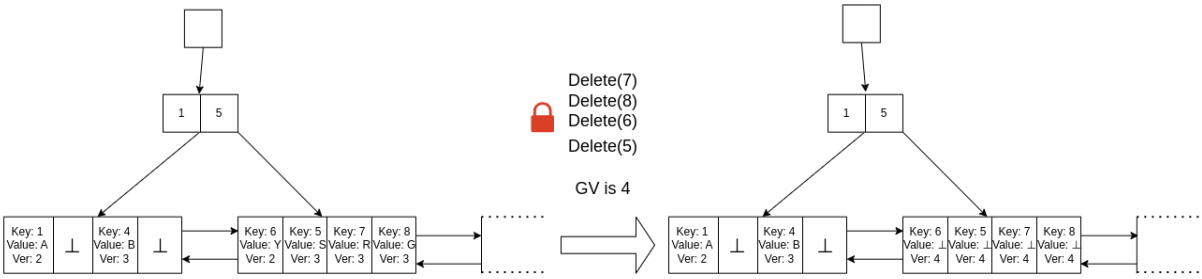
```
1: function cleanObsoleteKeys(node)
2:     minVersion ← ∞
3:     numberOfCleanedKeys ← 0
4:     for i ← 0 to ongoingScansArraySize - 1 do
5:         scanData ← OngoingScansArray[i]
6:         if scanData = ⊥ then
7:             continue
8:         end if
9:         scanDataVersion ← scanData.version
10:        if scanDataVersion = 0 then
11:            newVersion ← GV.getAndIncrement()
12:            if CAS(scanData.version, 0, newVersion) then
13:                scanDataVersion ← newVersion
14:            else
15:                scanDataVersion ← scanData.version
16:            end if
17:        end if
18:        if scanDataVersion < minVersion then
19:            minVersion ← scanDataVersion
20:        end if
21:    end for
22:    for i ← 0 to MAXNODESIZE - 1 do
23:        valueCell ← node.values[i]
24:        if valueCell = ⊥ then
25:            continue
26:        end if
27:        latestValue ← valueCell.helpAndGetValueByVersion(∞)
28:        if latestValue ≠⊥ then
29:            continue
30:        end if
31:
32:        latestVersion ← valueCell.getLatestVersion()
33:        if minVersion ≥ latestVersion then
34:            node.version++
35:            node.keys[i] ← 0
36:            node.values[i] ←⊥
37:            node.size–
38:            node.version++
39:            numberOfCleanedKeys++
40:        end if
41:    end for
42:    return numberOfCleanedKeys
43: end function
```

4.2.13 Helping updates

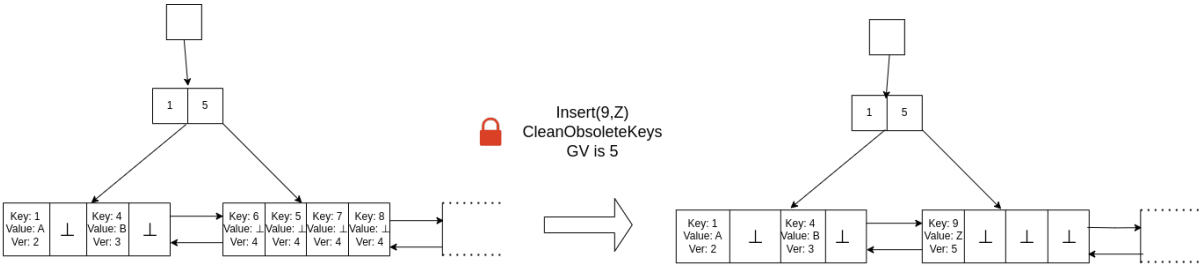
The update operations (insert and delete) rely on the current value of GV (Global Version), whereas a scan operation begins by atomically fetching and incrementing GV. This action ensures that all subsequent updates write versions greater than the fetched one. The scan then utilizes the fetched version, *ver*, as its reference time, guaranteeing that it returns the latest version for each scanned key that does not surpass *ver*. However, a potential race condition might arise if an update operation reads GV equal to *ver* for its data and then pauses momentarily. Simultaneously, a concurrent scan fetches GV, equal to *ver*, as its reference time. The scan may overlook or read the key before it is inserted or logically deleted with the version *ver*. In this situation, the key should be included if inserted or excluded if it is deleted in the scan since its version equals the reference time, but it may not be due to its delayed occurrence. To tackle this issue, scans are designed to help updates by assigning versions to the keys they write. Concerning the update operations, they will write the key in the target node keys array without a version, read GV, and then attempt to set the version to the key's value using CAS. If a scan encounters a key without a version, it will attempt to help the update thread by setting the GV to the key version using CAS.

4.2.14 Helping scans

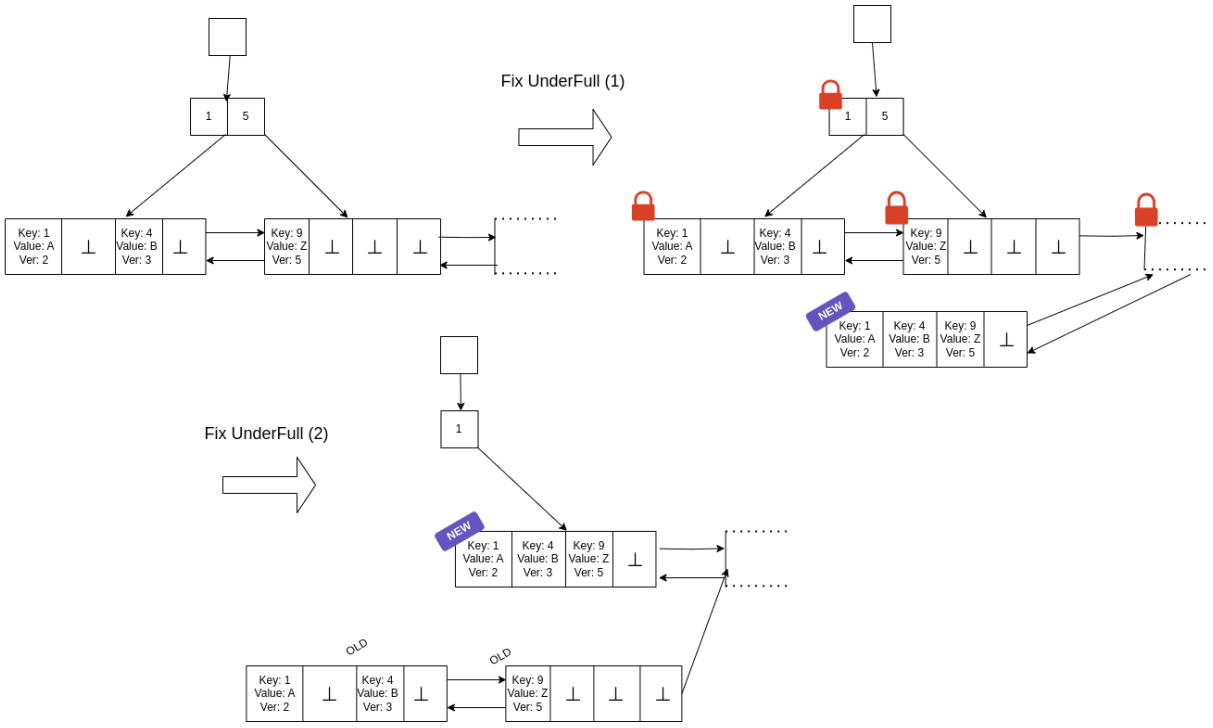
A potential race condition may occur when a scan publishes its version on the OSA, and the cleanObsoleteKeys function requires a scan version for compaction purposes. This situation arises if a scan operation fetches (and increments) GV as its version and then pauses momentarily. Concurrently, the cleanObsoleteKeys function reads all current scan versions from the OSA and may overlook the scan version. Although the scan operation version should be utilized in this scenario, its delayed occurrence could prevent its consideration. Like scan operations helping updates (insert and delete), cleanObsoleteKeys is designed to help scans by assigning versions to them. Concerning the scan operation, it first publishes its data to the OSA without a version, fetches and increments GV, and then attempts to set the version to its published data using CAS. Suppose cleanObsoleteKeys encounters a published scan without a version. In that case, it will try to assist by fetching and incrementing GV and subsequently setting the fetched version to the published scan data using CAS. cleanObsoleteKeys will reread the scan's version for its needs.



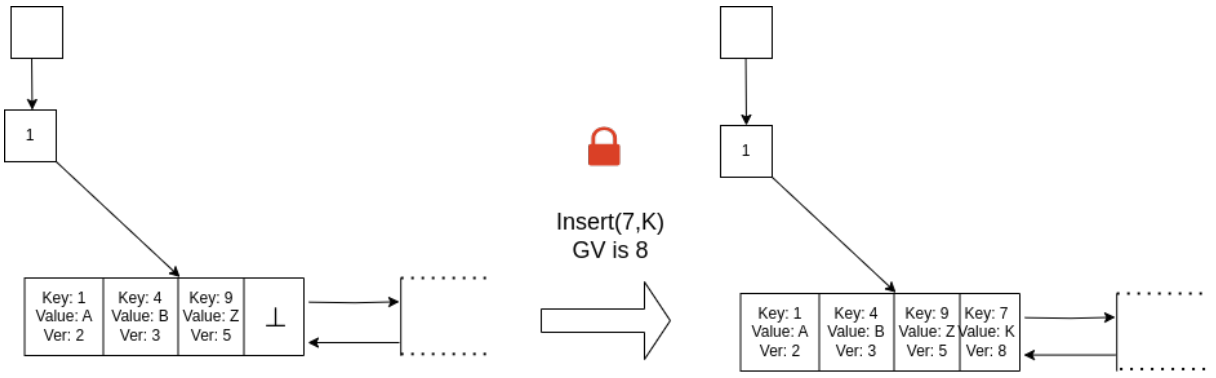
(a) All keys are logically deleted in the right leaf.



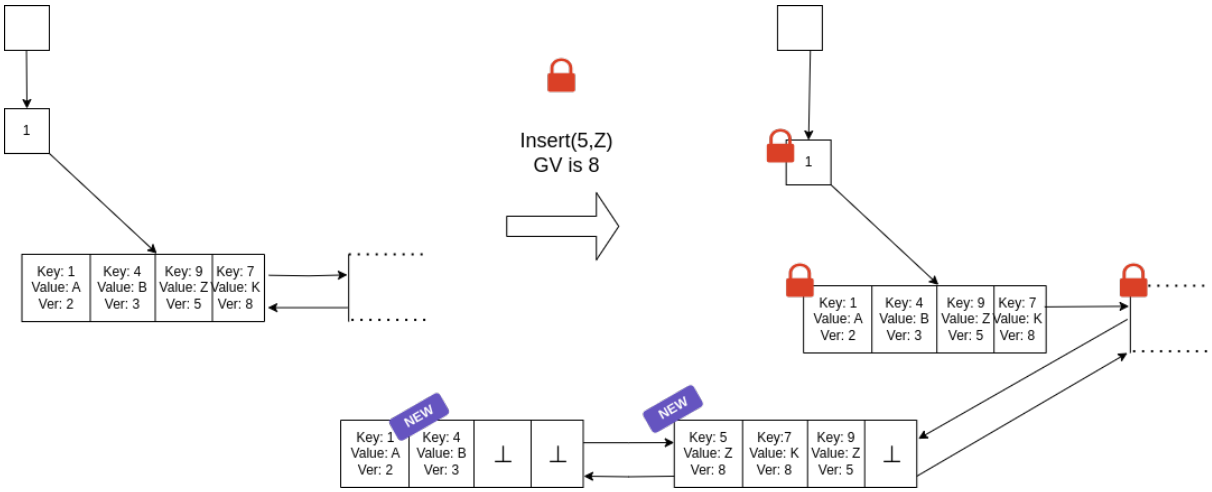
(b) Insert (9, Z) to trigger the cleanObsoleteKeys function, which will physically remove all keys that have been logically deleted. This is because no keys were logically deleted after all ongoing scans were linearized.



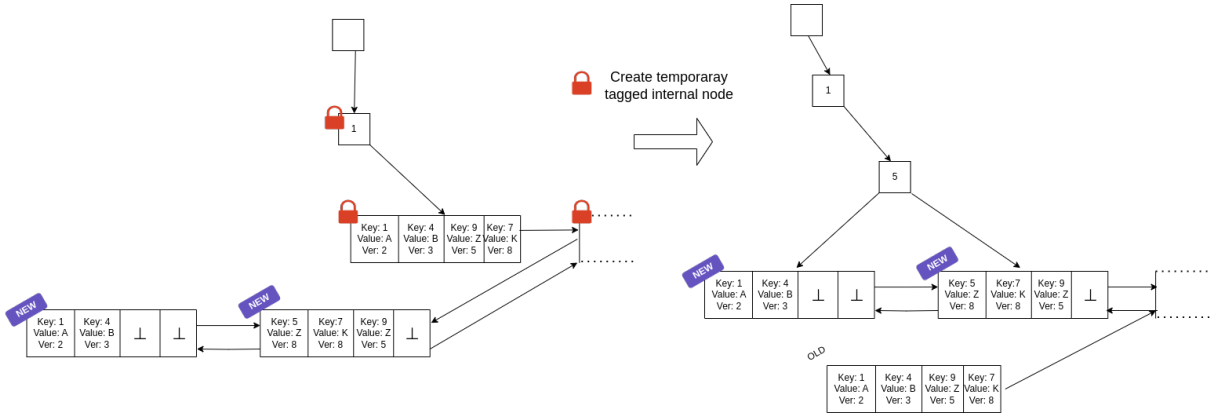
(c) FixUnderfull on an under full leaf node



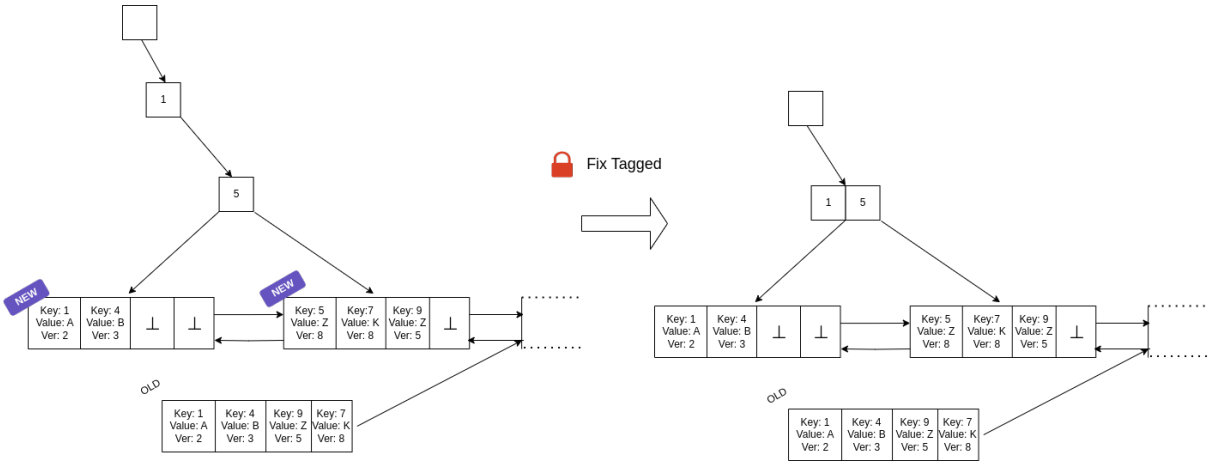
(d) Insert (7, K), causing the leaf node to become full



(e) Insert (5, Z) into the full leaf node, triggering a split insert and creating two leaf nodes.



(f) Create a temporary tagged internal node due to an insertion to a full leaf node.



(g) Invoke FixTagged to address the temporary imbalance by creating a new internal node with keys 1 and 5, and removing the tagged node. In any case, new leaf nodes are linked to the linked list before being linked to the tree, and the old leaf nodes remain linked to the linked list after being unlinked from the tree.

Figure 3: MTASet Operations

4.3 Correctness

This section proves that MTASet is linearizable. To clarify, an algorithm achieves linearizability when, during any concurrent execution, each operation seems to occur atomically at a certain point between its invocation and its response. The linearizability of MTASet involves establishing a connection between the tangible representation of MTASet, the data stored in the system's memory, and its conceptual set form. It involves demonstrating that the operations effectively modify the physical structure of the tree in a manner consistent with the abstract principles outlined at the end of Section 1.

4.3.1 Definitions

Definition 1 (Reachable Node)**.** *A node is considered reachable if it can be accessed by traversing child pointers starting from the entry node.*

Definition 2 (Key in MTASet)**.** *A key k is in the tree if the following conditions are met:*

1. *It is in some reachable leaf l's keys array.*
2. *The version of k's value in l is set.*
3. *The latest value of k in l is not $\perp$.*

Definition 3 (Key range)**.** *The key range of a node is a half-open subset (eg. [1,900)) of the set of all keys that can appear in the subtree rooted at that node.*

Definition 4 (Entry node key range)**.** *The key range of the entry node is the range of all keys present within the tree. Let n be an internal node reachable with a key range of [L, R). If n contains no keys, its child's key range remains as [L, R). However, if n does contain keys k_1 through k_m, then the key range of n's leftmost child (referred to by n.ptrs[0]) is [L, k_1), the key range of n's rightmost child (referred to by n.ptrs[m]) is [k_m, R). For any middle child referred by n.ptrs[i], the key range is [k_i, k_{i+1}). Intuitively, a node's key range represents the collection of keys permitted to exist within the subtree originating from that node.*

Definition 5 (Search Tree)**.** *Let n be an internal node within a tree, and let k be a key within n. A tree is a search tree when the following conditions are met:*

1. *All keys within the subtrees located to the left of k in n are strictly less than k.*

2. *All keys within the subtrees positioned to the right of k in n are either greater than or equal to k.*

4.3.2 Invariants

We establish a set of invariants regarding the tree's structure. These invariants remain valid for the tree's initial state, and any alteration to the tree upholds all these invariants. These established invariants are a foundation for proving the data structure's linearizability.

Theorem 1. *MTASet Invariants: The following invariants are true at every configuration in any execution of MTASet:*

1. *All reachable nodes (Definition 1) form a relaxed (a,b)-tree.*
2. *The range of keys within a reachable node that was removed remains constant.*
3. *An unreached node retains the same keys and values it held when it was last reachable and unlocked, meaning updates do not simultaneously detach and alter a node.*
4. *Each key appears only once in a leaf node among all leaf nodes.*
5. *If a node was once reachable and is presently unmarked, it remains reachable.*
6. *Let $l1$ be a full or underfull leaf node that is part of a merge or split operation, and let $l2$ be a new node created by the split or merge. The leaf node $l1$ can not reach $l2$ using $l1.right$ pointer.*
7. *Let l be a linked node that is about to be unlinked, then $l.right$ and $v.left$ are constant (may never change once it is unlinked)*
8. *In the search operation on a node with key k and target t, the key range of n contains k.*

Intuitively, invariants 1 through 4 stem from the sequential accuracy of the updates, alongside the assurance that any node subject to replacement or modification remains locked and accessible until the update takes effect. The correctness of the updates in a single-threaded execution can be discerned through examination of the pseudocode, thus we refrain from a detailed proof. The concurrent correctness of invariants 1 through

4 is briefly clarified. Invariants 5, 6, and 7 can be straightforwardly deduced from the pseudocode. Invariant 8 differs slightly as it focuses on verifying the correctness of an operation rather than a structural property. Its proof involves some complexity. Therefore, a detailed proof is provided.

Proof. The invariants are true at the initial state of the MTASet.

Lemma 1. *The MTASet constitutes a relaxed (a,b)-tree*

Proof. The updates applied to the tree align with those delineated by Larsen and Fagerberg in [18]. The authors establish that when these updates are executed atomically, the resulting tree always maintains the properties of a relaxed (a,b)-tree. Consequently, the subsequent portion of the proof aims to demonstrate that each update affects the tree atomically. This necessitates demonstrating for each update that:

- The update seemingly occurs at a single step, e.g., it is linearizable.
- The update is correct, e.g., the value is updated in the tree in accordance with the previous operations.

The first condition is straightforward. Simple inserts and deletes occur when the modified leaf is unlocked. For the second condition, assuming the updates are sequentially correct is easily confirmable by comparing the pseudocode in this paper to that in [18]. To establish concurrent correctness from sequential correctness, it suffices to demonstrate that the update operates on the correct data (i.e., the correct node with all preconditions of the sequential code met), the update affects data present in the tree, and the data used for constructing the update remains unchanged during its construction. Update (insert and delete) and Scan(low, high) operations rely on the search function to find a leaf suitable for updating a key or commencing the scan. As per invariant 8, the key range of the leaf includes the key. Invariant 1 guarantees the tree's adherence to a search tree structure. Consequently, a unique reachable leaf's key range includes the key. The rebalancing steps in the sequential code entail certain preconditions. For instance, the fixTagged rebalancing step necessitates that the node is tagged while its parent and grandparent nodes are not. Similarly, fixUnderfull requires that none of the involved nodes are tagged, the parent node is not underfull, and the target node is underfull. Both functions explicitly verify these conditions, ensuring the rebalancing steps operate on the

correct data. Furthermore, each update ensures that all involved nodes are not marked before proceeding. An unmarked node remains in the tree until the update unlinks it, as per invariant 5. Any children of the node are also considered part of the tree, as per Definition 1. Therefore, the data used to construct the update is within the tree. Lastly, the locks obtained by each update guarantee that any data involved in the update remains unchanged until the locks are released. This ensures the consistency of the data throughout the update process. □

Lemma 2. *The range of keys within a previously reachable node remains constant.*

Proof. We need to examine instances where existing nodes are connected to a new parent to ensure that the key range of all descendant nodes remains unchanged. This scenario arises in both fixTagged and fixUnderfull. In either function, the routing keys surrounding any remaining pointer remain consistent before and after the update. Consequently, the key range of the pointed-to-node remains unchanged. This principle also applies to the node's leftmost and rightmost children, as the grandparent's key range remains unaltered (as per this invariant), and the key range of the new parent matches that of the old parent. □

Lemma 3. *An unreached node retains the same keys and values it held when it was last reachable and unlocked, meaning updates do not simultaneously detach and alter a node.*

Proof. Updates that unlink a node follow a specific sequence: they first lock the node, then unlink it, and finally unlock it, all without altering the node's keys or values. □

Lemma 4. *Each key appears only once in a leaf node among all leaf nodes.*

Proof. Insert operations read the entire keys array of a leaf while it is locked before attempting to insert a key. Therefore, the operation ensures that a duplicate key is never inserted. fixUnderfull will not duplicate keys when merging two leaves because there exists a unique leaf whose key range includes a given key, and any keys within that key range are exclusively present in that leaf (as per invariant 1). Consequently, a key can only belong to one of the two leaves and cannot appear twice in the merged node. □

Lemma 5. *In the search(Key key, Node target) operation on a node n, the key range of n contains key.*

Proof. The search operation maintains the invariant that the key range of the node it is currently reading includes the search key. Let's denote this node as n. This invariant is satisfied for the entry node, as its key range spans the entire key space. As the routing keys of an internal node partition its key range, there exists a unique child whose key range includes the search key. Let c be the child followed by the search after reading node n. Even if n is not present in the tree when the search reads the pointer to c, c must have been designated as a child of n while n was part of the tree, as only nodes within the tree are modified (as per invariant 3). Thus, when n was in the tree and had c as its child, the key range of c included the search key (according to Definition 4). Since the key range of a node remains constant (as per invariant 2), it follows that the key range of c still contains the search key. □

□

4.3.3 Linearzability of Find

The linearizability of the find operation can be reasoned as follows. According to invariant 8, the leaf node at which search(key, target) returns was, at some point, the only leaf node that could potentially contain the key. The searchLeaf(key,leaf) operation will return successfully if, during an interval when the leaf was unlocked, it finds the search key and reads its non-$\perp$ latest value or if it fails to find the key by reading a value whose latest version is $\perp$ or by scanning the entire leaf. Invariant 4 guarantees that the key is unique within the leaf. Since the leaf remains unlocked throughout this interval and nodes are not modified while they are unlocked, the result of the find operation accurately reflects the state of the leaf during that time. If the leaf was part of the tree at any point during this interval, then the find operation can be linearized at that point and considered correct.

However, if the leaf was never part of the tree during the unlocked interval, the find operation linearizes just before the leaf was unlinked. This is because, as stated in invariant 3, updates do not alter an unlinked node. Thus, the value returned by find corresponds to what it would be if the find operation had occurred atomically just before the node was unlinked. To establish this, we need to show that the leaf's unlinking point must have occurred concurrently with the find operation. By demonstrating that each node visited during the search was part of the tree at some point, we can infer that if a node n was not in the tree during an unlocked interval, the unlinking of n must have happened

concurrently with the find operation.

Theorem 2. *Each node visited during the search operation was part of the tree at some point during the search.*

Proof. The theorem holds for the root node. If the root is also a leaf, the proof is complete. Otherwise, a child pointer is read from the root during the search. We need to show that any child pointer accessed from a node n, which was part of the tree during the search, points to a child that was also in the tree at some point during the search.

If n is still part of the tree when its child pointer is accessed, then by Definition 2, the child pointed to by n is also within the tree at that time. Hence, the child must have been in the tree during the search.

Alternatively, if n has been unlinked due to an update, the search must have occurred concurrently with the unlinking of n. This is because, by assumption, n was present in the tree during the search but was absent when the search accessed the child pointer. According to invariant 3, the pointers of n still point to its children just before it was unlinked. Therefore, the child followed by the search was also part of the tree at some point during the search.

□

4.3.4 Linearzability of Insert

The linearizability of insert operation in MTASet involves four potential linearization points for an insert(key, val) operation.

An insert operation that successfully locates its target key during the search process follows the exact linearization as a find operation. The return value of the search corresponds to the latest non-$\perp$ value associated with the key (validated by the correctness of find), making it the appropriate value to return for the insert.

Suppose an insert operation finds the key with a non-$\perp$ latest value in leaf l after acquiring l's lock. In that case, it can linearize at any point while holding l's lock. During this lock period, the key cannot be removed from l, its associated value remains unchanged, and l cannot be unlinked (as unlinking l necessitates marking it). As the leaf's version remains even, the associated non-$\perp$ latest value is the correct return value, as per Definition 2.

An insert operation that inserts a key-value pair into a non-full leaf l or modifies a key whose latest value is $\perp$ (logically deleted) in l linearizes at the second that a version is assigned to the new value that was written to the values arrays. Before this linearization point, the key or its corresponding new value was not present in MTASet since the insert operation accessed l while it was locked without finding the key, or finding a key with a $\perp$ associated value. l remains the only reachable leaf that might contain the key. After the linearization point, as defined in Definition 2, the key is part of MTASet because it is added to l, l remains reachable, and the latest value is non-$\perp$.

For splitting inserts, where searches can detect the change as soon as the pointer to the new subtree is updated in the parent node, the linearization must occur at the write to the parent node. Consider a scenario where a splitting insert writes the new pointer into the parent node p at time t. Let l be the leaf that was split and replaced by a tagged node t with children $l1$ and $l2$. Before the write to p, the inserted key is not in MTASet since the insert operation reads l while it is locked and does not locate the key with a non-$\perp$ value (and l is the only reachable leaf that might contain the key). However, after the write to p, the inserted key is in the tree because it resides in either $l1$ or $l2$, both of which are reachable since p is unmarked and thus accessible (invariant 5). Other keys in l are unaffected by splitting inserts as they are assigned to either $l1$ or $l2$ during the splitting operation. In cases where the insert operation succeeds, the returned value $\perp$ is correct, given that the insert operation was successful.

4.3.5 Linearzability of Delete

The linearization of delete and the justification of return values follow a similar rationale to the first three cases of the linearizability of insert

4.3.6 Linearzability of Scan

Scans are linearized when the GV surpasses the version used for collecting values. This typically occurs through Fetch-and-increment and occasionally with the assistance of cleanObsoleteKeys. Any key k inserted or deleted and linearized after scan s has been linearized will not be included in s because the version of k's value will be higher than that of s.

Due to rebalancing, leaf nodes are continuously and concurrently linked and unlinked

to and from the MTASet's underlying tree and the linked list of leaf nodes traversed by the scan. Therefore, we will demonstrate that these modifications do not compromise the accuracy of the values returned by a scan. Let scan(low,high) s be an ongoing scan operation that was linearized at time t, which starts traversing the linked-list of leaf nodes starting from a leaf node l, which is designated to hold to smallest key in the range, namely, low, and according to invariant 1, l is unique. While s traverses the linked list of leaf nodes, the following cases may occur:

The current leaf node was unlinked during a scan visit. According to Invariant 7, the scan can proceed to the next node from an unlinked node. According to Invariant 6, the scan can't reach the respective node(s), resulting from the unlinked node's rebalance, and, therefore, the scan will not scan a key more than once.

The next leaf node that s will visit was under full and has been replaced. A leaf node becomes under full due to keys that were physically removed. Keys are physically removed because they were logically deleted, and all of their previous versions are no longer needed. The scan will visit the respective node(s) resulting from the fixUnderFull procedure, which will miss keys removed before the scan was linearized and wouldn't be collected by the scan anyway.

The next leaf node that s will visit was full and replaced. A new key was attempted to be inserted into a full leaf node, but in this case, the full leaf node content is split into two new nodes, which also include the new key. The scan will not collect the new key since it was inserted after the scan was linearized.

5 Experiments

In this section, we compare MTASet with the OCC-ABTREE [26], OCC-ABTREE* [26] implemented with range scan [2], KiWi [4], and Java's ConcurrentSkipList (non-atomic) [19]

5.1 System and setup

In our experiments, we utilized a virtual machine on Azure (Standard_D96ads_v5) with the following specifications: an AMD EPYC 7763 64-Core Processor with 96 vCPUs and 384 GB of RAM. All data structures were implemented in Java. Both MTASet and the OCCAB-Tree were configured with a=2 and b=256. The machine was running Ubuntu 20.04.2 LTS.

5.2 Methodology

Each experiment begins with a seeding phase, where a random subset of integer keys and values is inserted into the data structure until its size reaches half of the key range. Following this, 80 threads are created and started simultaneously, consisting of k threads designated for scans and $80-k$ threads for other operations, marking the start of the measured phase of the experiment. During this phase, each of the $80-k$ threads repeatedly selects an operation (insert, delete, find) based on the desired frequency of each operation. This phase lasts 10 seconds, recording the total throughput (operations completed per experiment). Threads designated for the scan operation repeatedly perform scans, recording the total number of collected keys. Each experiment is conducted 10 times, and our graphs display the averages of these runs.

5.3 Scan 32k Keys

In the Scan-only experiment, MTASet scans approximately 2.5 times more keys than its update operations-optimized competitor, OCC-ABTREE*. KiWi demonstrates a significant advantage, achieving nearly a 3x improvement over MTASet in scans. This outcome is anticipated because KiWi is specifically optimized for scan operations.

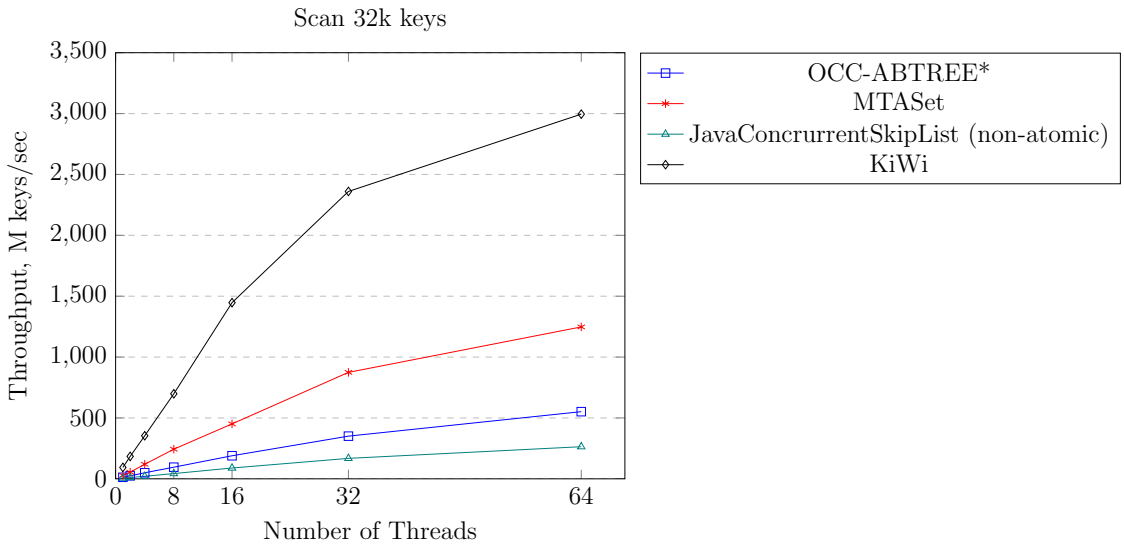


Figure 4: 100% Scan

5.4 Scan 32k Keys, parallel 80% Insert, 20% Delete

Scans with parallel updates: MTASet scans up to three times more keys than its competitor, OCC-ABTREE*, which is optimized for update operations. This difference arises because MTASet scans are wait-free, while OCC-ABTREE* scans are lock-free. Parallel insert operations frequently cause the OCC-ABTREE* scan operation to wait.

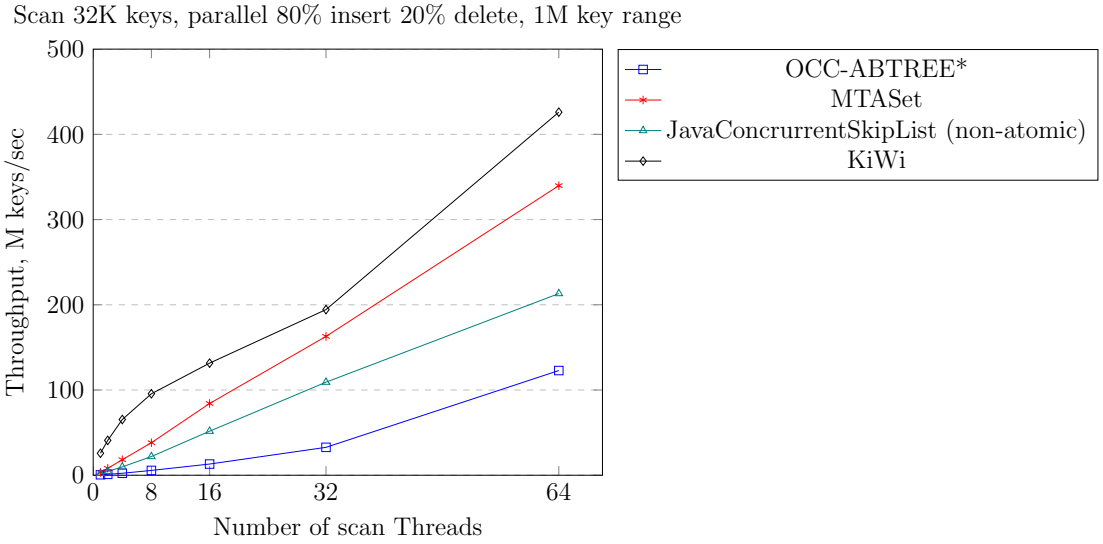


Figure 5: Scan, parallel 80% Insert, 20% Delete

5.5 Get

Reads only: MTASet reads up to approximately 20% more keys than the OCC-ABTREE*.

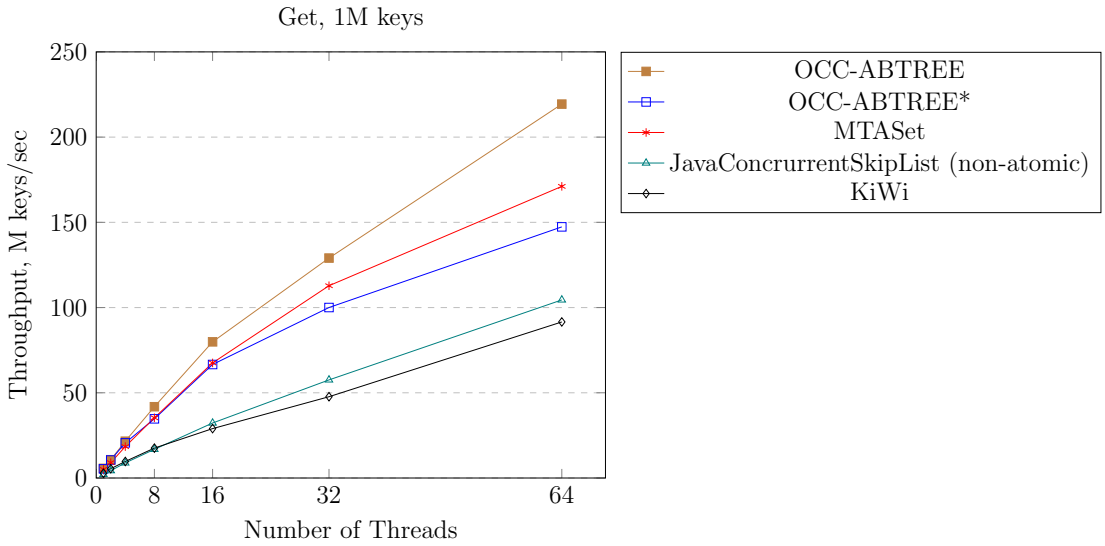


Figure 6: 100% Get

5.6 80% Insert, parallel 20% Delete

In the update-intensive experiment, primarily involving Inserts, MTASet inserts approximately 20% more keys compared to OCC-ABTREE*.

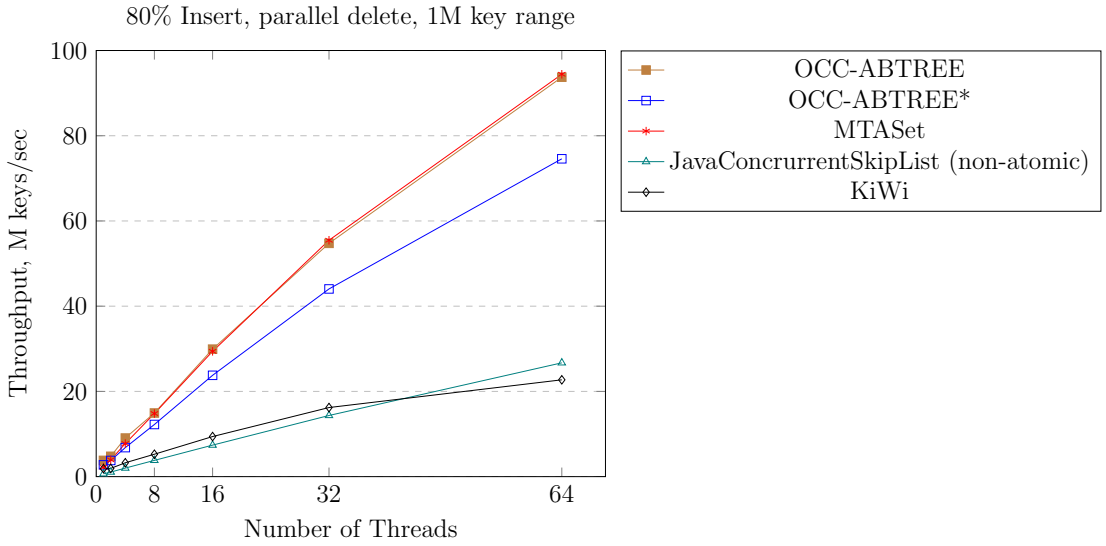


Figure 7: 80% Insert, parallel 20% Delete

5.7 100% Insert

In the update-intensive experiment involving inserts only, OCC-ABTREE* inserts approximately up to 20% more keys than MTASet.

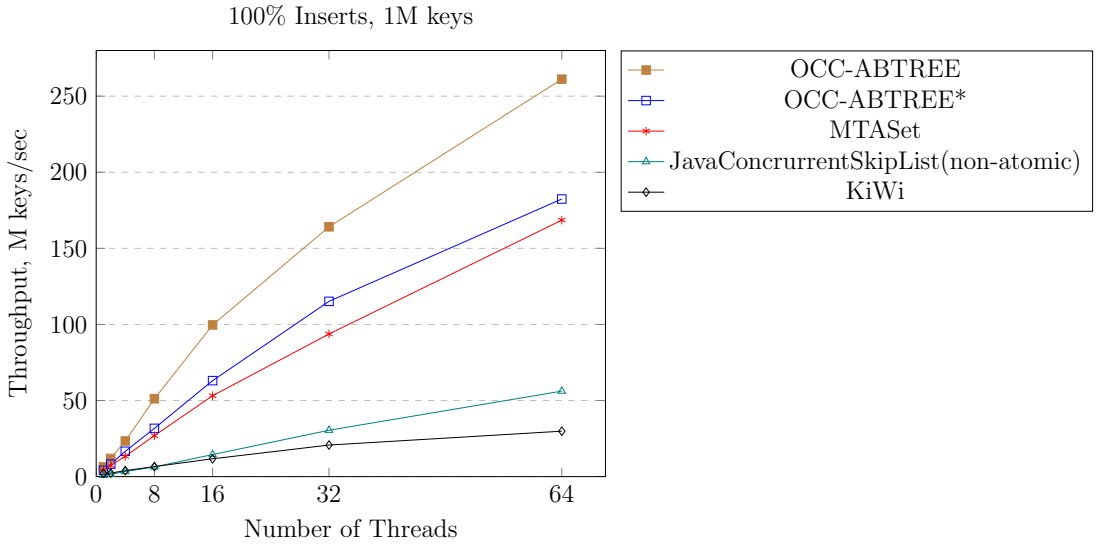


Figure 8: 100% Insert

5.8 90% Get, parallel 9% Insert, 1% Delete

During the 90% get with parallel insert (9%) and delete (1%) experiment, MTASet performs on par with OCC-ABTREE* and far outperforms both KiWi and the non-atomic JavaConcurrentSkipList

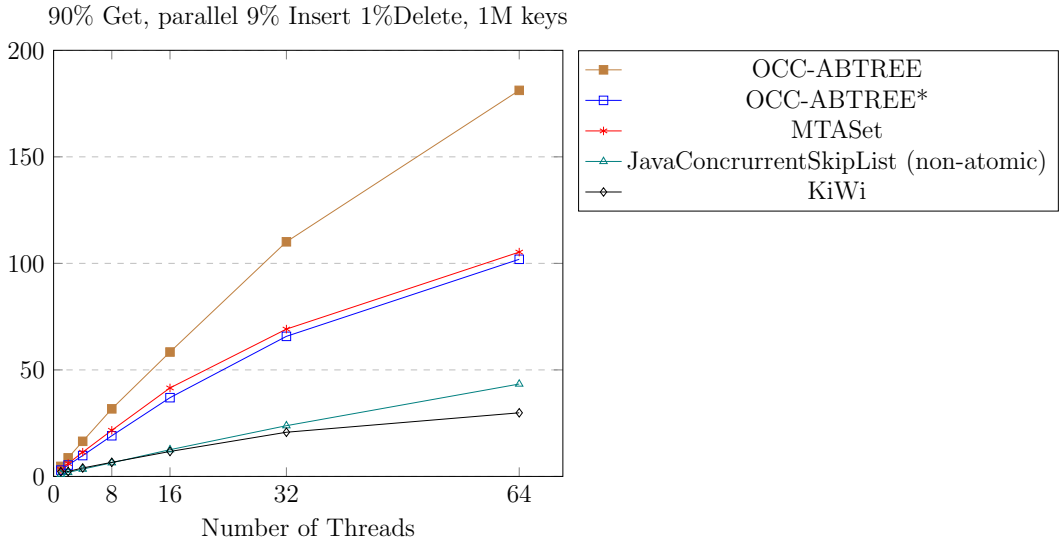


Figure 9: 90% Get, parallel 9% Insert, 1% Delete

6 Future Work and Conclusions

A promising avenue for future work involves implementing elimination [15, p.254] in MTASet and investigating its potential to optimize the range query operation. This enhancement could further elevate MTASet's performance across different workload types. In this study, we introduced MTASet, which has demonstrated strong performance characteristics in both read-mostly and update-heavy environments. Notably, its range query operation significantly outperforms that of OCC-ABTREE.

7 Artifact Description

The source code for all algorithms and experiments conducted in this paper is available on GitHub, https://github.com/danielmanordev/MTASet [20].

To compile and run, use Amazon Correcto 11 SDK.

8 Acknowledgements

I want to express my gratitude to Dr. Moshe Sulamy, my supervisor at the Academic College of Tel-Aviv Yaffo, for his in-depth introduction to "Multi-processor programming" during my MSc studies and for his continuous support and guidance throughout this research.

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *Journal of the ACM (JACM)*, 40(4):873–890, 1993.

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

[3] H. Avni, N. Shavit, and A. Suissa. Leaplist: lessons learned in designing tm-supported range queries. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing*, pages 299–308, 2013.

[4] D. Basin, E. Bortnikov, A. Braginsky, G. Golan-Gueta, E. Hillel, I. Keidar, and M. Sulamy. Kiwi: A key-value map for scalable real-time analytics. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 357–369, 2017.

[5] P. A. Bernstein, V. Hadzilacos, N. Goodman, et al. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley Reading, 1987.

[6] P. E. Black. Dictionary of algorithms and data structures, 1998.

[7] A. Braginsky and E. Petrank. A lock-free b+ tree. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 58–67, 2012.

[8] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. *ACM Sigplan Notices*, 45(5):257–268, 2010.

[9] T. Brown. *Techniques for constructing efficient lock-free data structures*. University of Toronto (Canada), 2017.

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

[11] T. Brown and J. Helga. Non-blocking k-ary search trees. In *Principles of Distributed Systems: 15th International Conference, OPODIS 2011, Toulouse, France, December 13-16, 2011. Proceedings 15*, pages 207–221. Springer, 2011.

[12] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 50–59, 2004.

[13] K. Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.

[14] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems (TOCS)*, 25(2):5–es, 2007.

[15] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear. *The art of multiprocessor programming*. Newnes, 2020.

[16] T. Kobus, M. Kokociński, and P. T. Wojciechowski. Jiffy: A lock-free skip list with batch updates and snapshots. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 400–415, 2022.

[17] S. Lakshman, S. Melkote, J. Liang, and R. Mayuram. Nitro: a fast, scalable in-memory storage engine for nosql global secondary index. *Proceedings of the VLDB Endowment*, 9(13):1413–1424, 2016.

[18] K. S. Larsen and R. Fagerberg. B-trees with relaxed balance. In *Proceedings of 9th International Parallel Processing Symposium*, pages 196–202. IEEE, 1995.

[19] D. Lea. ConcurrentSkipListMap. *java.util.concurrent*, 2006.

[20] D. Manor. MTASet code. `https://github.com/danielmanordev/MTASet`.

[21] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

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

[23] J. Nelson-Slivon, A. Hassan, and R. Palmieri. Bundling linked data structures for linearizable range queries. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 368–384, 2022.

[24] N. Shafiei. Non-blocking patricia tries with replace operations. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*, pages 216–225. IEEE, 2013.

[25] B. Sowell, W. Golab, and M. A. Shah. Minuet: A scalable distributed multiversion b-tree. *arXiv preprint arXiv:1205.6699*, 2012.

[26] A. Srivastava and T. Brown. Elimination (a, b)-trees with fast, durable updates. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 416–430, 2022.

תקציר

מגוון רחב של מימושים של מבני נתונים מסוג "קבוצה" התומכים בגישה מקבילית של חוטים, מותאמים וממוקדים בעיקר לפעולות קריאה, אך בדרך כלל, יעילותם יורדת ככל שעומסים המשתמשים בפעולות כתיבה עולה. בנוסף, מימושים של מבני נתונים מסוג זה, אשר מותאמים לפעולות כתיבה אינטנסביות, אינם יעילים בפעולות של שאילתות טווח. בהמשך לשאלת המחקר "כיצד ניתן לשפר את יעילותם של מבני נתונים מסוג "קבוצה" התומכים בגישה מקבילית של חוטים ותפוקתם גבוהה בפעולות כתיבה אינטנסיביות, בפעולות של שאילותות טווח, תוך כדי הקרבה מינימלית של יעילותם בעומסים המתמקדים בפעולות של כתיבה?", נציג את מבנה הנתונים MTASet, המבוסס על מימוש של עץ (a,b), ויעיל בפעולות קריאה, כתיבה ושאילתות טווח. MTASet מפגין יעילות פעולות בשאילתות טווח בעד פי 2 בהשוואה למבני נתונים עדכניים מסוג זה, תוך כדי שמירה על לינארזביליות.

המכללה האקדמית של תל- אביב-יפו

ביה"ס למדעי המחשב

חיבור זה מהווה חלק מהדרישות לקבלת תואר מוסמך במדעי המחשב
M.Sc

מאת: דניאל מנור

מנחה: ד״ר משה סולאמי

אישור יו"ר ועדת תזה:	תאריך:
חתימת המחב/רת:	תאריך:
אישור המנחה:	תאריך:
סיון תשפ"ד	יולי 2024

המכללה האקדמית של תל אביב-יפו