Practical Concurrent Traversals in Search Trees

Dana Drachsler-Cohen*
ETH Zurich, Switzerland
dana.drachsler@inf.ethz.ch

Martin Vechev
ETH Zurich, Switzerland
martin.vechev@inf.ethz.ch

Eran Yahav
Technion, Israel
yahave@cs.technion.ac.il

Abstract

Operations of concurrent objects often employ optimistic concurrency-control schemes that consist of a traversal followed by a validation step. The validation checks if concurrent mutations interfered with the traversal to determine if the operation should proceed or restart. A fundamental challenge is to discover a *necessary* and sufficient validation check that has to be performed to guarantee correctness.

In this paper, we show a necessary and sufficient condition for validating traversals in search trees. The condition relies on a new concept of *succinct path snapshots*, which are derived from and embedded in the structure of the tree. We leverage the condition to design a general lock-free membership test suitable for any search tree. We then show how to integrate the validation condition in update operations of (non-rebalancing) binary search trees, internal and external, and AVL trees. We experimentally show that our new algorithms outperform existing ones.

CCS Concepts • **Computing methodologies → Concurrent algorithms**;

Keywords Concurrency, Search Trees

ACM Reference Format:
Dana Drachsler-Cohen, Martin Vechev, and Eran Yahav. 2018. Practical Concurrent Traversals in Search Trees. In *Proceedings of PPoPP '18: 23nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Vienna, Austria, February 24–28, 2018 (PPoPP '18),* 12 pages.
https://doi.org/10.1145/3178487.3178503

1 Introduction

Concurrent data structures are critical components of many systems [26]. However, implementing them correctly and efficiently remains a difficult task [11]. This task is particularly challenging for search trees whose traversals may be performed concurrently with modifications that relocate nodes. Thus, a major challenge in designing a concurrent traversal operation is to ensure that target nodes are not missed, even if they are relocated. This challenge is amplified when a concurrent data structure is optimized for read operations and traversals are expected to complete without costly synchronization primitives (e.g., locks, CAS). Recent work offers various concurrent data structures optimized for lightweight traversals [1, 2, 4, 6–10, 12–14, 18, 19, 22, 25, 27]. However, they are mostly specialized solutions, supporting standard operations, that cannot be applied directly to other structures or easily extended with customized operations (e.g., rotations). Thus, when customized tree operations are needed, programmers resort to general locking protocols. Unfortunately, these protocols yield non-scalable solutions.

We present a necessary and sufficient condition for **pa**th traversal **v**alidation for search **t**rees (PaVT). PaVT is applicable to any search tree, including internal (all nodes have payloads) and external trees (only leaves have payloads). Informally, the PaVT condition states that to determine whether a certain key is in the tree, it suffices to observe only a *limited number of nodes* that are the *succinct path snapshot* (SPS) of the path leading to this key. Moreover, concurrent updates may modify this path and the traversal may still complete successfully if it has found the correct snapshot (Sec. 4).

We leverage PaVT to design a lock-free membership test (Sec. 5). To validate the PaVT condition efficiently, we extend nodes with their SPS, which precisely characterizes the path by which they are reached from the root of the tree. The existence of the SPS allows us to validate a lock-free traversal (e.g., a lookup operation) – and to linearize that traversal with respect to concurrent mutations – by verifying, locally, that the SPS matches the path just taken by the traversal.

We then show how to use PaVT in updates (Sec. 6). An update starts with a traversal and before validating, it blocks concurrent updates to the snapshot (with a local lock) to enable updates to proceed safely, if validation succeeds. We show that this approach provides simple algorithms for insertion and removal in binary search trees (BSTs).

Finally, we empirically evaluate the effectiveness of PaVT (Sec. 7). We observe that SPSs impose only modest space overhead and that maintaining them during tree mutations imposes only modest time overhead. We also experimentally show that the PaVT-ed BST and PaVT-ed AVL tree outperform existing solutions.

*Work done while at Technion.

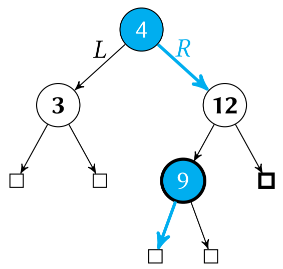
PPoPP '18, February 24–28, 2018, Vienna, Austria
© 2018 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-4982-6/18/02…$15.00
https://doi.org/10.1145/3178487.3178503

2 Preliminaries and Challenges

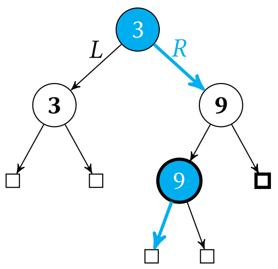
The PaVT condition guides how to efficiently traverse in *search trees*. Search trees consist of nodes. Each node n has one or more keys, denoted by $n.k_1, n.k_2, \ldots$ (or simply $n.k$ if there is one key) and m fields $f_1, \ldots, f_m$ pointing to other nodes. We assume $m \geq 2$. In addition to the m fields, a node has access to its parent via $n.P$. Membership tests in trees look for a key k and return a boolean indicating whether k is in the tree. They start with a traversal that begins at the `root` and proceeds to other nodes by repeatedly checking conditions corresponding to the fields. The traversal terminates when reaching $\perp$ (i.e., `null`) or when no condition is met. At this point, the membership test determines (tests) whether k is in the tree. In a sequential setting, k is in the tree if and only if the traversal has not reached $\perp$.

Examples Fig. 1 shows examples for search trees. An internal binary search tree (BST) consists of nodes storing a single key k and having two fields, `L` and `R`. Traversals begin at `n=root` and proceed to the node pointed to by `n.L`, if `n` $\neq \perp$ and `k<n.k`, or to `n.R`, if `n` $\neq \perp$ and `n.k<k`. An external BST is similar to internal BST except that payloads are kept only at the leaves and the inner nodes serve as routing nodes; thus, their keys often duplicate keys found at the leaves. A ternary search tree consists of nodes storing at most two keys k_1, k_2 and having three fields: `L`, `M`, and `R`. Traversals begin at `n=root` and proceed to the node pointed to by `n.L`, if `n` $\neq \perp$ and `k<n.`k_1, to `n.M`, if `n` $\neq \perp$ and `n.`k_1`<k<n.`k_2, or to `n.R`, if `n` $\neq \perp$ and `n.`k_2`<k` . A 2-D tree consists of nodes storing two dimensional pairs, (x, y), accessed by $n.x$ and $n.y$. Each node has two fields: L and R. Nodes also store the dimension they represent: `x` or `y`, where subsequent nodes represent different dimensions. A traversal for (x, y) proceeds as follows. For nodes representing the dimension `x`, the condition of L is $n.x > x$ and of R is $n.x \leq x$, and for nodes representing `y`, the conditions are $n.y > y$ and $n.y \leq y$ (resp.). A trie consists of nodes storing words and having an edge for each character. The parent of a node with word w is the node with the largest prefix of w (that is not w).

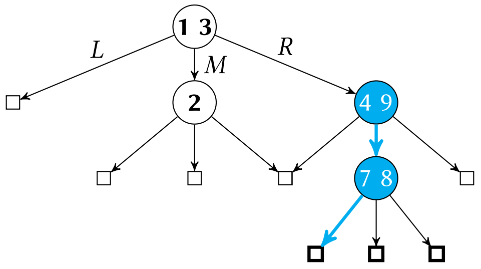
The Challenge of Concurrent Traversals In a concurrent setting, inferring whether a key is in the tree based solely on the traversal may be incorrect, since the traversal may have read an inconsistent state of the tree. To illustrate this, consider two threads A and B traversing the internal BST depicted in Fig. 1(a). Assume A looks for 9 and pauses after reading 12. Then, B executes a removal of 4 by relocating 9 in place of 4. When A resumes, it observes that 12 is a leaf, indicating the traversal has ended. However, inferring from this traversal that 9 is not in the tree is clearly incorrect. Thus, in a concurrent setting, another step is added after the traversal and before the test: a *validation check*. This check determines whether the state the traversal observed is consistent enough for the test. If not, the operation restarts.



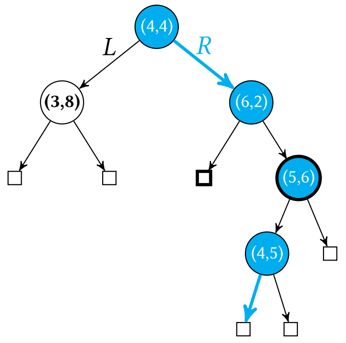
(a) SPS of `search(6)` in internal BST.



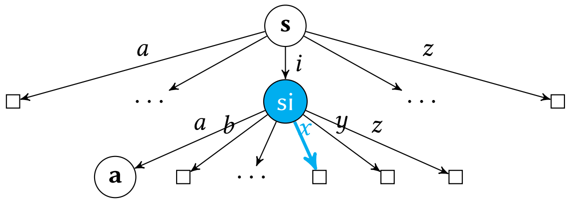
(b) SPS of `search(6)` in external BST.



(c) SPS of `search(6)` in ternary tree.



(d) SPS of `search(4,3)` in 2-D tree.



(e) SPS of `search(six)` in trie.

Figure 1. Succinct path snapshots (SPSs) in several trees.

Goal: Synchronization Free Traversals In this work, we design two building blocks – a traversal and a validation check – which do not require synchronization primitives. An immediate result is a lock-free membership test. We further show that these building blocks can be integrated into updates. Updates begin like membership tests but may write to the tree depending on the test result (indicating whether an update is required). For correctness, they require that the test result will not change while they write. We show that this can be guaranteed by acquiring locks just before our validation check, and not during the traversal. This immediately benefits performance since traversals usually read a large number of nodes (unlike our validation check and typical updates) and thus having to synchronize on these nodes would incur a significant overhead.

In the rest of the paper, we assume the following:

- Each node has a `marked` flag, which is used (as common) to first logically remove a node before physically changing the tree layout. The flag of a newly created node is `false`. When a node is removed from the tree (and eventually becomes unreachable), its `marked` flag is set to `true`. For simplicity's sake, `marked` flags are never toggled back to `false`. If a node's `marked` flag is `false`, the node is said to be *logically* in the tree.
- Nodes' keys are immutable. Thus, updating a node's key is possible only by creating a new node, linking it in place of the old one, and marking the old one. This property enables us to never recheck nodes' keys but rather check whether nodes are logically in the tree.
- Nodes that are logically in the tree are always reachable, even during modifications.

We further assume sequential consistency, though we believe our results can be extended in a straightforward fashion to weaker memory models.

3 Overview

In this section, we informally explain the PaVT condition.

In general, a validation check determines whether a previously traversed path is still suitable for the key being looked for or not, due to concurrent mutations. For example, in the internal BST in Fig. 1(a), the path: $P = (4, R), (12, L), (9, L)$ is suitable for the keys $k \in \{5, ..., 8\}$ but not for any other key. More precisely, a path is suitable for a key k if the condition associated with every node-field in the path (e.g., $< n.k$ or $> n.k$, in a BST) is met by k. To continue our example, for every key $k \in \{5, ..., 8\}$, the conditions $k > 4$, $k < 12$, and $k < 9$ are met. For any other key, at least one of these conditions is violated. For example, for $k = 3$, $k > 4$ is violated. Our main insight leverages the transitivity of the checked conditions to identify, for every path, a minimal set of node-field pairs, which is necessary and sufficient to guarantee that the path is suitable for a key k. This set consists of the maximal nodes with respect to the different conditions. We call this set the *succinct path snapshot* (SPS). In our example, the SPS is $\{(4, R), (9, L)\}$. Correctness follows since for every node-field pair (n, f) of the path, there is a pair (n', f) in the SPS whose condition implies that the condition of (n, f) is met by k. In our example, for $(4, R)$ and $(9, L)$, clearly there is a pair in the SPS whose condition implies their condition, and as for $(12, L)$, the pair $(9, L)$ implies its condition: for any key k for which $k < 9$, the condition $k < 12$ also holds.

The SPS is defined for any path in any search tree. For example, consider a `search(6)` traversing the external BST in Fig. 1(b). The suitable path is $(3, R), (9, L), (9, L)$ and the SPS is $\{(3, R), (9, L)\}$ (for the lower 9). The PaVT condition holds: for every node-field pair in the path, there is a pair in the SPS whose condition implies its condition, for example, for the routing node $(9, L)$, the leaf $(9, L)$ is in the SPS and it implies the condition of the routing node. Consider now a `search(6)` traversing the ternary tree of Fig. 1(c). The suitable path is $((1, 3), R), ((4, 9), M), ((7, 8), L)$ and the SPS is $\{((4, 9), M), ((7, 8), L)\}$. Note that for $((1, 3), R)$, the node ((4,9),M) implies its condition: if $4 < k < 9$, then $k > 3$. Consider now a `search(x=4,y=3)` traversing the 2-D tree in Fig. 1(d). The suitable path is $((4, 4), R), ((6, 2), R), ((5, 6), L), ((4, 5), L)$. The corresponding conditions are $4 \le x$, $2 \le y$, $5 > x$, and $5 > y$, respectively. Since none of the conditions implies the other, here, the SPS is identical to the path. Finally, consider a `search(six)` traversing the trie in Fig. 1(e). The suitable path is $(s, i), (si, x)$. Since the condition checks whether the node's word is a prefix of the searched word, each node implies the condition of its ancestors. Thus, in this example, the SPS is $\{(si, x)\}$.

We note that in internal BSTs, the SPS consists of predecessor and successor pairs in the tree (with respect to the total order of the keys). We have introduced this condition in [12], where we showed that traversals for a key k can be validated by checking k's predecessor and successor in the tree. In [12], we showed that this condition can be efficiently checked by storing at every node its predecessor and successor in the tree. Unfortunately, generalizing this condition to other trees results in an inefficient condition. For example, for m-ary trees, this condition requires extending every node with the predecessor and successor of each of its keys, which increases memory consumption. For external trees, where the predecessor-successor pairs are leaves and do not share a path, this condition prevents updating concurrently predecessor-successor pairs. This is too restrictive compared to other concurrent algorithms that, for example, always allow concurrent insertions to leaves. Even worse, it is unclear whether this condition is applicable for trees whose keys do not adhere to total order, such as K-D trees (e.g., $(3, 2)$ is non-comparable to $(2, 3)$). In contrast, the PaVT condition generalizes to arbitrary search trees.

4 The PaVT Condition

In this section, we formally define the PaVT condition. We define it in four steps. First, we give a condition by which to validate unsuccessful traversals (validating successful traversals is easier, as we shortly explain). This condition relies on having a set of nodes that are known to be on the same path in the tree (at some moment). Second, we characterize the minimal set of nodes that is required for this condition. This set of nodes acts as the path snapshot. Third, we define *succinct path snapshots* (SPSs), which meet stronger conditions compared to path snapshots. This definition enables us to store (succinct) path snapshots at the last node of each path in order to obtain snapshots in a lock free fashion. This later enables us to design a lock-free membership test. The succinct path snapshots can be seen as describing the paths that are logically in the tree, even if due to concurrent mutations the paths are temporarily broken. Finally, we show that the PaVT condition is necessary: any validation condition that does not check it may return incorrect results.

Preliminaries A traversal in a search tree T, `Traverse`$_T$`(k)`, is the exhaustive application of an operation called `nextField`$_T$ starting from the root (Algorithm 1). The operation `nextField`$_T$`(n,k)` takes a node in T and a key and returns either (1) a field f indicating that if k is in T, it is reachable from `n.f` or (2) $\perp$ indicating that k is a key in `n`.

The `nextField`$_T$`(n,k)` operation consists of a series of condition-field pairs $[c_1 \rightarrow f_1], \ldots, [c_l \rightarrow f_l]$ and it returns the first field whose condition is met. A condition is a function from a node n and a key k to a boolean value. We write $c(n,k) = 1$ if the condition c is met for n and k, and $c(n,k) = 0$ otherwise. We say that $c(n,k)$ (logically) implies a condition c' checked on n' and k' if whenever $c(n,k) = 1, c'(n',k') = 1$. The `nextField`$_T$ operation has the property that for every node n and key k, at most one condition is met, regardless of the evaluation order. We can thus refer to *the condition met* in `nextField`$_T$`(n,k)` (if exists). We assume that the conditions do not use the "or" operand. This does not affect generality since a condition of the form $[(c_1 \text{ or } c_2) \rightarrow f]$ can be represented by two distinct conditions: $[c_1 \rightarrow f]$ and $[c_2 \rightarrow f]$. This also explains how different conditions may return the same field (but not vice-versa). A positive example for this case is the `nextField`$_{2D}$ (of the 2-D tree), which for some nodes checks the x coordinate and for some nodes checks the y coordinate, although eventually either `L` or `R` is returned. We assume that for every node, there are no two keys that meet different conditions but return the same field. Another property of `nextField`$_T$ is that by the construction of search trees, if a node n' is reachable from a node n via a field f, then `nextField`$_T$$(n,n'.k_i)$= f for any key k_i in n'. In the following, we refer to this property as the *search trees' property*. For example, in the BST in Fig. 1(a), the nodes 12 and 9 are reachable from the node 4 via R and indeed `nextField`$_T$`(4,12)`= R and `nextField`$_T$`(4,9)`= R.

The PaVT Condition The PaVT condition determines when it is safe to determine that a key is *not* in the tree. This is the challenging decision because determining whether a key *is* in the tree is done by looking for a node (logically in the tree) that has this key. Our next theorem states the PaVT condition. It states that given a set of node-condition pairs that are maximal in their path w.r.t. the condition and given a key k that meets all these conditions, k is not in the tree.

Theorem 4.1 (The PaVT Condition). *Given a tree T, a key k, and a set of node-condition pairs $S = \{(n_{i_1}, c_{i_1}), \ldots, (n_{i_m}, c_{i_m})\}$. If there exists a moment between the traversal's invocation and response where all the following hold:*

Algorithm 1: Traverse$_T$(k)

```
Function nextField_T(n,k):
    if $c_1(n,k) = 1$ then return $f_1$
    ...
    if $c_l(n,k) = 1$ then return $f_m$
    return $\perp$
Function Traverse_T(k):
    n ← root
    while $n \neq \perp$ do
        f ← nextField_T(n,k)
        if $f = \perp$ then return true
        n ← n.f
    return false
```

(1) For every $(n_i, c_i) \in S$, $c_i(n_i, k) = 1$.
(2) $n_{i_1}, \ldots, n_{i_m}$ are logically in T (i.e., their `marked` *is* `false`*).*
(3) There is a path in T linking these nodes.
(4) This path is maximal: n_{i_m}.[`nextField`$_T$$(n_{i_m}, k)$*] $= \perp$.*
(5) For every node n logically in T, either no node in S is reachable from it, or there exists a pair $(n_i, c_i) \in S$ where n_i is reachable from n via a field f, and $c_i(n_i, k)$ implies a condition $c(n,k)$, such that $[c \rightarrow f]$ is in `nextField`$_T$.
Then, there is no node logically in T with the key k.

We illustrate this by example on the BST in Fig. 1(a); a full proof is provided in Appendix A. Given $k = 8$ and the set $\{(n_4, >), (n_9, <)\}$ (whose keys are 4 and 9), all conditions hold. Assume in contradiction 8 is in the tree in n_8. Clearly, n_8 is not n_4 or n_9. Also, n_4 and n_9 are not reachable from n_8, as this contradicts requirement (5). Lastly, n_8 is not reachable from n_9, as this contradicts requirement (4). Since n_4 is the root, this implies that n_8 is not anywhere in the tree. For the general case, where no node is the root, see the proof.

Minimal Set From this theorem, and in particular requirement (5), we can characterize when S is minimal: every condition appears at most once and conditions may be absent if they are implied by other conditions. This bounds the size of the required S by the number of conditions in `nextField`$_T$, which is typically linear in the number of fields.

Corollary 4.2. *Let T be a tree, k a key, and S a set of node-condition pairs satisfying requirements (1)–(5) of Theorem 4.1. S is minimal if for every $(n_i, c_i), (n_j, c_j) \in S$ such that $j > i$: $c_j(n_j, k)$ does not imply $c_i(n_i, k)$.*

From this characterization we can infer that S is unique:

Lemma 4.3. *Let T be a tree, k a key, and S_1, S_2 minimal sets of node-condition pairs satisfying requirements (1)–(5) of Theorem 4.1. Then, either $S_1 \subseteq S_2$ or $S_2 \subseteq S_1$.*

Proof is provided in Appendix A.

Motivation for Succinct Path Snapshots The last corollary and lemma uniquely define the minimal set of nodes, but do not explain how to construct it. Constructing it on-the-fly in a lock free fashion is possible, but may lead to reading an inconsistent snapshot, which requires restarting the traversal. Instead, we decouple requirement (1) of the PaVT condition (which pertains to the key the traversal searches for) from requirements (2)–(5) (which pertain to the nodes on the traversal path), and strengthen the latter to logically capture the paths in the tree. We call the set of nodes satisfying the strengthened requirements the *succinct path snapshots.* The importance of strengthening the requirements is that concurrent mutations can change nodes that link the snapshots' nodes *as long as they do not affect the snapshot.* For example, in the BST in Fig. 1(a) and for the snapshot $\{(n_4, >), (n_9, <)\}$, we would like to allow concurrent updates to the nodes linking them (e.g., n_{12}), as long as n_4 and n_9 are on the same path and they comprise this path's snapshot. For example, we would like to allow concurrent updates that remove n_{12} or add other nodes with keys $k \in \{10, 11, ...\}$. We call the set of keys that can be added to the path without affecting a snapshot S, the *valid keys w.r.t. S*. For example, 10, 11, ... are the valid keys w.r.t. $\{(n_4, >), (n_9, <)\}$. If a path snapshot is not affected by this kind of concurrent changes, we call it the *succinct path snapshot.* We next provide formal definitions.

Succinct Path Snapshots A path is a series of node-field pairs $(n_0, f_0), ..., (n_m, f_m)$ such that: (i) n_0 is the root, (ii) for every $i \in [0, ..., m-1]$: $n_i.f_i = n_{i+1}$, and (iii) $n_m.f_m = \perp$, i.e., no node follows $n_m.f_m$. We say that a condition c corresponds to a field f, if when c is met in $\texttt{nextField}_T$, f is returned. For $f = \perp$, the corresponding condition is $\neg c_1 \wedge ... \wedge \neg c_m$ (met when all other conditions are not met). A node-condition series $(n_0, c_0), ..., (n_m, c_m)$ corresponds to a path $(n_0, f_0), ..., (n_m, f_m)$ if every c_i corresponds to f_i. Given a path $P = (n_0, f_0), ..., (n_m, f_m)$ and its snapshot $S = (n'_0, c_0), ..., (n'_k, c_k)$, the *valid keys w.r.t. S* of a node n in P are all keys that n can have such that the nodes in S remain on the same path in the tree.

Definition 4.4 (A Succinct Path Snapshot.)**.** Let $P = (n_0, f_0), ..., (n_m, f_m)$ be a path and $C = (n'_0, c_0), ..., (n'_k, c_m)$ a node-condition series corresponding to P. $S \subseteq C$ is a succinct path snapshot (SPS) of P if:

(i) n'_0,...,n'_k are logically in T.
(ii) There is a path in T linking these nodes which is maximal: $n'_k.f_k$=$\perp$.
(iii) For every node n logically in T, either no node in S is reachable from it, or there exists a pair $(n_i, c_i) \in S$ where n_i is reachable from n via some field f, such that for every k, $c_i(n_i, k)$ implies $c(n, k)$, that returns f for all valid keys w.r.t. S of n.
(iv) For every $(n_i, c_i), (n_j, c_j) \in S$ such that $j > i$ and for every key k, $c_j(n_j, k)$ does not imply $c_i(n_i, k)$ for some valid key w.r.t. S of n_i.

The succinct path snapshots logically capture the paths in the tree. In particular, the fact that a succinct path snapshot S was obtained implies that the path logically captured by S meets requirements (2)–(5) of the PaVT condition. This simplifies the PaVT condition:

Theorem 4.5 (The PaVT Condition via Succinct Path Snapshots)**.** *Let T be a tree, k a key, and S a succinct path snapshot such that for every $(n_i, c_i) \in S$, $c_i(n_i, k) = 1$. Then, there is no node logically in T with the key k.*

Validating Succinct Path Snapshots is Necessary The validation of the SPS for an unsuccessful traversal (where the key k is not found) is not only sufficient but also necessary. We show that if the tree permits replacements of nodes, every validation check has to check that the snapshot meets the conditions in Theorem 4.1. Intuitively, this follows since omitting any check of these conditions enables a concurrent thread to change the tree in that unobserved spot and add the key under search in a different path reachable from this spot. Thus, any weaker condition may result in returning incorrect traversal result (and as a result possibly incorrectly modifying the tree). Proof is provided in Appendix A.

Lemma 4.6. *Let T be a tree, k a key not in T, and S a succinct path snapshot of k in T (logically capturing a path P in T). Any validation condition that does not verify requirements (1)–(5) of Theorem 4.1 for the nodes in S, cannot distinguish between T and another tree T' that contains k.*

We note that if a tree does not permit addition/replacement of nodes in the middle of the tree, then some conditions of Theorem 4.1 are implicitly guaranteed. The external tree of [14] is such a tree in which nodes can only be added as leaves and only leaves can be removed. This guarantees: (i) the first node of the SPS (which is not checked at the end of the traversal) can be changed but only to one of its ancestors, which guarantees conditions (1) and (3) and, combined with the fact that the last node in the SPS is logically in the tree, also condition (2); (ii) the last node is a leaf, which guarantees condition (4); and (iii) nodes cannot be added in the middle of the tree, which guarantees condition (5).

5 Lock-free PaVT Membership Test

In this section, we describe the PaVT membership test. $\texttt{PaVTTraverse}_T$ (Algorithm 2) leverages Theorem 4.5 to implement a lock-free membership test. It begins similarly to Algorithm 1: it starts from the root (which is a *sentinel* node, as we shortly describe) and continues according to $\texttt{nextField}_T$. If it reaches the end of a path, it validates with the succinct path snapshot, which is atomically read from the last node traversed (n). Being able to read the snapshot atomically from a node, makes $\texttt{PaVTTraverse}_T$ lock-free. $\texttt{PaVTTraverse}_T$ is linearized either (i) when an unmarked node with the key k is found (in which case there is a moment between the invocation and response where k is logically in the tree) or (ii) in the linearization point of the snapshot read at Line 8, which guarantees that this snapshot logically captures the suitable path for k in the tree.

Algorithm 2: PaVT-Traverse$_T$(k)

```
1  n ← root ; $f$ ← nextField$_T$(n,k)
2  while $n.f \neq \perp$ do
3      n ← n.f
4      f ← nextField$_T$(n,k)
5      if $f = \perp$ then
6          if $n.marked$ then restart
7          return $true$
8  $S \leftarrow snapshot(n, f)$
9  if $\exists(n', c') \in S. c'(n', k) = 0$ then restart
10 return $false$
```

To employ this traversal as part of an update (e.g., insertion or removal), we modify it to acquire the lock of n, which serves also as the lock of the snapshot, just before validating and returning n. We call this variation PaVTTraverse$_T$NLock. In the remainder of this section, we explain how to maintain the succinct path snapshots (required at Algorithm 2, Line 8) to guarantee conditions (2), (3) and (5) of Theorem 4.1. This means that a traversal only has to check for conditions (1) and (4) of Theorem 4.1.

Storing SPSs Although snapshots are (supposed to be) read only from leaves, it simplifies their maintenance if every node contains all snapshots containing it. That is, snapshots are stored in every node they consist of. More precisely, for every node n and field f, we store a set of succinct path snapshots containing $n.f$, denoted by $\mathcal{S}_f^n$. We begin with characterizing the sizes of the snapshot sets for different trees. Proof is provided in Appendix A.

Lemma 5.1. 1. *In BSTs, internal or external, where the conditions are $<$ for L and $>$ or $\geq$ for R, $|\mathcal{S}_f^n| \leq 1$ for every node n and field f.*
2. *In internal m-ary trees, where the conditions are $k < n.k_1$, $n.k_1 < k < n.k_2$,..., $n.k_{m-1} > k$, $|\mathcal{S}_f^n| \leq 2$ for every node n and field f.*
3. *In K-D trees, $|\mathcal{S}_f^n| \leq 2^{K-1}$ for every node n and field f.*
4. *In tries, where nodes contain the complete prefix (i.e., the keys are not only on the edges), $|\mathcal{S}_f^n| \leq 1$ for every node n and field f.*

We now describe how to maintain the snapshots to reflect the logical paths in the tree.

Sentinels To avoid edge cases, we make sure all snapshots are of the same size by assuming the tree has a *sentinel* node per condition. Namely, if $[c_1 \rightarrow f_1], ..., [c_l \rightarrow f_l]$ are the condition-field series of nextField$_T$, then there are l sentinel nodes, $n_1^S, ..., n_l^S$, such that $n_i^S.f_i = n_{i+1}^S$. Since every key k has to be reachable from all sentinels, traversals may begin from n_l^S. Thus, we can set the tree root to be n_l^S. Upon creating the tree, the last sentinel node n_l^S has one snapshot for c_l: $\{\{(n_1^S, c_1), ..., (n_l^S, c_l)\}\}$. The other sentinels do not have a snapshot.

Algorithm 3: UpdateSnaps(n)

```
1  Function UpdateSnaps(n):
2      UpdateSnaps($n$, $n$, $n$.P, $\{f_1, ..., f_l\}$)
3  Function UpdateSnaps(n,n',p, fSet):
4      if $fSet = \emptyset$ then return
5      $f$ ←the field such that $p.f = n'$
6      $c$ ←the condition such that $c(p, n'.k_1) = 1$
7      if $\forall k. c(n, k) = 0$ then return
8      Atomically change $(p, c)$ to $(n, c)$ in $S_f^p$
9      $S_f^n \leftarrow S_f^p$; $fSet \leftarrow fSet \setminus \{f\}$
10     UpdateSnaps($n$, $p$, $p$.P, $fSet$)
```

Synchronization and Updates To keep the snapshots in the nodes correct, we require operations that update paths to lock their snapshots. This is obtained by extending each node with a lock and associating the snapshot's lock to the lock of its last node (i.e., the leaf). In general, this means that updating a single path requires one lock. However, if the last node of the snapshot is being removed and is replaced by a node n, then it means that the snapshot changes its lock to n's lock. Thus, n's lock also has to be acquired before the update begins, to guarantee that throughout the operation, changes to the path are blocked.

After executing insertion, removal, or any other mutation, we potentially need to fix the snapshots of the mutated nodes and their descendants. This is done by invoking UpdateSnaps (Algorithm 3) on the mutated nodes. Updating the snapshots is done by traversing from the mutated node up the tree to fix the snapshots by overriding the old node-condition pairs. Traversing up the tree is possible due to the parent pointer that each node has, which is denoted by $n.P$ for a node n (as mentioned in Section 2). When a node is removed, either by replacing it with another node or setting its parent to point to $\perp$ instead of this node, the snapshot of the removed node is copied to the replacing-node/parent (which has to be locked as well, as it is being written to) and the replacing-node/parent looks for a replacement for the removed node in its snapshots. We omit this from the code for simplicity.

Note that since the path snapshot is already locked, UpdateSnaps does not require further synchronization. We also note that for some trees, the snapshot updates can be optimized. For example, in BSTs, the updated snapshots can be computed directly from the snapshots of the mutated nodes or their parents.

Correctness The correctness of UpdateSnaps follows from the fact that while a path is being updated, concurrent updates are blocked. Thus, the updates of the snapshot are guaranteed to reflect those of the path. Formally, this argument can be proven by induction. We omit this proof.

Linearization Although snapshots are not updated atomically with mutations to the tree, they always represent a consistent (logical) path in a tree, which enables us to linearize them (assuming that the tree operations invoke UpdateSnaps on all mutated nodes). To linearize snapshots, we determine a point where the snapshot is an SPS of a path in the tree. The simple case to linearize a snapshot is at the moment of reading its pointer (i.e., S^n_f). However, since snapshots are not updated atomically with mutations to the tree (e.g., insertions), they may be linearized with respect to insertions or removals (like the linearization point of an unsuccessful contains in [19]). Namely, if a snapshot contains a node which has been removed, that snapshot is linearized just before the linearization point of the removal, and if a snapshot does not contain a node whose insertion extends its path, the snapshot is linearized just before the insertion. Intuitively, these linearization points are correct because if a thread A reads a snapshot S, then S logically represents a path that existed in the tree after A has started this operation. Thus, even if in the meantime, a thread B changed the path represented by S, the fact that S has not been updated yet implies that B has not completed its operation. Thus, we can linearize S just before the linearization point of B's operation.

Complexity The complexity of `UpdateSnaps` is $O(h)$ where h is height of the tree. This follows because `UpdateSnaps` updates at most a single snapshot at every node, and the new snapshot is copied to nodes by copying a pointer to it (i.e., nodes of the snapshots are not copied individually). The number of such updates depends on the update operation, but is typically a constant. In Section 6, we show how to optimize `UpdateSnaps` to reduce the complexity of updating a single snapshot to $O(l)$ where l is the size of the snapshot.

6 PaVT-ing Updates of BSTs and AVLs

In this section, we demonstrate how to employ the PaVT traversal with a simple locking protocol to implement arbitrary BSTs, internal and external, and an AVL tree.

Locking Protocol We employ the following locking protocol. Each node has a lock and n_1's lock is acquired before n_2's if (i) n_2 is reachable from n_1 or (ii) n_1 and n_2 are reachable via f_i and f_j, respectively, from their lowest common ancestor and $i < j$. If a node n has to be locked against the locking order or it may be the case, an opportunistic attempt to acquire n's lock is taken (without blocking), and if it fails, locks are released and the operation restarts. To guarantee that restarts do not result in contaminated data, updates begin only after all required locks are acquired. This protocol is deadlock-free and livelock-free because locks are always acquired in the same order by all threads, and the thread that acquires the locks of the lowest nodes (according to the tree) is guaranteed to succeed in acquiring the locks.

PaVT-ing Operations We perform PaVT insertions and removals as follows. We invoke `PaVTTraverseNLock`, which is identical to Algorithm 2 except that right before Lines 7 and 8, it acquires the lock of the last node traversed, n. Then, if the validation of Line 9 fails, the lock is released and the operation restarts. Otherwise, instead of returning `true/false`, `PaVTTraverseNLock` returns n (which is now locked). After `PaVTTraverseNLock` completes, other nodes that are read or written to are locked as well (with respect to the locking order). Then, updates are executed. Finally, the snapshots are updated as described in the previous section, locks are released, and the operation result is returned.

Optimizing the Snapshots As discussed in Section 3, in BSTs, snapshots are of size two (they are predecessor-successor pairs). Thus, instead of maintaining the complete snapshot, nodes can store only the other nodes in their snapshots and infer the condition of that node. For example, in the internal BST in Fig. 1(a), the node 9 can store for L the snapshot: 4. When reading this snapshot, one can infer that the snapshot contains $(9, <)$ and $(4, >)$, since there are only two conditions, and the node corresponding to L is 9.

We next provide the implementation details of insertion and removal, which extend the sequential operations with the PaVT traversal (`contains(k)` is exactly Algorithm 2). To demonstrate the similarity of the PaVT-ed operations to the sequential operations, we mark with rectangles statements added on top of the sequential implementation: the pink, wider rectangles are the PaVT operations, while the gray, shorter rectangles are the lock acquisitions.

Insertion The BST insert operation inserts a key k if k is not in the tree. It begins with traversing from the tree root, turning left or right depending on whether the current node's key is greater or smaller than k, and terminating when k is found or the path has ended. If k is found in a node n, it returns `false`, otherwise it adds a new node after the last node read. The `PaVT-BSTInsert(k)` (Algorithm 4) is very similar to the BST insert and the differences are marked with rectangles. The differences involving PaVT are `PaVTTraverseNLock` and `UpdateSnaps`, while the differences involving locks are: (i) acquisition of the lock of the last node traversed in `PaVTTraverseNLock`, and (ii) release of locks upon completion or inside `PaVTTraverseNLock` if validation fails. Another difference compared to the sequential insertion is that after locking n (that is, n's snapshot suits k), we validate that n is the last node on the path; if not, we restart, since this means that n's lock does not block concurrent updates to its snapshot. The linearization point of an insertion is at the moment the new node becomes reachable.

Removal The BST removal operation is PaVT-ed similarly, and its unique aspect is that it contains a nested traversal in case the node to remove has two children, in which case a nested traversal is required to look for the successor, starting from the node's right child (Line 22).

The BST remove operation removes a key k if k is in the tree. It begins with traversing from the tree root, turning left or right depending on whether the current node's key is greater or smaller than k, and terminating when k is found or the path has ended. If k is found in a node n, the removal is one of the following:

Algorithm 4: PaVTBST-Insert(k)

```
n ← PaVTTraverseNLock(k)
if n.k=k || (k > n.k && n.R ≠ ⊥) || (k < n.k && n.L ≠ ⊥) then
    unlock(n)
    return false
newNode ← new node(k)
newNode.P ← n
(k > n.k? n.R : n.L) ← newNode
UpdateSnaps(newNode)
unlock(n)
return true
```

- If n has at most one child, n's parent is updated to point to this child.
- If n has two children and its right child r does not have a left child (i.e., r is n's successor), n's parent is set to r, and r's left child is set to n's left child.
- If n has two children and its right child has a left child, n's successor, s, is obtained by looking for the leftmost node in the tree rooted with r. Then, s relocates to n's location to remove n.

The `PaVT-BSTRemove` (Algorithm 5) is very similar to the BST removal and the differences are marked with rectangles. The differences involving PaVT are: (i) `PaVTTraverseNLock` is invoked to find n and to find s if n has two children, and (ii) `UpdateSnaps`. The differences involving locks are:

- Locks are acquired on the last node traversed in `PaVTTraverseNLock`.
- Additional locks are acquired on all nodes the removal involves (i.e., n's parent, n's children, and s's parent and right child). In case a lock is acquired against the locking order (e.g., n's parent is locked after n is locked), an optimistic attempt to lock it is taken, and if it fails all locks are released and the operation restarts.
- Locks are released as described for the insertion.
- n's *mark* flag is set right after all locks are acquired.

The linearization point is when n is marked as removed.

Internal AVL The AVL extends the BST's `insert(k)` and `remove(k)` with a rebalance operation that restores the tree after updates, if the updates have violated the AVL invariants. The rebalance operation traverses the tree bottom up from the modified nodes and may *rotate* nodes, i.e., relocate adjacent nodes. Since the snapshots of m-ary trees consist of predecessor-successor pairs, AVL rotations do not affect the snapshots in internal BSTs. Thus, rotations are implemented by traversing the mutated path upwards, acquiring the nodes' locks, and rotating the tree if necessary. This guarantees that at a quiescence state the tree is a valid AVL tree [5].

External BST External trees may be PaVT-ed similarly to an internal BST, but can be further optimized because inner nodes do not have payloads and leaves do not have children. This enables us to condense the node structure by using the same field to denote a payload of a leaf or a child of an inner node. Also, synchronization can be more efficient if we limit the locks to the inner nodes. This is correct since leaves cannot be updated without modifying their parent (insertion modifies the parent to point to the new inner node and removal triggers the removal of the parent). Thus, leaves are synchronized indirectly through their parents and a leaf is indirectly marked as removed by marking its parent.

Algorithm 5: BST-PaVTRemove(k)

```
while true do
    n ← PaVTTraverseNLock(k)
    if n.k ≠ k then
        unlock(n)
        return false
    p ← n.P; l ← n.L; r ← n.R
    if !lock(p,l,r) then
        unlockAll(); continue
    if l = ⊥ || r = ⊥ then
        n.marked ← true
        c ← l = ⊥? r : l
        (p.L = n? p.L : p.R) ←c
        UpdateSnaps(p, c)
        unlockAll()
        return true
    if r.L = ⊥ then
        n.marked ← true
        r.L ← l; l.P ← r; r.P ← p
        (p.L = n? p.L : p.R) ←r
        UpdateSnaps(p, r)
        unlockAll()
        return true
    s ←PaVTTraverseNLock(r,k)          // If failed
     locking s, release locks and restart
    if !lock(s.P,s.R) then
        unlockAll(); break
    n.marked ← true
    sP ← s.R; sR ← s.R
    s.R← r; r.P← s
    s.L← l; l.P← s
    s.P ← p
    (p.L = n? p.L : p.R) ←s
    sP.L← s.R
    if sR ≠ ⊥ then  sR.P= sP
    UpdateSnaps(sR,sP,s,p)
    unlockAll()
    return true
```

7 Evaluation

We evaluated the PaVT-ed internal BST, external BST, and internal AVL. To evaluate PaVT, we compare to state-of-the-art algorithms that have online available Java implementations.

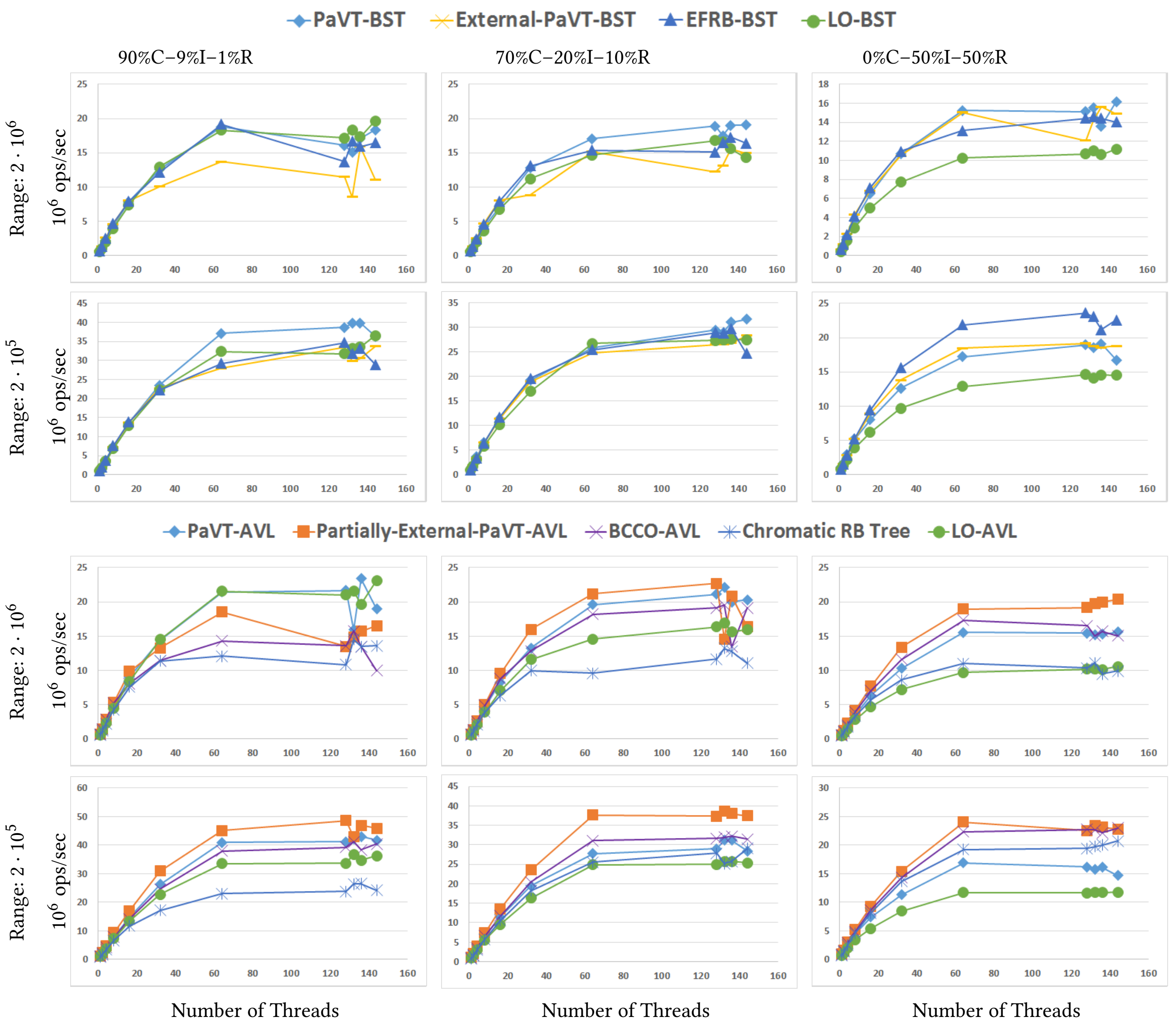


Figure 2. Comparison of binary search tree and AVL implementations.

We compare the (internal) PaVT-BST and External-PaVT-BST to: (i) LO-BST: a lock based, internal BST [12] and (ii) EFRB-BST: a lock-free, external BST [14].

We compare the (internal) PaVT-AVL to:

- LO-AVL: a lock based, internal AVL [12].
- Chromatic RB Tree: a lock-free, external red-black tree, using the LLX/SCX primitives, optimized for updates: a path is fixed when there are six violations [8].
- BCCO-AVL: a lock based, partially-external AVL [7], where internal nodes initially store data (keys) but may become routing nodes if their key is removed, by logically marking them. These nodes may become data nodes again if their key is re-added to the tree.

To fairly compare to BCCO-AVL, which in heavy update benchmarks easily revives logically removed nodes and avoids rebalancing, we compare to a partially external AVL-PaVT. We note that non-blocking data structures provide stronger progress guarantees than lock-based, which require additional overhead in practice. Thus, comparing non-blocking algorithms to blocking algorithms may seem like comparing apples to oranges. However, since sometimes they were shown to obtain better performance than blocking algorithms [8], we compare to these structures as well. The comparison of internal trees to external trees demonstrates the tradeoff between contention and memory consumption.

Experiments ran on an AMD Opteron Processor 6376 with 128GB RAM and 64 cores: 4 processors with 16 cores each and with hyper-threading support. OS was Ubuntu 14.04 LTS and Java was SE Runtime (build 1.8.0_45-b14) with Java HotSpot 64-Bit Server VM (build 25.45-b02, mixed mode).

Our experiments follow the standard empirical evaluation (e.g., [7, 8, 12]). We ran five-second trials and we report the throughput (number of operations per second). In every trial, threads randomly chose the operation type according to a given workload distribution and then randomly chose the key from a given range. The workloads were: (i) 90% `contains`, 9% `insert`, 1% `remove`, (ii) 70% `contains`, 20% `insert`, 10% `remove`, and (iii) 0% `contains`, 50% `insert`, 50% `remove`. The ranges were $2 \cdot 10^5$ and $2 \cdot 10^6$. Before each trial, the tree was prefilled to its expected size: 90% full in workload (i), 2/3 full in workload (ii), and 50% full in workload (iii). Every experiment was run 10 times and the arithmetic average is reported. Every 10 trial batch was run in its own JVM instance and a warm-up phase was run before to avoid HotSpot effects. Threads were not bound to processors. The number of threads was varied over the range 1, 2, 4, 8, 16, 32, 64, 128, 132, 134, 144.

Experimental results (Fig. 2) indicate that the PaVT condition is effective and improves performance over current approaches: PaVT-BST on average outperforms the internal LO-BST on average by 17% and up to 48%, and PaVT-AVL outperforms the LO-AVL on average by 22% and up to 60%. The improvement of PaVT-BST and PaVT-AVL over LO-BST and LO-AVL stems from the fact that the LO trees acquire more locks (as for each predecessor-successor pair there is an additional lock). External-PaVT-BST and EFRB-BST mostly have similar results (on average EFRB-BST is better by 5%). We remind that the EFRB-BST is lock-free and thus has additional overhead compared to the lock-based External-PaVT-BST. The advantage of lock-free algorithms is usually observant under heavy contention (e.g., when the number of threads is bigger than the number of cores). The Partially-External-PaVT-AVL outperforms the partially external BCCO-AVL on average by 13% and up to 65%. This can be attributed to the lock-free membership test that Partially-External-PaVT-AVL has, in contrast to BCCO-AVL.

Lastly, we provide results on the space consumption. Table 1 shows the results for all workloads, with 64 threads and key range $2 \cdot 10^6$ (results for the other scenarios were similar). Results show that the PaVT implementations provide the lowest memory consumption, which indicates that explicitly maintaining the SPSs in the nodes has low overhead.

	90C,9I,1R	70C,20I,10R	0C,50I,50C
PaVT-BST	104,132	140,523	78,111
External-PaVT-BST	175,021	223,410	131,263
EFRB-BST	148,118	193,623	111,236
LO-BST	239,563	323,149	179,733
PaVT-AVL	104,177	140,612	78,144
Partially-External-PaVT-AVL	118,454	145,631	97,101
BCCO-AVL	115,677	145,301	91,041
Chromatic RB Tree	188,375	237,524	142,018
LO-AVL	249,999	337,455	187,544

Table 1. Space consumption in kB for 64 threads, range $2 \cdot 10^6$.

8 Related Work

Current solutions addressing the challenge of validating traversals range from standard locking protocols and universal constructions, through validation conditions for sets, to highly-optimized solutions for specific tree algorithms.

General solutions such as the standard locking protocols (two-phase locking [15], hand-over-hand locking [3], tree locking [24]) address this challenge, however employing them results in poor scalability since they *acquire locks on every node along the traversal path*. Universal constructions are either practical only for data structures with few contention points, such as stacks and queues (e.g., [16, 20, 21]), or provide only general guidelines (e.g., [10]).

Brown et al. [8] introduce the LLX/SCX primitives, which are a multi-word generalization of the LL/SC primitives. However, programmers have to integrate them manually. Further, LLX/SCX is a lock-free technique and thus updates take place by first preparing a new modified subtree and then atomically changing the pointer to the old subtree to point to the new one. Thus, when multiple nodes are modified (e.g., in a balancing operation), *all, but their lowest common ancestor, are reallocated*, which affects performance. In [12], a simple validation condition is proposed for BST, suitable for data structures implementing totally ordered sets. The validation check validates a search for a key k by checking k's predecessor and successor in the tree, which is done efficiently by maintaining the nodes' predecessor and successor. This was shown for internal trees, where the predecessor-successor pairs always share a path. However, in external trees, such approach would link the leaves, which do not share a path, and thus introduce redundant conflicts and contention. DomLock [23] presents a new locking protocol for hierarchal data structures where nodes' keys are contained in their parents' keys (an example for such tree is one whose nodes have intervals for keys and keys of descendant nodes are contained intervals). However, since trees are not always hierarchical, this protocol is inapplicable in our setting.

Lastly, there are many practical implementations of trees (e.g., [7]), with most of them supporting lock-free searches (e.g., [1, 9, 10, 13]), and some offer complete lock-free implementations (e.g., [6, 8, 18, 22, 25]), which typically cope better under heavy contention but less when updates are sparse and searches are dominant [17]. Unfortunately, these support only standard operations and extending them to other operations or trees is challenging. In contrast, we provide a general condition and a lock-free membership test and demonstrate the simplicity of integrating our traversal and validation condition in update operations.

9 Conclusion

We presented PaVT, a necessary and sufficient condition for validating concurrent traversals in search trees. PaVT relies on *succinct path snapshots* to validate traversal paths. Based on this condition, we designed a lock-free membership test. We also showed how to leverage the traversal and validation of the membership test to implement binary search trees, internal and external, and AVL, and experimentally showed they outperform state-of-the-art trees.

References

[1] Maya Arbel and Hagit Attiya. 2014. Concurrent updates with RCU: search tree as an example. In *PODC '14*.

[2] Maya Arbel and Adam Morrison. 2015. Predicate RCU: an RCU for scalable concurrent updates. In *PPoPP '15*.

[3] R. Bayer and M. Schkolnick. 1977. Concurrency of operations on B-trees. *Acta Informatica* 9, 1 (1977).

[4] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Bradley C. Kuszmaul. 2005. Concurrent cache-oblivious b-trees. In *SPAA '05*.

[5] Luc Bougé, Joaquim Gabarró, Xavier Messeguer, and Nicolas Schabanel. 1998. Height-relaxed AVL rebalancing: A unified, fine-grained approach to concurrent dictionaries. (1998).

[6] Anastasia Braginsky and Erez Petrank. 2012. A lock-free B+tree. In *SPAA '12*.

[7] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A practical concurrent binary search tree. In *PPoPP '10*.

[8] Trevor Brown, Faith Ellen, and Eric Ruppert. 2014. A General Technique for Non-blocking Trees. In *PPoPP '14*.

[9] Tyler Crain, Vincent Gramoli, and Michel Raynal. 2013. A Contention-Friendly Binary Search Tree. In *Euro-Par '13*.

[10] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *ASPLOS '15*.

[11] Simon Doherty, David L. Detlefs, Lindsay Groves, Christine H. Flood, Victor Luchangco, Paul A. Martin, Mark Moir, Nir Shavit, and Guy L. Steele, Jr. 2004. DCAS is Not a Silver Bullet for Nonblocking Algorithm Design. In *SPAA '04*.

[12] Dana Drachsler, Martin Vechev, and Eran Yahav. 2014. Practical Concurrent Binary Search Trees via Logical Ordering. In *PPoPP '14*.

[13] Dana Drachsler-Cohen and Erez Petrank. 2014. LCD: Local Combining on Demand. In *OPODIS '14*.

[14] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-blocking binary search trees. In *PODC '10*.

[15] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM* 19, 11 (1976).

[16] Panagiota Fatourou and Nikolaos D. Kallimanis. 2011. A Highly-efficient Wait-free Universal Construction. In *SPAA '11*.

[17] Vincent Gramoli. 2015. More Than You Ever Wanted to Know About Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms. In *PPoPP 2015*.

[18] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *DISC '01*.

[19] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, Bill Scherer, and Nir Shavit. 2005. A Lazy Concurrent List-based Set Algorithm. In *OPODIS '05*.

[20] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Scalable Flat-Combining Based Synchronous Queues. In *Distributed Computing*. LNCS, Vol. 6343.

[21] Maurice Herlihy. 1991. Wait-free Synchronization. *ACM Trans. Program. Lang. Syst.* 13 (1991).

[22] Shane V. Howley and Jeremy Jones. 2012. A non-blocking internal binary search tree. In *SPAA '12*.

[23] Saurabh Kalikar and Rupesh Nasre. 2016. DomLock: A New Multi-granularity Locking Technique for Hierarchies. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*.

[24] Zvi Kedem and Abraham Silberschatz. 1981. A characterization of database graphs admitting a simple locking protocol. *Acta Informatica* 16, 1 (1981).

[25] A. Natarajan and N. Mittal. 2014. Fast Concurrent Lock-Free Binary Search Trees. In *PPoPP '14*.

[26] Nir Shavit. 2011. Data structures in the multicore age. *Commun. ACM* 54, 3 (2011).

[27] V. Luchangco Y. Lev, M. Herlihy and N. Shavit. 2006. A Provably Correct Scalable Skiplist. In *OPODIS '06*.

A Proofs

In this section, we provide all the proofs.

Proof (Sketch) of Theorem 4.1. Assume in contradiction that there is a node n logically in T with the key k. That node is not any of n_{i_1},...,n_{i_m} as otherwise it would not be true that for every $(n_i, c_i) \in S$, $c_i(n_i, k) = 1$ (since $\texttt{nextField}_T(n_i\texttt{,k})$ has to return $\perp$ in this case). Also, there is no n' in S such that n' is reachable from n, since otherwise there is a pair $(n_i, c_i) \in S$ such that n_i is reachable from n, $c_i(n_i, k) = 1$, and $c_i(n_i, k)$ logically implies a condition $c(n, k)$ in $\texttt{nextField}_T\texttt{(n,k)}$. Thus, $\texttt{nextField}_T\texttt{(n,k)} \neq \perp$ and thus k is not in n. Also, n cannot be reachable from all nodes in S, since the last node is maximal: n_{i_m}.[$\texttt{nextField}_T(n_{i_m}$,k)]=$\perp$, indicating no node follows n_{i_m}. Thus, n is not reachable from any of n_{i_1},...,n_{i_m} and n_{i_1},...,n_{i_m} are not reachable from n. Consider the lowest common ancestor of n_{i_1} and n, denoted by n', and assume n is reachable from n' via f and n_{i_1} via f'. On the one hand, since n_{i_1} is reachable from n' via f', there exists a pair $(n_i, c_i) \in S$ such that n_i is reachable from n' and $\texttt{nextField}(n_i\texttt{,k})$ logically implies some $c(n, k)$ such that $[c \to f']$ is in $\texttt{nextField}_T$. Thus, $\texttt{nextField}_T\texttt{(n,k)}=f'$. On the other hand, since k is in n which is reachable from n' via f, $\texttt{nextField}_T\texttt{(n',k)}= f$ by the search trees' property. Since $f \neq f'$, two different conditions are met in $\texttt{nextField}_T\texttt{(n',k)}$ – contradiction. □

Proof (Sketch) of Lemma 4.3. Assume there are two minimal sets $S_1 \neq S_2$. Then, w.l.o.g. there exists $(n, c) \in S_1 \setminus S_2$. Since (n, c) is not in S_2, there exists $(n', c') \in S_2$ such that $c'(n', k)$ logically implies $c(n, k)$. However, since (n, c) is in S_1, (n', c') is not in S_1, as otherwise S_1 would not be minimal. Thus, there exists $(n'', c'') \in S_1$ such that $c''(n'', k)$ logically implies $c'(n', k)$. By transitivity, $c''(n'', k)$ logically implies $c(n, k)$, namely S_1 is not minimal – contradiction. □

Proof (Sketch) of Lemma 4.6. Let v be a validation condition that does not check all requirements (1)–(5) of Theorem 4.1

for the snapshot of k in T, S_k. We show for every check that without checking it, v cannot distinguish between T and a tree containing k. This implies that v is insufficient.

(1) If there is $(n_i, c_i) \in S$, for which it is not checked that $c_i(n_i, k) = 1$. Since $(n_i, c_i) \in S$, for every $j > i$, $c_j(n_j, k)$ does not logically imply $c_i(n_i, k)$ for any valid key w.r.t. S of n_i. In particular, the conjunction $c_{i+1}(n_{i+1}, k) \wedge \ldots \wedge c_{i_m}(n_{i_m}, k)$ does not logically imply $c_i(n_i, k)$ for any valid keys w.r.t. to S of n_i. However, this conjunction is satisfiable (e.g., by the nodes in S). Thus, there exists a valid key k' for n_i for which $c_i(n_i, k) = 0$. Consider a tree T' identical to T. Now let a thread A traverse T' and stop just before executing v. Then, another thread B replaces n_i with a node containing the key k'. Now B adds k to T'. Since $c_i(n_i, k) = 0$, k is added either to n_i or through a different field than the one returned by c_i. Now A resumes and executes v. However, v cannot distinguish between T and T'.

(2) If there is $(n_i, c_i) \in S$, for which it is not checked that n is logically T. Consider a tree T' identical to T but with a node n_k containing k linked to the last node in S (note that the search trees' property is met). Now let a thread A traverse T' and stop just before reading n_k. Then, another thread B replaces n with a node containing the key k' (as described in the previous bullet), connects to it n_k (as described in the previous bullet), unlinks n_k from its old position, and marks n as logically removed. Now A resumes and executes v. However, v cannot distinguish between T and T'.

(3) If it is not checked whether there is a path between two nodes n_i and n_{i+1} in the succinct path snapshot. Consider n, the parent of n_i in T, linking to it via f ($n.f = n_i$). Let T' be a tree identical to T but with a node n_k containing k linked to the last node in S. Let a thread A traverse T' and stop just before reading n_k. Then, another thread B replaces n_i with a node n' with k'. The key k' is obtained similarly to bullet 1, but if there are multiple possible keys satisfying the conjunction but not $c_i(n_i, k)$, k' is selected to be such that connecting $n'.f' = n_i$ meets the search trees' properties, for $f' \neq f$ (if there is no such k', n_i and n_{i+1} must be linked in a path and thus this check is implicitly verified). Then, B connects n_k to n_i and unlinks n_k from its old position. Finally, A resumes and executes v; however, v cannot distinguish between T and T'.

(4) If the path is not maximal. Consider a tree that right after $n_{i_m}.f_{i_m}$ is linked to a node containing k. Note that this tree respects the search trees' property, but v cannot distinguish between T and this tree.

(5) If there is a node n logically in T, such that there is a node in S reachable from n and there is no pair $(n_i, c_i) \in S$ where n_i is reachable from n via some field f, and $c_i(n_i, k)$ logically implies a condition $c(n, k)$, such that $[c \longrightarrow f]$ is in $\texttt{nextField}_T$: similar to bullet 1.

□

Proof (Sketch) of Lemma 5.1.

1. Assume in contradiction there is n and f such that $|\mathcal{S}_f^n| > 1$ and assume w.l.o.g that $f = L$. If $|\mathcal{S}_f^n| > 1$, there are (at least) two paths that contain $(n, <)$ in their snapshots. Thus, $n.L \neq \perp$ (as leaves are part of one path). Thus, there are two paths for which $n_1.R = n_2.R = \perp$ and their snapshots are $(n, <), (n_1, >)$ and $(n, <), (n_2, >)$. Consider n_{lca}, the lowest common ancestor of n_1 and n_2. We split to cases:
   - If n is reachable from n_{lca}, then w.l.o.g. n_1 is reachable from n_2. If n_2 is reachable from $n_1.R$, then $n_1.R \neq \perp$ – contradiction. Otherwise, n_2 is reachable from $n_1.L$. Thus, the condition $< (n_1, k)$ is not logically implied by $(n, <)$ or $(n_2, >)$. Thus, $(n, <), (n_2, >)$ is not a valid snapshot – contradiction.
   - If n_{lca} is reachable from n, then w.l.o.g. n_1 is reachable from n_{lca} via L. Thus, the condition $< (n_{lca}, k)$ is not logically implied by $(n, <)$ or $(n_1, >)$. Thus, $(n, <), (n_1, >)$ is not a valid snapshot – contradiction.
2. Succinct path snapshots of internal m-ary trees consist of nodes whose keys are each other predecessor-successor in the tree. This follows from the requirement that the path snapshots contain node-condition pairs that imply the former conditions on the path. This means that snapshots are either a single node-condition pair of the form $(n, n.k_i < k < n.k_{i+1})$ or two node-condition pairs, where at least one is $< n.k_1$ or $< n.k_{m-1}$. Since every key has at most one predecessor and one successor, the outer fields ($< n.k_1$ and $< n.k_{m-1}$) are part of exactly one snapshot (where they are the successor and predecessor, resp.), and the inner fields ($n.k_i < k < n.k_{i+1}$) are part of at most two snapshots (one where k_i is the predecessor and one where k_{i+1} is the successor).
3. By a similar argument, the succinct path snapshots consist of predecessor-successor pairs in every dimension (i.e., snapshots are of size $2K$). Since dimensions are independent, if n is reachable from n', and n's condition depends on a dimension a whereas n''s condition depends on a dimension b, n' can be part of the snapshots of both $n.L$ and $n.R$. That is, a predecessor-successor pair of dimension k_1 can be part of every path depending on the other dimensions: it can be part of a path continuing to L or to R on dimension k_2, then on a path continuing both to L or R from both these nodes, etc., up to the k_K dimension. Since the number of different paths containing a given predecessor-successor is at most 2^{K-1}, we get the bound.
4. Follows because if every node contains the prefix, its condition logically implies all preceding nodes on its path. Thus, snapshots consist of a single node, which is the last node on the path and thus $|\mathcal{S}_f^n| \leq 1$.

□