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

Daniel Manor
The Academic College of Tel Aviv-Yaffo

Mor Perry
The Academic College of Tel Aviv-Yaffo

Moshe Sulamy
The Academic College of Tel Aviv-Yaffo

***Abstract*—In concurrent data structures, the efficiency of set operations can vary significantly depending on the workload characteristics. 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.**

***Index Terms*—Concurrent data structure, key-value map, set, dictionary, a-b tree, update-heavy, range query**

I. 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 [1]–[8] and sets [9]–[11]. These innovations consistently enhance performance over existing solutions and introduce features like atomic range scan operations (i.e., range queries).

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. [8], SnapTree [3] 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 in concurrent data structures is inherently challenging due to the extensive coordination needed across elements. Unlike single-key operations, atomic range queries must traverse multiple nodes, risking inconsistent results if other threads update the data during the query, thus adding further complexity to manage manage concurrent updates. While locking or snapshotting methods can help maintain consistency, they impose synchronization overhead and lead to contention, especially in update-heavy environments. These issues make it difficult to efficiently support range queries at scale without compromising performance or data accuracy.

In response, we introduce MTASet, a concurrent set and dictionary 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 [12] 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 KiWi [2], MTASet's range query 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 [13], specifically the OCC-ABtree [14]. 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.

A. Background

A *set* data structure stores unique elements without any particular order. It supports three operations: *Insert* adds an element to the set if it is not already in the set, *delete* removes an element from the set if it exists, and *contains* tests whether an element is in the set.

An (a, b)-*tree* [15] is a balanced leaf-oriented search 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.

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 a *concurrent* (a, b)-tree data structure, based on OCC-ABtree [14], which includes optimizations tailored for update-heavy workloads. OCC-ABtree does not include a range query operation. However, it was noted in [14] that a range query capability could be implemented for the OCC-ABtree using the following technique detailed in [16]:

In 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. During deletion TS is read and written to the *deletionTime* field of the deleted key, and stored in the thread that executes the deletion in a 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's range query introduces significant improvements, detailed in Section IV, that enhance the range query operation throughput beyond this technique, while maintaining the performance advantages of the OCC-ABtree on update-heavy workloads.

Our correctness notion is *linearizability*, which intuitively means that the object "appears to be" executing sequentially. It is defined for a *history*, which is a sequence of operation invoke and return steps, possibly by multiple threads. A history partially orders operations: operation $op1$ *precedes* operation $op2$ in a history if $op1$'s return precedes $op2$'s invoke; two operations that do not precede each other are *concurrent*. An object is specified using a *sequential specification*, which is the set of its allowed sequential histories. Roughly speaking, a history σ is *linearizable* [17] if it has a sequential permutation that preserves σ's precedence relation and satisfies the object's sequential specification.

B. Contributions

The primary contribution of this paper is the development and analysis of MTASet, a concurrent set specifically optimized for high update throughput and frequent range queries. MTASet enhances performance by leveraging a tailored multiversioning approach, which maintains only the necessary versions of keys for ongoing scans, thus minimizing overhead and optimizing range query efficiency. Notably, MTASet supports wait-free atomic range queries, ensuring that range query operations complete in a bounded number of steps regardless of the workload. By adapting concepts from OCC-ABtree and incorporating other techniques to manage versioning effectively, MTASet significantly improves atomic range queries while sustaining high performance in update-heavy workloads. Experimental results demonstrate that MTASet surpasses many existing concurrent data structures in both read-mostly and update-intensive scenarios. This work addresses a critical gap in concurrent data structures, and can also be used as a robust framework for balancing update and range query operations, making MTASet a valuable addition to the toolkit of multicore developers.

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 all keys within the range [fromKey, toKey].

Evaluation results: The MTASet Java implementation can be found on GitHub [18]. In Section IV, we benchmark its performance under various workloads. In most experiments, it significantly surpasses OCC-ABtree*, a variant of OCC-ABtree [14] tailored for update-heavy workloads with atomic range query capabilities [16]. 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 [2] 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 updates, MTASet exceeded the performance of the OCC-ABtree* [14], [16] 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 [20], and MTASet's updates are up to 3.6 times faster than those of the Java Skiplist.

C. Related Work

Various data structure designs and techniques have been developed to optimize performance in concurrent environments, focusing on skip lists, trees, and range query methods.

In the category of skip lists, KiWi [2] is a Key-Value Map that supports linearizable, wait-free range scans via a multi-versioned architecture similar to MTASet, and its operations utilize the Compare-and-swap (CAS) atomic instruction for lock-free functionality. While KiWi achieves high throughput in range scans, it is not optimized for updates as MTASet is. LeapList [1] also supports linearizable range scans, employing fine-grained locks for concurrency control, similar to MTASet. Another related structure, Jiffy [8], is a linked-list data structure that offers arbitrary snapshots and atomic batch updates. Nitro [21] leverages multiversioning to create snapshots, though these snapshots are not thread-safe during concurrent insert/remove operations.

In the category of tree-based structures, OCC-ABtree [14] is a concurrent (a,b)-tree tailored for update-heavy workloads but lacks native range scan support. However, a general method for range queries is proposed [16], demonstrating lower throughput than MTASet. SnapTree [3] is a lock-based, relaxed-balance AVL tree that provides atomic snapshots and range scans through a linearizable clone operation. Minuet [22] is a distributed, in-memory B-tree that enables linearizable snapshots using a costly copy-on-write approach, which allows snapshot sharing across multiple range scans. BCCO10 [3] introduces a Binary Search Tree with optimistic concurrency control, similar to MTASet, utilizing version-based validation for efficient search operations. Additionally, LF-ABtree [23] is a lock-free (a,b)-tree structure similar to the relaxed (a,b)-tree [13] employed in MTASet.

For range query techniques, Arbel-Raviv and Brown [16] discuss implementing range queries in concurrent set data structures using epoch-based memory reclamation, proposing a traversal algorithm to ensure that all items within a range are accessed during the traversal's lifetime. Nelson et al. [24] present a technique for achieving linearizable range queries on lock-based linked data structures.

II. MTASet Algorithm

In this section, we discuss the MTASet algorithm, exploring its core data structures, node types, and coordination mechanisms that support concurrent operations. We examine the specific roles of different node types, including leaf, internal, and tagged nodes, and the functionality of the linked-list structure of the leaf nodes. Furthermore, we discuss MTASet's operations, highlighting mechanisms such as versioning, locking, and the ongoing scans array (OSA).

MTASet contains a permanent entry pointer to a sentinel node, a reliable starting point for all operations. This guarantees that every thread starts traversal from a well-defined, stable location. The sentinel node contains no keys and a single child, the root node.

A node is *underfull* if it contains fewer keys than the minimum a, and it is *full* when its number of keys equals the maximum b.

An example of the MTASet tree is shown in Fig. 1.

A. Data structures

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

Each node contains a *lock* field, using MCS locks [25] where threads awaiting the lock spin on a local bit, efficiently scaling across multiple NUMA nodes. A thread modifies a node only if it holds the corresponding lock. Leaf nodes include a *version* field which tracks the number of modifications made to the leaf and indicates whether it is currently changing. Upon acquiring a lock, a thread increments the version before initiating modifications, and increments it again once it completed its changes before releasing the lock. Thus, the version is even when the leaf is not being modified and odd when it is. Searches utilize the version 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, allowing updates to determine whether a node is present in the tree. Once marked, nodes are never unmarked.

MTASet's operations use a helper *search* operation which returns a *PathInfo* structure. It 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.

1) Leaf Nodes: Leaf nodes consist of arrays for keys and values. A keys entry is considered empty when represented as $\perp$ and does not have a corresponding value, as shown in Fig. 1. The keys array is unordered, allowing for empty slots, supporting faster updates by eliminating the need to rearrange keys during insertions and deletions. The latest value and version of each key are kept in the corresponding cell in the values array, while older values are organized in a binary search tree.

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-$\perp$ value.

Neighboring leaf nodes are linked through left and right pointers. This setup forms a linked list of leaf nodes with the property: for each leaf l, 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. This list aims to facilitate scan operations, enabling it to traverse leaf nodes directly without traversing the entire tree, as these are the only ones containing values.

2) Internal Nodes: An *internal* node is a non-leaf node that serves as a routing point to direct searches through the structure towards the appropriate leaf nodes where data is stored. 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.

A *tagged* internal node is a non-leaf node that represents a 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.

3) Coordination data structure: To coordinate scan and rebalancing operations, MTASet utilizes a Global Version field (GV) and ongoing scans array (OSA), which keeps track of the versions of ongoing scans and is used by rebalancing for compaction purposes. The OSA and GV are updated by the scan operation, and emplyed by other operations and internal functions. Their usage is described in the description of each operation in Section II-B. The full descriptions and algorithms of all operations are provided in the full paper [26] and the source code [18].

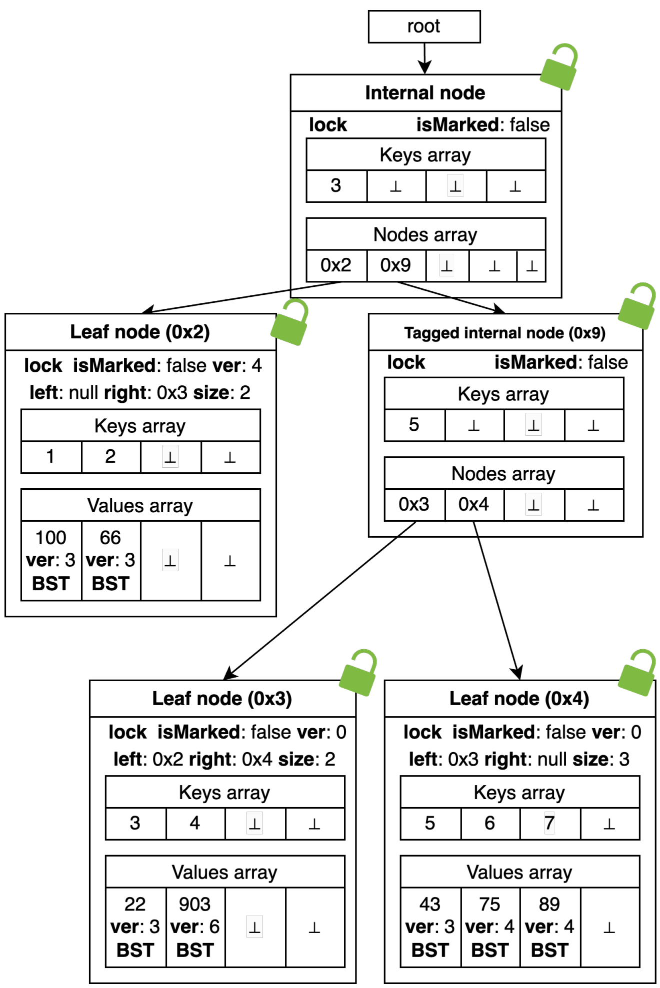

Fig. 1. A snapshot of MTASet: An internal node pointing to a tagged internal node and a leaf node. The tagged internal node points to two leaf nodes. The locks are MCS, no locks are acquired

B. Operations

Each operation invokes the Search function, which takes a key k and traverses the tree from the root to locate the leaf node where k resides. This function is identical to the one used in OCC-ABtree [14]. The function searchLeaf 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 classic double-collect snapshot algorithm [27], 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 designed to run lock-free. This enhances concurrency by allowing updates to internal nodes to occur simultaneously with searches, boosting performance in environments with frequent reads and writes.

The find(k) operation is used by MTASet to locate the relevant leaf node and retrieve the value associated with k in the tree. It simply calls the search and searchLeaf functions and returns the corresponding value.

1) Insert and Delete: During the insert(key, value) operation, a thread starts by executing a search(key, target) and searchLeaf(key,leaf) 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, described in the full paper [26]). If obsolete keys are removed, the new key is inserted, and the fixUnderfull function is called to ensure the node meets the minimum size requirement. If the node's size falls below this minimum, it will either merge the underfilled node with a sibling or redistribute keys between them (using the fixUnderfull helper function).

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 [14] to eliminate the tagged node from the tree.

Deleting a key involves writing (key, $\perp$) by calling the Insert function. If a key is not found or has already been logically deleted, $\perp$ is returned. If 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.

2) Scan: In the scan(lowKey, highKey) operation, 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 value read from GV using CAS. Upon invoking the search operation, the thread identifies the node intended to contain lowKey. From this node, using the scanLeaf function, it traverses the leaf nodes, reading the values corresponding to keys within the [lowKey, highKey] range. These values meet the criteria of having a 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.

3) Helping updates: The update operations (insert and delete) rely on the current value of GV, 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) Helping scans: The cleanObsoleteKeys function, discussed in detail in the full paper [26], is responsible for managing obsolete keys to support efficient memory use and consistency during compaction processes.

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.

III. 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 I.

A. 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 all 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 (e.g., [1,900)) of the set of all keys that can appear in the subtree rooted at that node.*

Definition 4 (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 to the left of k in n are strictly less than k.*
2) *All keys within the subtrees to the right of k in n are either greater than or equal to k.*

1) 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 of 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 key range (Definition 3) of a reachable node that was removed remains constant.*
3) *An unreachable node (which does not satisfy Definition 1) 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 $l.left$ 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. A brief clarification of invariants 1 through 4 regarding concurrent correctness can be found in [14]. 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. A detailed proof can be found in [14].

2) Linearizability of Find: The linearizability of the find operation is established by ensuring that the result accurately reflects the tree's state at a specific moment during the search interval. During a find operation, if the target key k is found, or if its absence is confirmed by either reading a placeholder value $\perp$ or by scanning the entire key array within a leaf node l that was unlocked during the search interval, then the result represents an accurate and stable view of l's state at that moment.

If l was part of the tree at any point during the unlocked interval of the search, then the result of the find operation can be linearized to any instant within this interval when l was verified as part of the tree. However, if l was not part of the tree during the unlocked interval, then its absence implies that it was unlinked from the tree concurrently with the search. In this case, the find operation is linearized to the instant just before l was unlinked, ensuring that each node visited in the traversal was part of the tree at some point during the operation.

Therefore, whether the leaf l was present or unlinked, the result of find accurately reflects the tree's state in real time at one distinct instant during the search interval, meeting the requirements for linearizability.

3) Linearizability of Insert and Delete: The linearizability of the insert operation in MTASet involves four potential linearization points. First, if the insert operation i finds its target key k during the search, it linearizes similarly to the find operation, returning the value corresponding to the located key. Second, if the insert operation i finds the target key k after acquiring the lock on leaf l, it can linearize at any point while holding l's lock. During this time, the key k and value cannot change, and the leaf l cannot be unlinked, ensuring the correctness of the returned value. Third, when i inserts into a non-full leaf l or modifies a logically deleted key, the operation linearizes at the moment the key k and value are written to their respective arrays with a version. Before this point, the key k did not exist in MTASet, afterward, it is part of MTASet with a non-$\perp$ value. For splitting inserts, linearization occurs when the new subtree pointer is written to the parent node. Before this, the key k is not in MTASet, afterward, it is in one of the newly created leaf nodes, with other keys correctly assigned to new nodes.

For a full explanation, refer to the full paper [26]. The linearization of the delete operation and the justification of return values follow a similar rationale as the first three cases of insert linearizability.

B. Linearizability of Scan

The linearizability of a scan operation s is established by the moment when the global version GV surpasses the version v_s used by s to collect values, ensuring that any key inserted or deleted after this point is excluded from s's result. Despite ongoing rebalancing operations that dynamically link and unlink leaf nodes, these changes do not compromise the correctness of s.

Specifically, when s, linearized at time t, initiates from the designated leaf node containing the smallest key in its specified range, it maintains correctness through several cases: if s encounters a node n that is concurrently unlinked, it proceeds directly to the next node, avoiding revisits and thereby preserving a consistent snapshot. In cases where s visits new nodes that replaced underfull nodes, any missing keys in n were removed prior to time t and thus are not included in s. Similarly, if s encounters nodes created by a split of a full node, the key that triggered the split was inserted after t and therefore will not be collected by s.

For a full explanation, refer to the full paper [26].

IV. Evaluation

In this section, we compare MTASet with the OCC-ABtree* [14] implemented with range scan [16], OCC-ABtree without a range query implementation, KiWi [2], and Java's ConcurrentSkipList (non-atomic) [19].

A. 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 OCC-ABtree were configured with a=2 and b=256. The machine was running Ubuntu 20.04.2 LTS.

B. Methodology

Each experiment begins with a seeding phase, where a random subset of integer keys and values are 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.

C. Experiments

We present six experiments (a)-(f), with results shown in Fig. 2. In the 100% scan experiment (a), all threads perform only range queries. KiWi achieves the highest throughput, which is expected as it is optimized for range scans, whereas MTASet is designed for intensive updates. However, the gap narrows in the scan-with-parallel-updates experiment (b), where threads perform range queries in parallel with updates. Here, MTASet's scan throughput significantly surpasses OCC-ABtree* by nearly fivefold, highlighting the impact of this comparison.

In the get experiment (c), where all threads only perform find operations, OCC-ABtree takes the lead, and MTASet performs slightly better than OCC-ABtree*. In experiment (d), with 80% inserts alongside deletes, MTASet and OCC-ABtree (without range query support) achieve the highest throughput, with MTASet's throughput nearly four times that of KiWi. In the 100% inserts experiment (e), OCC-ABtree leads, showing the best performance, with OCC-ABtree* slightly outperforming MTASet.

In the 90% get, 9% insert, and 1% delete experiment (f), where threads perform 90% Get in parallel with 9% Insert and 1% Delete, MTASet performs comparably to OCC-ABtree* and far exceeds both KiWi and the non-atomic JavaConcurrentSkipList.

Overall, MTASet significantly outperforms OCC-ABtree* in range scan workloads while maintaining competitive update performance. This balance makes MTASet a strong candidate for workloads that require efficient range queries without sacrificing significant update efficiency.

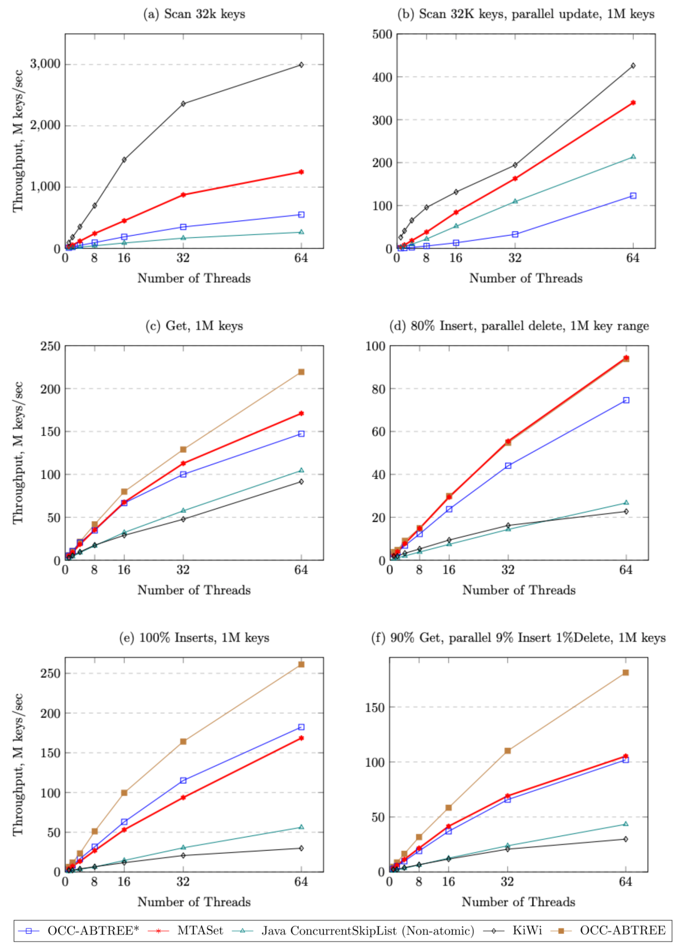

Fig. 2. Experiments (a) and (b) measure Scan throughput, (c) and (f) measure Get throughput, and (d) and (e) measure Insert throughput. (*) OCC-ABtree implemented with scan support.

V. Discussion

In this study, we introduced MTASet, a concurrent set data structure designed to excel in both high update throughput and efficient, wait-free atomic range queries. MTASet combines a multi-versioning approach to optimize range query performance while preserving exceptional efficiency in environments with intensive updates. Notably, MTASet demonstrates a significant advantage over the OCC-ABtree* in range query operations, while maintaining competitive performance in update-heavy workloads. The results from our experiments show that MTASet strikes a remarkable balance between these two operations, making it a versatile solution for modern applications requiring both fast updates and efficient range scans.

Looking ahead, an exciting avenue for future work is the potential integration of elimination [28]–[30] into MTASet. This enhancement could further improve its range query performance, offering additional optimizations for diverse workload types and solidifying MTASet as a robust, balanced data structure for concurrent applications.

References

[1] 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*, 2013, pp. 299–308.

[2] 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*, 2017, pp. 357–369.

[3] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun, "A practical concurrent binary search tree," *ACM Sigplan Notices*, vol. 45, no. 5, pp. 257–268, 2010.

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

[5] 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*. Springer, 2011, pp. 207–221.

[6] 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*, 2004, pp. 50–59.

[7] K. Fraser, "Practical lock-freedom," University of Cambridge, Computer Laboratory, Tech. Rep., 2004.

[8] 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*, 2022, pp. 400–415.

[9] 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*, 2012, pp. 58–67.

[10] 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*, 2014, pp. 317–328.

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

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

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

[14] 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*, 2022, pp. 416–430.

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

[16] M. Arbel-Raviv and T. Brown, "Harnessing epoch-based reclamation for efficient range queries," *ACM SIGPLAN Notices*, vol. 53, no. 1, pp. 14–27, 2018.

[17] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, p. 463–492, Jul. 1990. [Online]. Available: https://doi.org/10.1145/78969.78972

[18] D. Manor, "MTASet code," https://github.com/danielmanordev/MTASet.

[19] D. Lea, "Java's ConcurrentSkipListMap," https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentSkipListMap.html.

[20] K. Fraser and T. Harris, "Concurrent programming without locks," *ACM Transactions on Computer Systems (TOCS)*, vol. 25, no. 2, pp. 5–es, 2007.

[21] 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*, vol. 9, no. 13, pp. 1413–1424, 2016.

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

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

[24] 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*, 2022, pp. 368–384.

[25] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems (TOCS)*, vol. 9, no. 1, pp. 21–65, 1991.

[26] D. Manor, M. Perry, and M. Sulamy, "MTASet: A tree-based set for efficient range queries in update-heavy workloads," 2025. [Online]. Available: https://arxiv.org/abs/2507.20041

[27] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "Atomic snapshots of shared memory," *Journal of the ACM (JACM)*, vol. 40, no. 4, pp. 873–890, 1993.

[28] D. Hendler, N. Shavit, and L. Yerushalmi, "A scalable lock-free stack algorithm," in *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 206–215. [Online]. Available: https://doi.org/10.1145/1007912.1007944

[29] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit, "Using elimination to implement scalable and lock-free fifo queues," in *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 253–262. [Online]. Available: https://doi.org/10.1145/1073970.1074013

[30] W. N. Scherer, D. Lea, and M. L. Scott, "Scalable synchronous queues," *Commun. ACM*, vol. 52, no. 5, p. 100–111, may 2009. [Online]. Available: https://doi.org/10.1145/1506409.1506431