Quadboost: A Scalable Concurrent Quadtree

Keren Zhou, Guangming Tan, Wei Zhou

Abstract—Building concurrent spatial trees is more complicated than binary search trees since a space hierarchy should be preserved during modifications. We present a non-blocking quadtree (*quadboost*) that supports concurrent insert, remove, move, and contain operations, in which the move operation combines the searches for different keys together and modifies different positions atomically. To increase its concurrency, a decoupling approach is proposed to separate physical adjustment from logical removal within the remove operation. In addition, we design a continuous find mechanism to reduce the search cost. Experimental results show that quadboost scales well on a multi-core system with 32 hardware threads. It outperforms existing concurrent trees in retrieving two-dimensional keys with up to 109% improvement when the number of threads is large. Furthermore, the move operation achieves better performance than the best-known algorithm with up to 47%.

Index Terms—Concurrent Data Structures, Quadtree, Continuous Find, Decoupling, LCA

1 Introduction

As multi-core processors are popular in computer systems, there is a need to develop data structures that provide efficient and scalable multi-threaded execution. At present, concurrent data structures [1], such as stacks, linked-lists, and queues have been extensively investigated, causing significant performance benefits in parallel programs [2], [3].

The design of concurrent trees is much more challenging due to its adjusting and rebalancing characteristics. Recent research is focused on binary search trees (BSTs) [4], [5], [6], which are considered as the fundamental parts of tree-based algorithms. The concurrent paradigms of BSTs are extended to design concurrent spatial trees like R-Tree [8], [9]. However, there remains another unaddressed spatial tree–quadtree, which is widely used in applications for multi-dimensional data. For instance, spatial databases, like PostGIS [10], adopt octree, a three-dimensional variant of quadtrees, to build spatial indexes. Video games apply quadtrees to handle collision detection [11]. In image processing [12], quadtrees are used to decompose pictures into separate regions.

There are different categories of quadtrees according to the type of data a node represents, where two major types are region quadtree and point quadtree [13], [14]. The point quadtree stores data points in each node. It is hard to design concurrent algorithms for the point quadtree since an insert operation might involve re-balance issues. Up-to-date solutions for concurrent balanced trees involve specialized search algorithms [15] or mechanisms that block threads [16]. Besides, a remove operation needs to copy a whole subtree under the removed node and re-insert it, leading to a great number of memory operations that decrease scalability in multi-core environment. The region quadtree divides a given region into several sub-regions, where internal nodes represent regions and leaf nodes store data points. Our work focuses on the region quadtree for two reasons: (1) The shape of the region quadtree is independent of insert/remove operations' order, allowing us to avoid complex re-balance rules. (2) We can remove nodes from the region quadtree without copying a whole subtree such that the performance scales up with the growth of threads. Therefore, we will refer to region quadtree as *quadtree* in the following context.

In this paper, we design a non-blocking quadtree, referred to as *quadboost*, that supports concurrent contain, insert, remove, and move operations [13], [17]. There are three primary obstacles: (1) The move operation may modify the quadtree at two different places. To make it correct, we need to ensure that threads are aware of two changes simultaneously. Hence, we attach an object on the nodes before the first changes happen, letting threads detect on-going modifications. (2) The second obstacle comes from removing nodes that do not have data points. Because a remove operation only erases nodes without adjusting the quadtree's structure, some subtrees may not have data points. During the traversal, we push every node from the root to the terminal into a stack and compress nodes from the bottom up. (3) By applying traditional concurrent trees' paradigms [4], [5] to remove nodes from the quadtree, we have to flag or lock two levels of nodes before deletion. In our implementations, we first erase data points and only adjust the structure when all children do not contain data points. Thus, this decoupling approach reduces the complexity of design and increases concurrency.

To the best of our knowledge, this is the first in-depth study of concurrent quadtrees. Our key contributions are as follows:

- We propose the first non-blocking quadtree, which records traversal paths to remove nodes and adjust the structure, adopts a decoupling technique to increase the concurrency, and devises a continuous find mechanism to reduce the cost of retries induced by compare-and-swap (CAS) failures.
- We design a lowest common ancestor (LCA) based

- *K.Zhou and G.Tan are with Institute of Computing Technology, Chinese Academy of Sciences.*
- *W.Zhou is with School of Software, Yunnan University.*

move operation, which traverses a common path for two different keys and modifies two distinct nodes atomically.

- We prove the correctness of quadboost and provide a Java implementation. Comparing with other concurrent trees, the experiments demonstrate that quadboost is highly efficient for concurrent updates at different contention levels.

The rest of this paper is organized as follows. In Section 2, we overview some basic operations. Section 3 describes a simple CAS quadtree for motivating our work. Section 4 provides detailed algorithms for quadboost. We provide a sketch of correctness proof in Section 5. Experimental results are discussed in Section 6. Section 7 summarizes related works. And Section 8 concludes the paper.

2 Preliminary

A quadtree is a non-duplicated dictionary for retrieving two-dimensional key pairs $\langle keyX, keyY \rangle$, where are planar coordinates in some given region. Figure 1 illustrates a sample quadtree and its corresponding region, where we use numbers to indicate keys. Labels on edges refer to routing directions–Southwest (sw), Northwest (nw), Southeast (se), and Northeast (ne). The right picture is a mapping of the quadtree on a two-dimensional region, where keys are located according to their coordinates, and regions are divided by their corresponding width (w) and height (h). There are three types of nodes in the quadtree, which represent different regions on the right side of Figure 1. *Internal* nodes are circles on the left side of Figure 1, and each of them has four children which indicate four equal sub-regions on different directions. The root node is an *Internal* node, and it takes the largest region. *Leaf* nodes and *Empty* nodes are located at the terminal of the quadtree; they indicate the smallest regions on the right side of Figure 1. *Leaf* nodes are solid rectangles that store keys, and they represent regions with the same numbers on the right side. *Empty* nodes do not contain keys.

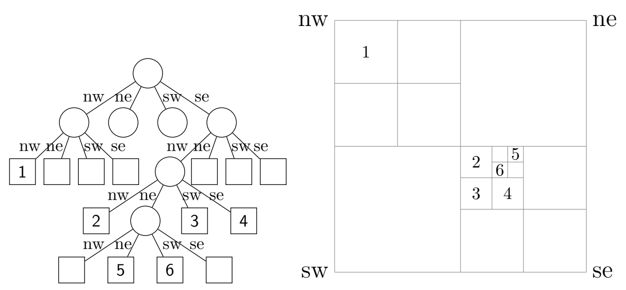


Fig. 1: A sample quadtree and its corresponding region

We describe the detailed structures of the quadtree in Figure 2. An *Internal* node maintains its space information–$\langle x, y, w, h \rangle$ to represent an area that starts from $\langle x, y \rangle$ and has width w and height h. Four child nodes (nw, ne, sw, se) stand for four equal size sub-areas that start from $\langle x, y \rangle$, $\langle x + w/2, y \rangle$, $\langle x, y + h/2 \rangle$, $\langle x + w/2, y + h/2 \rangle$ respectively. A *Leaf* node stores key and value, and an *Empty* node does not have any field. We avoid some corner cases by splitting

```
class Node<V> {}

class Internal<V> extends Node<V> {
    double x, y, w, h;
    Node nw, ne, sw, se;
}

class Leaf<V> extends Node<V> {
    double keyX, keyY;
    V value;
}

class Empty<V> extends Node<V> {}
```

Fig. 2: Quadtree node structures

the root node and its children to form two layers of dummy *Internal* nodes with a layer of *Empty* nodes at the terminal.

Figure 3 shows a sequential quadtree algorithm adopted from [13]. To locate a *Leaf* node, the algorithm uses the find function at line 45 to find out sub-areas iteratively. For example, if we intend to locate node 1 in Figure 1, the getQuadrant function compares node 1's $\langle keyX, keyY \rangle$ with the root's space information $\langle x, y, w, h \rangle$ and locate it in the the nw by *getQuadrant* (line 52). Next, the getQuadrant function compares $\langle keyX, keyY \rangle$ with nw's routing information and pinpoints node 1.

There are four basic operations that rely on the find function:

1) *insert(keyX, keyY, value)* adds a node that consists of $\langle keyX, keyY \rangle$ and *value* into a quadtree. It returns *true* if $\langle keyX, keyY \rangle$ does not exist in the tree. It returns false if $\langle keyX, keyY \rangle$ exists. We first locate $\langle keyX, keyY \rangle$'s candidate position by calling the find function (line 17). If we find an *Empty* node at the terminal, we directly replace it with the node (line 23). Otherwise, if we find a *Leaf* node, we replace it with a subtree that contains both new key and the key of the *Leaf* node. Figure 4a illustrates a scenario of inserting node 7 (*insert(7)*) as a neighborhood of node 2. The parent node is split, and node 7 is added on the *ne* direction.
2) *remove(keyX, keyY)* deletes an existing node from a quadtree. It returns *true* if $\langle keyX, keyY \rangle$ exists. It returns false if the $\langle keyX, keyY \rangle$ does not exist. To remove a node, we also begin by locating $\langle keyX, keyY \rangle$ (line 27) and erase the node (line 29). Then, we take another traversal to compress the tree if necessary (line 31). Particularly, if an *Internal* node in the path contains a single *Leaf* node, we record the node and re-insert it into the upper level. Or if an *Internal* contains four *Empty* children, we replace the *Internal* node with an *Empty* node. Take Figure 4b as an example, if we remove node 5 (*remove(5)*) from the quadtree, node 6 will be linked to the upper level.
3) *move(oldKeyX, oldKeyY, newKeyX, newKeyY)* replaces an existing node with $\langle oldKeyX, oldKeyY \rangle$ by a node with $\langle newKeyX, newKeyY \rangle$. It returns true if the old key exists and the new key does not exist. It returns false if the old key does not exist or the new key exists. A move operation is the combination of insert and remove operations. Consider the scenario

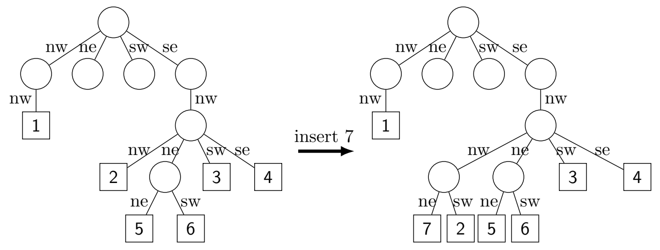
```
11  bool contain(double keyX, double keyY) {
12    p, l := find(root, keyX, keyY);
13    if (inTree(l, keyX, keyY)) return true;
14    return false;
15  }

16  bool insert(double keyX, double keyY, V value) {
17    p, l := find(root, keyX, keyY);
18    if (inTree(l, keyX, keyY)) return false;
19    // createNode: Creates a sub-tree by splitting l until
20    // the candidate region of <keyX, keyY> is not a Leaf
21    // node and returns the sub-tree's root node.
22    Node newNode := createNode(p, l, keyX, keyY, value);
23    replace(p, l, newNode);
24    return true;
25  }

26  bool remove(double keyX, double keyY) {
27    p, l := find(root, keyX, keyY);
28    if (!inTree(l, keyX, keyY)) return false;
29    replace(p, l, new Empty<V>());
30    // compressEmpty: Erase unnecessary Empty nodes
31    compressEmpty(p);
32    return true;
33  }

34  bool move(double oldKeyX, double oldKeyY, double newKeyX, double
          newKeyY) {
35    ip, il := find(root, newKeyX, newKeyY);
36    if (inTree(il, newKeyX, newKeyY)) return false;
37    rp, rl := find(root, oldKeyX, oldKeyY);
38    if (!inTree(rl, oldKeyX, oldKeyY)) return false;
39    newNode := createNode(ip, il, newKeyX, newKeyY, rl.value);
40    replace(rp, rl, new Empty<V>());
41    replace(ip, il, newNode);
42    compressEmpty(rp);
43    return true;
44  }

45  Node find(Node l, double keyX, double keyY) {
46    while (l.class == Internal) {
47      p := l;
48      l := getQuadrant(l, keyX, keyY);
49    }
50    return p, l;
51  }

52  Node getQuadrant(Node l, double keyX, double keyY) {
53    if (keyX < l.x + l.w / 2) {
54      if (keyY < l.y + l.h / 2) l := l.nw; else l := l.sw;
55    } else {
56      if (keyY < l.y + l.h / 2) l := l.ne; else l := l.se;
57    }
58    return l;
59  }

60  void replace(Internal p, Node oldChild, Node newChild) {
61    if (p.nw == oldChild) p.nw := newChild);
62    else if (p.ne == oldChild) p.ne := newChild;
63    else if (p.sw == oldChild) p.sw := newChild;
64    else p.se := newChild;
65  }

66  bool inTree(Node node, double keyX, double keyY) {
67    return node.class == Leaf and node.keyX == keyX and node.keyY ==
            keyY;
68  }
```

Fig. 3: Sequential quadtree algorithm

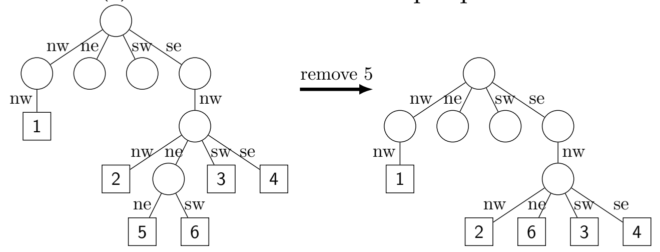
in Figure 4c, after removing node 5 and inserting node 7 with the same value (*move(1, 7)*), the new tree appears on the right part.

4) *contain(keyX, keyY)* checks whether $\langle keyX, keyY \rangle$ is in a quadtree by calling the find function at line 12. It returns *true* if $\langle keyX, keyY \rangle$ exists; it returns false if the $\langle keyX, keyY \rangle$ does not exist.

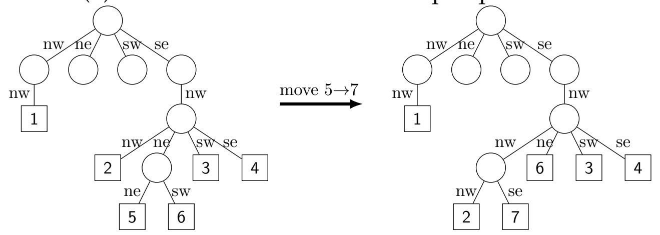
1. *Empty* nodes are not drawn



(a) Insert node 7 into the sample quadtree



(b) Remove node 5 from the sample quadtree



(c) Move the value of node 5 to node 7 from the sample quadtree

Fig. 4: Sample quadtree operations[1]

3 CAS Quadtree

There are a plenty of concurrent tree algorithms, yet a formally presented concurrent quadtree has not been studied. In the sequential algorithm, only the replace function at line 60 modifies the quadtree. Inspired by previous concurrent designs [4], [5], [6], we can devise the helpReplace function at line 81 to substitute the replace function, which adopts CAS instructions to swing pointers atomically, making concurrent insert and remove operations correct. If a CAS fails, it tries to locate the target node again (line 70 and line 76). Besides, we eliminate the compress process (line 31) in the remove operation. We name the new algorithm CAS quadtree.

CAS quadtree is non-blocking. However, it has several limitations: (1) It cannot implement the concurrent operation by using the helpReplace function to substitute the replace function. Since the move operation may modify two different nodes in a quadtree, if we update two positions separately, concurrent threads that are modifying one of them cannot be aware of the ongoing move operation and may return incorrect results. Figure 5 shows such a scenario induced by the incorrect move implementation. (2) Consider if there are a considerable proportion of remove operations. By applying the CAS quadtree algorithm, we still have a large number of nodes in the tree because remove operations substitute existing nodes with *Empty* nodes without structural adjustment. Figure 7 illustrates a detailed example, showing that there remains a chain of *Internal* nodes that do not contain *Leaf* nodes. In this way, not only do we have to traverse a long path to locate the terminal node, but

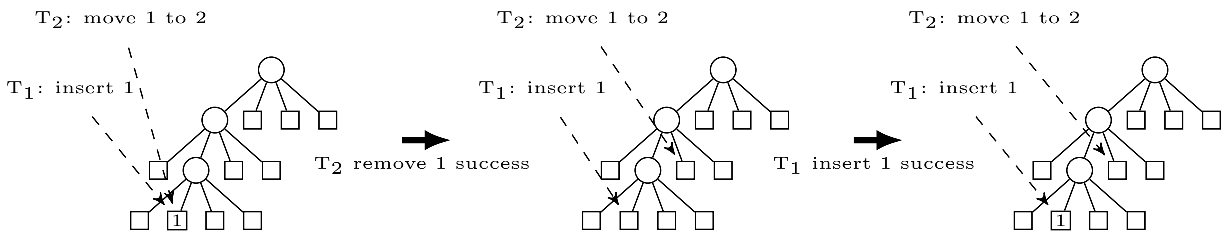


Fig. 5: An example of an incorrect move operation. Thread T_1 intends to insert node 1 into a quadtree, and thread T_2 plans to move the value from node 1 to node 2. T_2 first successfully removes node 1 and then attempts to insert node 2 into the quadtree. In the interval node 1 is added back by T_1, and is not aware of the ongoing move action by T_2.

```
bool insert(double keyX, double keyY, V value) {
  while (true) {
    ... // Line 17-22
    if (helpReplace(p, l, newNode)) return true;
  }
}

bool remove(double keyX, double keyY, V value) {
  while (true) {
    ... // Line 27-28
    if (helpReplace(p, l, newNode)) return true;
  }
}

bool helpReplace(Internal p, Node oldChild, Node newChild) {
    if (p.nw == oldChild) return CAS(p.nw, oldChild, newChild);
    else if (p.ne == oldChild) return CAS(p.ne, oldChild, newChild);
    else if (p.sw == oldChild) return CAS(p.sw, oldChild, newChild);
    else if (p.se == oldChild) return CAS(p.se, oldChild, newChild);
    return false;
}
```

Fig. 6: CAS quadtree algorithm

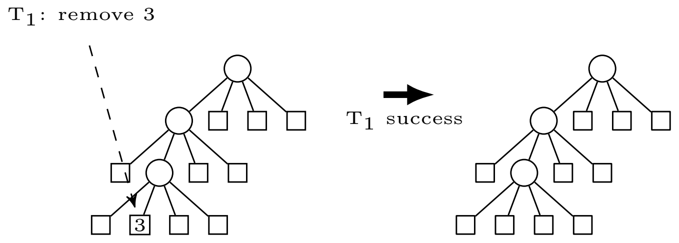


Fig. 7: Thread T_1 intends to remove node 3 from the quadtree. After its removal, there remains a chain of *Internal* nodes that do not contain *Leaf* nodes

also a plenty of *Empty* nodes are left in the memory after the remove operation.

These drawbacks therefore motivate us to develop a new concurrent algorithm to make the move operation correct and employ an efficient mechanism to compress the quadtree.

4 QUADBOOST

4.1 Rationale

In this section, we describe how to design *quadboost* algorithms to solve the two problems addressed in Section 3.

Support the move operation To make the move operation correct, we should ensure that threads know whether a terminal node is being modified. Hence, we attach each internal node a separate object–*Operation* (*op*) to represent its state and record sufficient information to complete the operation. We instantiate the attachment behavior as a CAS and call it a *flag* operation. We also design different *op*s for insert, remove and move operations. The detailed descriptions of structures and a state transition mechanism are presented in Section 4.2.

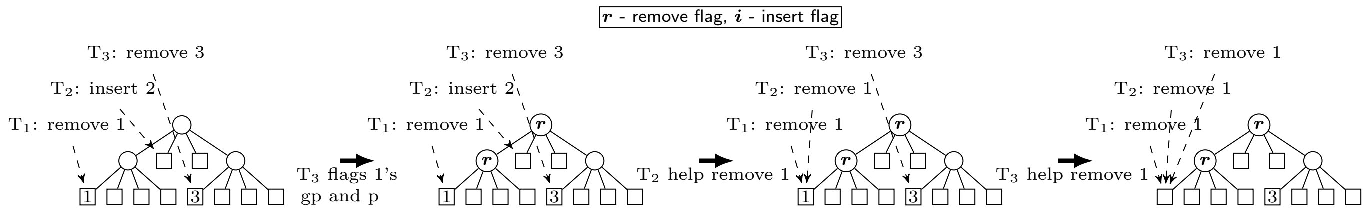
Decouple physical adjustment from logical removal To erase *Empty* nodes from a quadtree, we can apply a similar paradigm in the concurrent BST's removal [4] as shown in Figure 8a, which flags both the parent and the grandparent of a terminal node. However, this method lacks concurrency as other children of the grandparent could not be modified concurrently. Different from the BST's removal in which every time the parent has to be adjusted, we observe that the quadtree only have to compress the parent when there is only a single *Leaf* child. We specify the adjustment condition to where the parent could be compressed if all children are *Empty*. Hence, we devise two steps in the remove operation to increase concurrency. First, we attach an *op* on the parent to indicate one of its children has to be replaced, which is called **logical removal** because the parent is not adjusted. Then, we attach another *op* on the parent if all children are *Empty*, indicating the parent has to be adjusted, which is the **physical adjustment** since the tree's structure will be changed. Consequently, we separate the removal of a key and the adjustment of the structure into two phases. Figure 8b illustrates a concrete example: three threads that handle different *op*s could run in parallel.

Compress quadtree There is still a problem left after applying the above two methods. Recall the example in Figure 7. We flag the bottom *Internal* node to indicate that it should be compressed, but after replacing the bottom *Internal* node with an *Empty* node, it results in four *Empty* nodes in the last level. How do we remove a series of nodes from a quadtree in a bottom-up way? Our solution is to record the entire traversal path from the root to a terminal node in a stack.

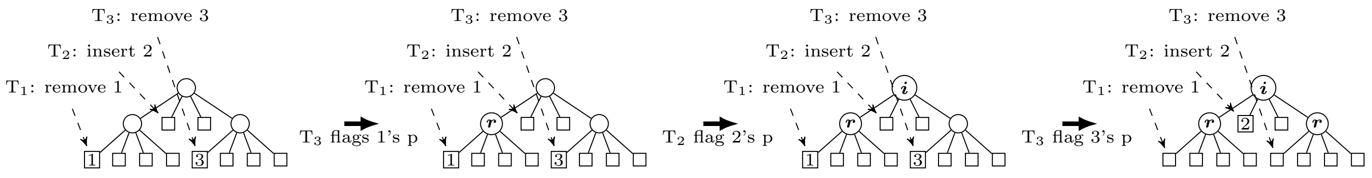
Continuous find Since the traversal path will be altered when a node is compressed, we only have to restart locating the terminal node from its parent if the parent has an *op* other than *Compress*. We name it as the continuous find mechanism.

4.2 Structures and State Transitions

Figure 9 shows the data structures of *quadboost*. As mentioned in Section 4.1, we add an *Operation* structure to handle concurrency issues. Four sub-classes of *Operation*,



(a) Apply traditional removal



(b) Our decoupling approach

Fig. 8: At the beginning, three threads are performing different operations concurrently: (1) T_1 removes key 1 in the lower level. (2) T_2 inserts key 2 in the upper level. (3) T_3 removes key 3 in the lower level. Consider the scenario that T_1 precedes others threads, and then both 1's parent and grandparent will be flagged. Hence, in Figure 8a T_2 and T_3 will help T_1 remove key 1 before restarting their operations. By applying our decoupling method (Figure 8b, only the parent node should be flagged. Hence three threads could run without intervention.

```
class Internal<V> extends Node<V> {
  ...// line 3-4
  Operation op : Clean;
}

class Leaf<V> extends Node<V> {
  ...// line 7-8
  Move op;
}

class Operation {}

class Substitute extends Operation {
  Internal parent;
  Node oldChild, newNode;
}

class Compress extends Operation {
  Internal grandparent, parent;
}

class Move extends Operation {
  Internal iParent, rParent;
  Node oldIChild, oldRChild, newIChild;
  Operation oldIOp, oldROp;
  bool allFlag : false, iFirst : false;
}

class Clean extends Operation {}

class Record {
  Operation pOp;
  Internal p;
  Node l;
  Stack<Node> path;
  Internal lca : root;
};
```

Fig. 9: quadboost structures

including *Substitute*, *Compress*, *Move*, and *Clean*, describe all states in our algorithm. *Substitute* provides information for the insert and remove operations that are designed to replace an existing node with a new node. Hence, we shall let other threads be aware of its parent, child, and a new node for substituting. *Compress* provides information for physical adjustment. We erase the parent node, previously connected to the grandparent, by swinging the link to an *Empty* node. *Move* stores both *oldKey*'s and *newKey*'s terminal nodes, their parents, their parents' prior *op*s, and a new node. Moreover, we use a bool variable–*allFlag* to indicate whether two parents have been attached on *Move op*s. Another bool variable–*iFirst* is used to indicate the attaching order. For instance, if *iFirst* is true, *iParent* will be attached with a *Move op* before *rParent*. *Clean* means that there is no thread modifying the node. In contrast to Figure 6, we add an *op* field in *Internal* and *Leaf* nodes to hold their states. To hold necessary nodes and *op*s during search, a *Record* structure is created.

Each basic operation, except for the contain operation, starts by changing an *Internal* node's *op* from *Clean* to other states. The three basic operations, therefore, generate a corresponding state transition diagram in Figure 10 which provides a high-level description of quadboost algorithms. In the figure, we use flag operations–*iflag*, *rflag*, *mflag*, *cflag* to represent the helpFlag function (line 119) that attaches a *Substitute* (created by insert and remove operations), *Move*, and *Compress newOp* respectively. Besides, *unflag* represents the helpFlag function that changes a node's *op* to *Clean*. Here, we describe how these transitions execute from a state as follows:

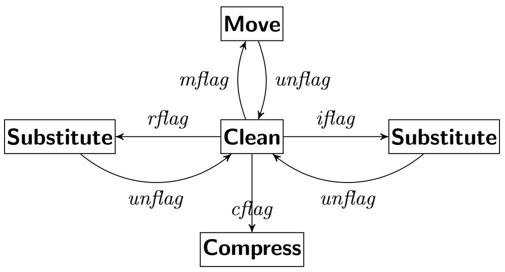


Fig. 10: State transition diagram

- **Clean** (line 119). For the insert transition, it constructs a new node and changes the parent's *op* to *Substitute* by *iflag*. For the remove transition, it creates an *Empty* node and changes the parent's *op* from *Clean* to *Substitute* by *rflag*. For the move transition, it flags both *newKey*'s and *oldKey*'s parents by *mflag*. For the compress transition, it uses a *cflag* operation to flag the parent if necessary.

- **Substitute** (line 122). It uses a CAS to change the existing node with a given node stored in the *op*. It then restores the parent's state to *Clean* by *unflag*.
- **Move** (line 129). It flags *newKey*'s and *oldKey*'s parents. Then, it replaces *oldKey*'s terminal with an *Empty* node and *newKey*'s terminal with a new node. Finally, it *unflag* their parents' *op* to *Clean*.
- **Compress** (line 126). It erases the parent from the tree. Different from other states, the parent with a *Compress op* cannot be set to *Clean* by *unflag*.

```
118  // iflag, rflag, cflag, iflag and unflag
119  bool helpFlag(Internal node, Operation oldOp, Operation newOp) {
120    return CAS(node.op, oldOp, newOp);
121  }

122  void helpSubstitute(Substitute op) {
123    helpReplace(op.parent, op.oldChild, op.newNode);
124    helpFlag(op.parent, op, new Clean());
125  }

126  bool helpCompress(Compress op) {
127    return helpReplace(op.grandparent, op.parent, Empty<V>());
128  }

129  bool helpMove(Move op) {
130    if (op.iFirst) helpFlag(op.rParent, op.oldROp, op);
131    else helpFlag(op.iParent, op.oldIOp, op);
132    pOp := op.iFirst ? op.rParent.op: op.iParent.op;
133    if (pOp == op) { // all flags have been done
134      op.allFlag := true;
135      op.oldRChild.op := op; // atomic set
136      if (op.oldRChild == op.oldRChild) {
137        helpReplace(op.rParent, op.oldRChild, op.newIChild);
138      } else {
139        helpReplace(op.iParent, op.oldIChild, op.newIChild);
140        helpReplace(op.rParent, op.oldRChild, Empty<V>());
141      }
142    } else help(pOp);
143    if (op.iFirst) {
144      if (op.allFlag) helpFlag(op.rParent, op, Clean());
145      if (op.rParent != op.iParent) helpFlag(op.iParent, op, Clean());
146    } else {
147      if (op.allFlag) helpFlag(op.iParent, op, Clean());
148      if (op.rParent != op.iParent) helpFlag(op.rParent, op, Clean());
149    }
150    return op.allFlag;
151  }

152  void help(Operation op) {
153    if (op.class == Compress) helpCompress(op);
154    else if (op.class == Substitute) helpSubstitute(op);
155    else if (op.class == Move) helpMove(op);
156  }
```

Fig. 11: quadboost state transition functions

4.3 Concurrent Algorithms

We describe contain, insert, remove, and move operations in this section. The contain operation starts from the root and invokes the find function (line 160) to reach a terminal node. It does not modify the quadtree during the traversal.

The insert and remove operations modify a node in the tree, following such a paradigm:

1) Locate a terminal node by the find function (line 166 and line 184) and examine its parent's *op*.

```
158  bool contain(double keyX, double keyY) {
159    rec := new Record(Clean(), null, root, Stack<Node>());
160    find(rec, keyX, keyY);
161    if (inTree(rec.l, keyX, keyY) and !moved(rec.l)) return true;
162    return false;
163  }

164  bool insert(double keyX, double keyY, V value) {
165    rec := new Record(Clean(), null, root, Stack<Node>());
166    find(rec, keyX, keyY);
167    while (true) {
168      if (inTree(rec.l, keyX, keyY) and !moved(rec.l)) return false;
169      if (rec.pOp.class == Clean) {
170        newNode := createNode(rec.p, rec.l, keyX, keyY, value);
171        op := new Substitute(rec.p, rec.l, newNode);
172        if (helpFlag(rec.p, rec.pOp, op)) {
173          helpSubstitute(op);
174          return true;
175        } else rec.pOp := p.op;
176      }
177      help(rec.pOp);
178      continueFind(rec, keyX, keyY);
179      find(rec, keyX, keyY);
180    }
181  }

182  bool remove(double keyX, double keyY, V value) {
183    rec := new Record(Clean(), null, root, Stack<Node>());
184    find(rec, keyX, keyY);
185    while (true) {
186      if (!inTree(rec.l, keyX, keyY) or moved(rec.l)) return false;
187      if (rec.pOp.class == Clean) {
188        op := new Substitute(rec.p, rec.l, Empty<V>());
189        if (helpFlag(rec.p, rec.pOp, op) {
190          helpSubstitute(op);
191          compress(rec);
192          return true;
193        } else rec.pOp := p.op;
194      }
195      help(rec.pOp);
196      continueFind(rec, keyX, keyY);
197      find(rec, keyX, keyY);
198    }
199  }

200  void continueFind(Record rec) {
201    if (rec.pOp.class == Compress) { {
202      while (!rec.path.isEmpty()) {
203        rec.l := rec.path.pop();
204        rec.pOp := rec.l.op;
205        if (rec.pOp.class == Compress) helpCompress(rec.pOp);
206        else break;
207      }
208    } else {
209      rec.l := rec.p;
210    }
211  }

212  void find(Record rec, double keyX, double keyY)
213    while (rec.l.class == Internal) {
214      rec.path.push(rec.l);
215      rec.pOp := rec.l.op;
216      rec.l := getQuadrant(rec.l, keyX, keyY);
217    }
218    rec.p := rec.path.pop();
219  }

220  void compress(Record rec) {
221    while (true) {
222      rec.pOp := rec.p.op;
223      if (rec.pOp.class == Clean) {
224        gp := rec.path.pop();
225        if (gp == root) return;
226        op := new Compress(gp, rec.p);
227        // checkEmpty: whether all children of p are Empty
228        if (!checkEmpty(rec.p) or !helpFlag(rec.p, rec.pOp, op))
229          return;
230        else helpCompress(op);
231        rec.p := gp;
232      } else return;
233    }
234  }

235  bool moved(Node l) {
236    // hasChild: whether oldIChild is a child of iParent
237    return l.class == Leaf and l.op != null and
            !hasChild(l.op.iParent, l.op.oldIChild);
238  }
```

Fig. 12: quadboost contain, insert, and remove operations

2) If the parent's *op* is *Clean* and successfully changed by *iflag* or *rflag*, use the helpReplace function (line 123) to change the terminal node and return true.
3) Otherwise, help the parent's *op* finish (line 177 and line 195), pop nodes with a *Compress op* from *path* (line 178 and line 196), and return to step 1.

An extra step in the remove operation is the compress function at line 191. The remove operation performs the compress function before it returns true. To ensure the correctness, the compress function should examine three conditions before compressing the parent node. First, because the remove operation must return true, we do not compress nor help if the state of the parent is not *Clean* (line 223). Second, we check if the grandparent node (*gp*) is the root (line 225) as we maintain two layers of dummy *Internal* nodes. At last, we check whether four children of a parent node are all *Empty* (line 229).

The contain, insert and remove operations check whether the key of a terminal node is in the tree and whether the node is moved at line 168. We will show the reasons why we should check a node's *moved* status in the next paragraphs.

Figures 13 presents the algorithm of the move operation. It begins by searching old key and new key separately (line 242 and line 243). It checks two parents' *op*s (*iRec.pOp* and *rRec.pOp*) before flag operations. If neither of them is *Clean* (line 250-251), or *rRec.p* and *iRec.p* are the same but their *op*s are different (line 252), it will help *op*s finish (line 273-274). Otherwise, it creates a new node for inserting and an *op* to hold essential information for a CAS. There are two specific cases. If two terminal nodes share a common parent, we directly call the helpMove function at line 287. Or else, to avoid live locks, we flag two nodes in a specific order. The quadboost algorithm adopts the getSpatialOrder function at line 256 to compare *iRec.p* and *rRec.p* in the following order: $x \rightarrow y \rightarrow w$, where x, y, and w are the fields of an *Internal* node. We prove that this method produces a unique order among all *Internal* nodes in a quadtree in appendix A.

Next we explain how the helpMove function coordinates between threads. It flags a parent and checks whether both parents are successfully flagged at line 130-132. As illustrated in the previous paragraph, the flag order is determined by two parents' space infomation. *allFlag* is set to true (line 134) only if both parents are flagged, and it implies threads that detect *pOp=op* will help it finish. Note that we assign the *op* to *oldRChild* at line 135 to let other threads aware whether the CAS on *iParent* has finished. If the CAS has finished, threads that call *moved(oldRChild)* will return true simultaneously, which explains why we should call the moved function to check out the status of a node. If two terminal nodes are the same, we could combine two helpReplace functions into a single one. If not, we should remove old key's terminal and replace new key's terminal in order. In the end, we reset the parents' *op* to *Clean* in the reverse order (line 143-149).

```
239 bool move(double oldKeyX, double oldKeyY, double newKeyX, double
        newKeyY) {
240   rRec := new Record(Clean(), null, root, Stack<Node>());
241   iRec := new Record(Clean(), null, root, Stack<Node>());
242   find(rRec, oldKeyX, oldKeyY);
243   find(iRec, newKeyX, newKeyY);
244   while (true) {
245     if (!inTree(rRec.l, oldKeyX, oldKeyY) || moved(rRec.l))
246       return false;
247     if (inTree(iRec.l, newKeyX, newKeyY) and !moved(iRec.l))
248       return false;
249     cFail := iFail := rFail := false;
250     if (rRec.pOp.class != Clean) rFail := true;
251     if (iRec.pOp.class != Clean) iFail := true;
252     if (iRec.pOp != rRec.pOp and iRec.p == rRec.p) cFail = true;
253     if (!iFail and !rFail and !cFail) {
254       newNode := createNode(iRec.l, iRec.p, newKeyX, newKeyY,
              rRec.l.value);
255       // getSpatialOrder: Compare x, y, w fields
256       iFirst := getSpatialOrder(iRec.p, rRec.p);
257       op := new Move(iRec.p, rRec.p, iRec.l, rRec.l, newNode,
              iRec.op, rRec.op, iFirst);
258       if (rRec.p != iRec.p) { // two parents are different
259         hasFlag := iFirst ? helpFlag(iRec.p, iRec.pOp, op) :
                helpFlag(rRec.p, rRec.pOp, op);
260         if (hasFlag) {
261           if (helpMove(op)) {
262             compress(rRec);
263             return true;
264           }
265         } else { // one of two flags fail
266           if (iFirst) {iRec.pOp := iRec.p.op;}
267           else {rRec.pOp := rRec.p.op;}
268         }
269       } else { // two parents are the same
270         if (helpMove(op)) return true;
271       }
272     }
273     help(rRec.pOp);
274     help(iRec.pOp);
275     continueFind(rRec);
276     continueFind(iRec);
277     find(rRec, oldKeyX, oldKeyY);
278     find(iRec, newKeyX, newKeyY);
279   }
280 }
```

Fig. 13: quadboost move operation

4.4 LCA Optimization

Tree-based structures share a property that two nodes in a tree have a common path starting from the root, and the lowest node in the path is called the lowest common ancestor (LCA). Based on the observation, our LCA-based move operation is defined to find two different terminals that share a common path, remove a node with old key, and insert a node with new key.

Figure 14 shows the LCA-based move operation, which has two advantages over the move operation in Figure 13: (1) The LCA-based move operation begins by calling the findCommon function (line 283) that combines searches for two nodes together, reducing duplicated traversal for nodes in the common path. We use two paths to record nodes: the shared path and the remove path are pushed into *rRec.path*; the insert path is pushed into *iRec.path*. If the old key and the new key are not in the same direction (line 304), it terminates the common traversal and searches for individual keys separately (line 309-310). (2) It detects where the helpFlag function fails and sets different bool variables–*rFail* for *rRec.pOp*, *iFail* for *iRec.pOp*, and *cFail* for the helpFlag function on the LCA node. If *rFail* or *iFail* is true, and *cFail* is false (line 314 and line 320), only one *continueFind* is invoked to update *rRec* and *iRec* accordingly.

```
281 bool move(double oldKeyX, double oldKeyY, double newKeyX, double
          newKeyY) {
282   ... // Line 240-241
283   findCommon(oldKeyX, oldKeyY, newKeyX, newKeyY);
284   while (true) {
285     ... // Line 245-259
286     // two parents are different and a flag succeeds
287     if (helpMove(op)) {
288       ... // Line 262-263
289     } else iFail = rFail = true;
290     ... // Line 265
291     // two parents are different and flag operations fail
292     if (iFirst) {iRec.pOp := iRec.p.op; iFail := true;}
293     else {rRec.pOp := rRec.p.op; rFail := true;}
294     ... // Line 268-269
295     // two parents are the same
296     if (helpMove(op)) return true;
297     else cFail := true;
298     ... // Line 271-272
299     continueFindCommon(cFail, iFail, rFail, rRec, iRec, oldKeyX,
               oldKeyY, newKeyX, newKeyY);
300   }
301 }

302 void findCommon(Record rRec, double oldKeyX, double oldKeyY, Record
          iRec, double newKeyX, double newKeyY) {
303   // sameDirection: whether two keys in the same subarea of rRec.l
304   while (rRec.l.class == Internal and sameDirection(oldKeyX,
          oldKeyY, newKeyX, newKeyY, rRec.l)) {
305     rRec.path.push(rRec.l); // rRec.l is a common ancestor
306     rRec.l := getQuadrant(rRec.l, oldKeyX, oldKeyY);
307   }
308   rRec.lca := iRec.l := rRec.l := rRec.path.pop();
309   find(rRec, oldKeyX, oldKeyY); // find old key's terminal
310   find(iRec, newKeyX, newKeyY); // find new key's terminal
311 }

312 void continueFindCommon(bool iFail, bool rFail, bool cFail, Record
          iRec, Record rRec, double oldKeyX, double oldKeyY, double
          newKeyX, double newKeyY) {
313   if (cFail) { help(rRec.pOp); help(iRec.pOp); } // same parent
314   if (rFail and !cFail) {
315     help(rRec.pOp);
316     continueFind(rRec);
317     if (rRec.path.size() <= indexOf(rRec.lca)) cFail := true;
318     if (!cFail) find(rRec, oldKeyX, oldKeyY);
319   }
320   if (iFail and !cFail) {
321     help(iRec.pOp);
322     continueFind(iRec);
323     if (iRec.pOp.class == Compress) { // iRec.path is empty
324       // jump to LCA, fetech a subpath from root to LCA
325       rRec := <iRec.pOp, null, rRec.lca, rRec.path(root, rRec.lca)>;
326       cFail := true;
327     }
328     if (!cFail) find(iRec, newKeyX, newKeyY);
329   }
330   if (cFail) {
331     iRec.path.clear();
332     continueFind(rRec);
333     findCommon(rRec, iRec, oldKeyX, oldKeyY, newKeyX, newKeyY);
334   }
335 }
```

Fig. 14: quadboost LCA move operation

If *cFail* is true (line 330), it clears nodes in *iRec.path* (line 331) and points the head of *rRec.path* to the parent of the LCA node (line 325), avoiding accessing nodes between parents and the LCA node.

In practice, we notice that pushing the whole path into a stack during the traversal is highly expensive, especially for the move operation where two paths should be maintained. When detecting a failed flag operation, we only have to restart from the parent of a terminal node because we do not change *Internal* nodes unless the compress function erases them from a quadtree. And if the parents are erased, we can first jump to the LCA node and then the root node. Thus, to reduce the pushing cost, we can constrain the length of *path* in *iRec* and *rRec*.

5 Proof Sketch

In this section, we prove that quadboost is both linearizable and non-blocking, and we give a complete proof in appendix A.

In Section 5.1, we start by presenting some invariants in the quadboost algorithm. Then, we demonstrate three categories of successful CAS transitions in Section 5.2 according to Figure 10 and derive their properties. Using the properties, we prove that a quadtree's structure is maintained during concurrent modifications in Section 5.3. We also show that quadboost is linearizable because the insert, remove, move, and contain operations can be correctly ordered by their linearization points in Section 5.4. Finally, we prove that quadboost is non-blocking in Section 5.5.

5.1 Invariants

Invariant 1. *The root node is never changed.*

According to different *Operation* structures, only *Compress* leads to *Internal* nodes be substituted. But a *Compress op* will not be created if *op.gp* is the root (line 225).

Invariant 2. *Only an Internal node with all children Empty could be attached a Compress op.*

We check children of an *Internal* node before flagging a *Compress op* at line 229.

Invariant 3. *Two different nodes in a quadtree share a common path that starts from the root.*

Because the root is not changed (Invariant 1), nodes that are descendants of root must share a common path.

5.2 Transitions

We use *flag* to denote helpFlag function calls that use *Clean oldOp*, *unflag* to denote helpFlag function calls that use *Clean newOp*, *replace* to denote helpReplace function calls. To prove the correctness of three CAS transitions in Figure 10, we show that they follow specific orders. $flag_i$ attaches op_i on a node, and $replace_i$ and $unflag_i$ use op_i and take effect after $flag_i$. We say $flag_i$, $replace_i$, and $unflag_i$ belong to the same *op*. If mulitple *flag* and *replace* operations belong to the same *op* occur, we let $replace_i$ be a sequence of replace operations: $replace_i^0$, $replace_i^1$, ..., $replace_i^k$. The similar notation is used for $flag_i^k$ and $unflag_i^k$ operating on $op_i.p^k$ (parent) or $op_i.gp^k$ (grandparent).

Three kinds of CAS transitions: (1) $flag \rightarrow replace \rightarrow unflag$, (2) $flag \rightarrow replace$, (3) $flag \rightarrow unflag$ have following properties:

Lemma 1. *Transitions (1) and (2) that contain successful replace operations has following properties:*

1) $replace_i^k$ *is the first successful replace operation on* $op_i.p^k$ *or* $op_i.gp^k$ *after time instant* T_{i1} *when* $op_i.oldChild^k$ *is read from the tree.*
2) $replace_i^k$ *that belongs to* op_i *is the first successful replace operation on* $op_i.p^k$ *or* $op_i.gp^k$.

3) *No unflag operation succeeds before* $replace_i$.
4) *For Substitute and Move* op_i, $unflag_i^k$ *that belongs to* op_i *is the first successful unflag operation on* $op_i.p^k$ *or* $op_i.gp^k$ *after* $flag_i^k$; *for Compress* op_i, *no unflag operation succeeds after* $creplace_i$.
5) *The first replace operation that belongs to* op_i *must succeed on* $op_i.p^k$ *or* $op_i.gp^k$.

Lemma 2. *Transition (3) has following properties:*

1) $unflag_i$ *that belongs to Move* op_i *is the first successful unflag operation on* $op_i.rParent$.
2) *The first flag operation on* $op_i.iParent$ *must fail, and no later flag operation will succeed.*
3) $op_i.iParent$ *and* $op_i.rParent$ *are different.*
4) $replace_i$ *is empty.*

5.3 Quadtree properties

In this section, we show that a quadtree's properties are maintained during concurrent modifications.

$snapshot_{T_i}$ is defined to be the keys and structures of a quadtree at time instant T_i. We use *active* and *inactive* sets to represent different kinds of nodes. We say a node is *active* in $snapshot_{T_i}$ at T_i if it is located in the correct position of the quadtree in $snapshot_{T_i}$; otherwise, the node is *inactive*. We use $path^k$ to denote *rec.path* (line 214) updated by the $find(keys^k)$, which contains only *Internal* nodes. We say $path^k$ is *active* if all nodes from the root to every node $n \in path^k$ are active. We use l^k to denote $rec.l$ that is read at line 216 in $find(keys^k)$.

According to Lemma 1, *replace* operations always follow *flag* operations and read the *op* it attaches. Thus, we denote different *replace* operations by their *op* types. *ireplace* and *rreplace* represent the helpReplace function for a *Substitute op* created by insert and remove operations respectively, *mreplace* represents the helpReplace function for a *Move op*, and *creplace* represents the helpReplace function for a *Compress op*.

Lemma 3. *In* $path^k$, *if* n_t *is active, then* $n_0, ..., n_{t-1}$ *that are pushed before* n_t *are active.*

Proof. We prove the lemma by contradiction. Assume that node n in $n_0, ..., n_{t-1}$ is inactive, its *op* should be set to *Compress* before T_i when n_t is read. Then, all children of n are *Empty* (Invariant 2) and cannot be changed before accessing n^t. Thus, n^t is *Empty* and will not be pushed into $path^k$. It derives a contradiction. □

Lemma 4. *If two Leaf or Empty nodes' share an LCA node at* T_i *which is active at* $T_{i1} \succ T_i$, *the node is still the LCA at* T_{i1}.

Proof. We prove the lemma by contradiction. Suppose that the node is not the LCA of two *active* nodes at T_{i1}, there should be some *replace* operations succeed on the ancestors of two nodes at T_{i2}, $T_{i1} \succ T_{i2} \succ T_i$.

Because *ireplace*, *rreplace*, *mreplace* operate on *Leaf* and *Empty* nodes, their ancestors will not be changed. The only possible scenario is that *creplace* removes it from the quadtree. But if *creplace* succeeds, all of its children are removed, which contradicts our assumption that two nodes are *active* at T_{i1}. □

Lemma 5. *For* $find(keys^k)$ *that returns at* T_i, *there exists* $snapshot_{T_{i1}}, T_{i1} \prec T_i$ *before reading* l^k *such that* $path^k$ *and* l^k *are active.*

Proof. We prove the lemma by induction. Because l^k is pushed into $path^k$ in the find function, we consider the pushing sequence by $find(keys^k)$ as $l_0^k, l_1^k, ..., l_{n-1}^k$, and $l^k = l_n^k$.

Base case ($i = 0$): We prove that before reading l_0^k, l_0^k and $path^k$ are active.

Case 1: $find(keys^k)$ is invoked at line 166, line 188, line 243, or line 242. l_0^k is the root node that will never be changed (Invariant 1), and $path^k$ is empty. The claim is true.

Case 2: Or else, l_0^k is set to an active *Internal* node because its *op* is not *Compress* at $T_{i1} \prec T_i$ (line 201 and line 323). As l_0^k is active, nodes above it are also active so that $path^k$ is active (Lemma 3). Futher, l_0^k is the correct ancestor of $keys^k$, because it is either read from $path^k$, or the active LCA node (Lemma 4). The claim is true.

Induction step ($i > 0$): Assume the claim holds for $i - 1$, we have to prove it holds before reading l_i^k.

Case 1: If no replace operation succeeds on l_{i-1}^k after reading l_{i-1}^k, it is obvious that we have active $path^k$ and l^k in the same snapshot before reading l_{i-1}^k.

Case 2: If some replace operations succeed on l_{i-1}^k after reading l_i^k, we have active $path^k$ and l_i^k in the same snapshot before l_{i-1}^k. As replace operations happen on l_{i-1}^k, it is an active node that contains an *op* other than *Compress* (Invariant 2). So l_i^k is also reachable from the root in the same snapshot.

Case 3: If some replace operations succeed on l_{i-1}^k after reading l_{i-1}^k and before l_i^k. We have active $path^k$ and l_i^k in the snapshot after the replace operation and before reading l_{i-1}^k. □

We prove a quadtree's properties by Lemma 3-5.

Theorem 1. *A quadtree has two categories of properties maintained in every snapshot:*

1) *Two layers of dummy Internal nodes are never changed.*
2) *An Internal node n has four children, located in the direction* $d \in \{nw, ne, sw, se\}$ *respectively, according to their* $\langle x, y, w, h \rangle$, *or* $\langle keyX, keyY \rangle$.
   ***Internal**: For Internal nodes that reside on four directions:*
   - $n.nw.x = n.x$, $n.nw.y = n.y$;
   - $n.ne.x = n.x + w/2$, $n.ne.y = n.y$;
   - $n.sw.x = n.x$, $n.sw.y = n.y + n.h/2$;
   - $n.se.x = n.x + w/2$, $n.se.y = n.y + h/2$,

   and all children have their $w' = n.w/2$, $h' = n.h/2$.
   ***Leaf**: For Leaf nodes that reside on four directions:*
   - $n.x \le n.nw.keyX < n.x + n.w/2$, $n.y \le n.nw.keyY < n.y + n.h/2$;
   - $n.x + n.w/2 \le n.ne.keyX < n.x + n.w$, $n.y \le n.ne.keyY < n.y + n.h/2$;
   - $n.x \le n.sw.keyX < n.x + n.w/2$, $n.y + n.h/2 \le n.sw.keyY < n.y + n.h$;
   - $n.x + n.w/2 \le n.se.keyX < n.x + n.w$, $n.y + n.h/2 \le n.se.keyY < n.y + n.h$.

Proof. We prove the above two categories separately by discussing different *replace* operations.

1) *ireplace*, *rrepalce*, and *mreplace*: They swing *Leaf* or *Empty* nodes so that dummy *Internal* nodes are not changed.
   creplace: Invariant 1 points out that the root will never be attached on a *Compress op*, so dummy nodes will not be changed.
2) *creplace*: It replaces an *Internal* node with an *Empty* node. Because the *Internal* node has been attached on a *Compress op* before *creplace*, all of its children are *Empty* (Invariant 2). Thus, *creplace* does not violate a quadtree's structural properties.
   rreplace and *mreplace* that attach an *Empty* node: They replace *Leaf* nodes with *Empty* nodes and therefore do not affect a quadtree's properties.
   ireplace and *mreplace* that attach a subtree: We have to check that whether the subtree with *newKey* and *oldChild.key* is active and in the correct position with regard to $op_i.p^k$ before and after a successful replace operation.
   By Lemma 5, l^k that contains $keys^k$ for $find(keys^k)$ is active in a snapshot before the function returns. After the replacement, p^k is reachable from the root since it's *op* is not *Compress*. Hence, nodes above *p* are also active. Since new subtree is in the same position as l^k before the replacement, it is active and correctly located after a successful replace operation.

□

5.4 Linearizability

In this Section, we define linearization points for the quadboost algorithm. If an algorithm is linearizable, its result history could be ordered equivalently as a sequential one by each operation's linearization point. As the compress operation is included in the move and remove operations that return true, we do not define it individually.

Theorem 2. *quadboost is linearizable.*

Proof. We list out the **linearization points** of basic operations as follows:

1) For *insert(key)*, *remove(key)*, or *move(oldKey, newKey)* that returns true, *replace* that belongs to the op created by an operation succeeds before it returns. Thus, the linearization point is at the first successful replace operation (line 123 or line 139).
2) For *contain(key)*, it does not create an *op* before it returns. We linearize it at a snapshot based on *rec.l* is moved or not.
   - If *contain(key)* returns true, it indicates that *rec.l* is a active *Leaf* node that contains *key* and not *moved*. Hence, we can linearize it at a snapshot that checks *rec.l* is not *moved* (line 161).
   - If *contain(key)* returns false, it indicates that *rec.l* is a active *Empty* node or a *Leaf* node that is *moved* on the path for *key*. For the first case, we can linearize it at a snapshot that reads *rec.l* (line 216). For the second case, we can linearize it at a snapshot that checks *rec.l* is *moved* (line 161).
3) For *insert(key)*, *remove(key)*, or *move(oldKey, newKey)* that returns false, no *replace* that belongs to the op created by the operation succeeds before it returns (Lemma 17). Thus, we can linearize them similarly as *contain(key)*.
   - *insert(key)*: We linearize it at a snapshot that checks *rec.l* is not *moved* (line 168).
   - *remove(key)*: If *rec.l* is moved, we linearize it at a snapshot that checks *rec.l* (line 186). Otherwise, we linearize it at a snapshot that reads *rec.l* (line 216).
   - *move(oldKey, newKey)*: It returns false at two positions.
     Line 246: If *rRec.l* is moved, we linearize it at a snapshot that checks *rRec.l* (line 247). Otherwise, we linearize it at a snapshot that reads *rRec.l* (line 216).
     Line 248: We linearize it at a snapshot that checks *iRec.l* is not *moved* (line 245).

□

5.5 Non-blocking

An algorithm is non-blocking if the system as a whole is making progress even if some threads are starving. We have to prove two parts: (1) Quadboost is terminable. We use a set of lemmas to show that every while loop in our algorithm must terminate, starting from the inner most loops to the outer loops. (2) The quadtree must be changed between each iteration of the outermost while loops (line 167, line 185, and line 284).

In the following lemmas, we prove that inner loops must terminate.

Lemma 6. *Every call to the find and findCommon functions must terminate.*

Lemma 7. *Every call to the compress function must terminate.*

Lemma 8. *Every call to the continueFind function must terminate.*

In the next lemmas, we prove that the outermost loops at line 167, line 185, and line 284 must terminate. We first prove that they will terminate apart from calling the help function. Then, we prove that the help function will terminate.

Lemma 9. *There is an unique spatial order among Internal nodes in a quadtree in every snapshot.*

Proof. Each *Internal* node contains spatial information–$\langle x, y, w, h\rangle$. In our quadtree, we only consider square partitions on two-dimensional space. Therefore, w is always equal to h. At line 256, the getSpatialOrder function compares *ip* with *rp* by the order: $x \rightarrow y \rightarrow w$.

We prove the lemma by contradiction. Assume there are two different *Internal* nodes with the same $\langle x, y, w\rangle$, they represent the same square starting with left coroner $\langle x, y\rangle$ with width w and height w. By Theorem 1, a quadtree's properties maintain in every snapshot. There cannot be two squares with the same left corner and w, which contradicts to our hypothesis. Hence, in $snapshpt_{T_i}$, *Internal* nodes consist of unique $\langle x, y, w\rangle$ tuples that can be ordered correctly. □

Lemma 10. 1) *There are finite number of successful $flag \rightarrow replace \rightarrow unflag$ transitions.*
2) *There are finite number of successful $flag \rightarrow replace$ transitions.*
3) *There are finite number of successful $flag \rightarrow unflag$ transitions.*

Proof. 1) There is a unique *op* that leads to every $flag \rightarrow replace \rightarrow unflag$ transition. Hence, for remove, insert, and move operations, there are finite number of $flag \rightarrow replace \rightarrow unflag$ transitions.
2) The compress function creates *Compress* ops and is called by remove and move operations that return true. As Lemma 7 shows that the compress function is terminable, the number of *Compress* ops created is finite. Thus, there are a finite number of $flag \rightarrow replace$ transitions.
3) The $flag \rightarrow unflag$ transition executes only in the move function where *op.iParent* cannot be flagged, if *op.iFirst* is true. The case that *op.iFirst* is false is symmetrical.
We prove this part by contradiction. If $flag \rightarrow unflag$ executes infinitely, there are two move operations $move_{i1}$ and $move_{i2}$ depend on each other. In other words, $move_{i1}$ which flags rp_i but fails on ip_i, and $move_{i2}$ which flags ip_i but fails on rp_i infinitely. Thus, we order rp_i and ip_i as $rp_i > ip_i$ by $move_{i1}$'s order, and $ip_i > rp_i$ by $move_{i2}$'s order. But by Lemma 9, all *Internal* nodes in a snapshot can be ordered uniquely. Therefore it derives a contradiction.
Hence, there are finite number of $flag \rightarrow unflag$ transitions.

□

Lemma 11. *Every call to the help function must terminate.*

Proof. The help function invokes *ireplace*, *rreplace*, *creplace* and *mreplace* based on the type of the *op*. If *creplace*, *ireplace*, or *rreplace* is called, the help function will return instantly. For *mreplace*, it might invoke the help function at line 142. We prove it will terminate by contradiction.

Assume there is a calling sequence: $help \rightarrow helpMove \rightarrow ... \rightarrow help$. If it is not terminable, a ring exists in the sequence, such that the last helpMove function will call the first help function.

We consider the dependency among all $helpMove_i$. If $helpMove_{i1}$ which fails on $op_{i1}.p^1$ which has been flagged by $helpMove_{i2}$, it will help $helpMove_{i2}$'s replace operations. We link a directed edge from $helpMove_{i1}$ to $helpMove_{i2}$ by their spatial order on $op_{i1}.p^1$ (Lemma 9), because $op_{i1}.p^0 \rightarrow op_{i1}.p^1 \rightarrow op_{i2}.p^0$. In this way, the last helpMove has a directed edge linking to the first helpMove. However, according to Lemma 10 (3), the last node in the dependency graph must have no out-going edge. Therefore, the last helpMove will set its *op* on *op.p*^{1} successfully. Then, after erasing the former head node from the graph, there will be other nodes that do not have an out-going edge. Finally, all dependency edges are erased. Hence, the invocation sequence is terminable, which contradicts to the hypothesis. □

Next, we have to show every pending *op* can be processed. We prove that between each iteration of the while loop at line 167, line 185, and line 284, the quadtree is changed by successful CAS transitions. We label $find(keys)$ outside the while loop as $find_0(keys)$ and later calls as $find_i(keys), i \geq 1$ for each iteration.

Lemma 12. *If rec.pOp is read at T_{i1} in $find_i(keys)$, and $help_{i+1}$ is returned at $T_{i2} \succ T_{i1}$, the quadtree is changed between T_{i1} and T_{i2} by finishing successful CAS transitions.*

Proof. There are three possible scenarios that $help_{i+1}$ is called after $find_i(keys)$:

Case 1: *rec.pOp* is not *Clean* at line 169, line 187, line 251, and line 250. It indicates that some CAS transitions has started by flagging *rec.p* at T_{i2}. Lemma 11 points out that the help function terminates in finishing replace operations.

Case 2: The helpFlag function fails at line 172, line 189, line 271, and line 259. It indicates that either $flag \rightarrow replace \rightarrow unflag$ or $flag \rightarrow unflag$ happens. For the former case, the quadtree is changed before T_{i2}. For the latter case, the quadtree is changed before T_{i2} if the failed *p.op* is *Clean*. Or else, in the $flag \rightarrow unflag$ transition, it helps to change the quadtree by calling the help function that helps finish replace operations (Lemma 11) before T_{i2}.

Case 3: *rRec.pOp* is not the same as *iRec.pOp*, and *rRec.p* equals to *iRec.p* at line 252. It indicates that *lca.op* is changed between $find_i(oldKey)$ and $find_i(newKey)$. If it is changed from *Clean* to *Clean*, we use the same proof as Case 2. Otherwise, we adopt Case 1 to prove it. □

Theorem 3. *quadboost is non-blocking.*

Proof. Lemma 6-11 show that the quadboost algorithm is terminable, and Lemma 12 indicates that the system as a whole will make progress even if some threads starve. Thus the quadboost algorithm is non-blocking. □

6 Evaluation

The experiments were setup on a machine with 64GB main memory and two 2.6GHZ Intel(R) Xeon(R) 8-core E5-2670 processors with hyper-threading enabled, rendering 32 hardware threads in total. RedHat Enterprise Server 6.3 with Linux core 2.6.32 was installed, and all experiments were ran under Sun Java SE Runtime Environment (build 1.8.0_65). To avoid significant run-time garbage collection cost, we set the initial heap size to 6GB.

For each experiment, we ran eight 1-second cases, where the first 3 cases were performed to warm up JVM, and the median of the last 5 cases was used as the real performance. Before the start of each case, we inserted half keys from the key set into a quadtree to guarantee that the insert and the remove operation have equal success opportunity initially.

We applied uniformly distributed key sets that contain two-dimensional points within a square, where $range$ is used to denote the length of the square. Thus, points are located inside a $range * range$ square. In our experiments, we used two different key sets: 10^2 keys to measure the performance under high contention, and 10^6 keys to measure the performance under low contention. For simplicity, we let the $range$ of the first category experiment be 10, rendering $1 - 10^2$ consecutive keys for one-dimensional structure. For the second category experiment, we let the $range$ be 1000, generating $1 - 10^6$ consecutive keys. Given T threads, the probability C that more than one thread contends on a node can be presented as:

$$C = 1 - \frac{P_T^{range}}{range^T} \tag{1}$$

TABLE 1: Concurrent tree techniques

Type	Decoupling	Compression	Continuous find	Move
QC	-	-	-	-
QB-0	-	-	-	✓
QB-0-D	✓	✓	-	✓
QB-1	✓	✓	✓	✓
QB-N	✓	✓	✓	✓
K-ARY	-	-	-	-
CTRIE	✓	✓	-	-
PATRICIA	-	-	-	✓

The experiments contain two parts: (1) In Section 6.1, we evaluate throughput metrics. We present both CAS quadtree (QC) and quadboost (QB), where QB-1 represents an implementation that the size of path ($record.path$) is limited to one, and QB-N indicates an implementation using path without limitation. (2) In Section 6.2, we examine the incremental effects of the optimization strategies proposed in Section 4.1. To determine how quadboost algorithms improve the performance, we devise two algorithms that incrementally use parts of techniques in QB-N:

- QB-0 is an implementation without decoupling, compression, and continuous find. Thus, it does not maintain a path.
- QB-0-D uses decoupling to separate the physical adjustment in the remove operation and move operation based on QB-0.

To the best of our knowledge, a formal concurrent quadtree has not been published yet. Hence, we also compare our quadtrees with three one-dimensional non-blocking trees:

- K-ARY [18] is a non-blocking k-way search tree, where k represents the number of branches maintained by an internal node. We use K-ARY for comparison because its structure is similar to quadtrees when $k = 4$. However, K-ARY does not have a series of internal nodes representing the two-dimensional space hierarchy.
- CTRIE [19] is a concurrent hash trie, where each node can store up to 2^k children. We use $k = 2$ to make a 4-way hash trie that resembles quadtrees. The hash trie also incorporates a compression mechanism to reduce unnecessary nodes. Different from quadtrees, it uses a control node (`INODE` as the paper indicates) to coordinate concurrent updates. Hence, the search depth is longer than quadtrees.
- PATRICIA [20] presents a concurrent patricia trie which adopts Ellen's BST techniques [4]. As the author points out, it can be used as a quadtree by interleaving the bits of x and y. It also supports the move (replace) operation. Unlike our LCA-based move operation, it searches two positions separately without the continuous find mechanism.

All trees are implemented by Java using its `compareAndSet` function for CAS. We use `AtomicReferenceFieldUpdater` to instantiate each field that is changed by CAS (e.g. *op* in *Internal*). Table 1 summarizes concurrent tree techniques proposed by Section 4.1. **Compression** represents whether unnecessary nodes are removed from the tree. **Continuous find** indicates whether it restarts from the nearest parent when CASs fail. **Decoupling** shows whether logical removal is separated from physical adjustment.

6.1 Throughput

Since K-ARY and CTRIE only store one-dimensional keys, we transform a two-dimensional key into a one-dimensional key for comparison. We devise a general formula: $key^1 = key_x^2 * range + key_y^2$. Given a two-dimensional key-key^2, and $range$, we transformed it into a one-dimensional key-key^1. To refrain from trivial transformations by floating numbers, we only considered integer numbers in this section.

Due to the lack of the move operation in QC, we compare throughput with/without the move operation respectively. Figure 15 plots throughput without any move operation for the concurrent algorithms. It is not surprising to observe that QC achieves the highest throughput and speedup in most scenarios. To some extent, QC represents an upper bound of throughput because its remove operation applies a single CAS, leading to less contention than other concurrent algorithms. Both QB-N and QB-1 can achieve comparable throughput when the key set becomes larger, or the insert and remove ratio decreases. For instance, in Figure 15(d), QB-1 and QB-N are only 5% worse than QC. This phenomenon occurs due to: (i) fewer thread interventions results in a lesser number of CAS failures on nodes; and (ii) both algorithms compress nodes and use the continuous find mechanism to reduce the length of traverse path.

As a comparison, CTRIE, K-ARY, and PATRICIA show lower performances with the increasing number of threads. For example, in Figure 15 (c), QB-1 outperforms CTRIE by 49%, PATRICIA by 79%, and K-ARY by 109% at 32 threads. Both K-ARY and PATRICIA flag the grandparent node in the remove operation, which allows less concurrency than CTRIE, QB-1 and QB-N with the decoupling approach shown in Figure 8. QB-N is worse than QB-1 due to its extra cost of recording elements and compressing nodes recursively. As we discuss in the next section, QB-N and QB-1 save a significant number of nodes as shown in Figure 18. It implies that QB-N and QB-1 occupy less memory and result in a shorter path for traversal than QC.

Figure 16b demonstrates that quadboost has an efficient move operation. Using the small key set, where the depth is not a significant impact, Figure 16a shows that QB-1 is more efficient than PATRICIA especially when contention is high. For example, it performs better than PATRICIA by 47% at 32 threads because it adopts the continuous find mechanism to traverse less path and decouples physical adjustment for higher concurrency. However, QB-N is similar to PATRICIA

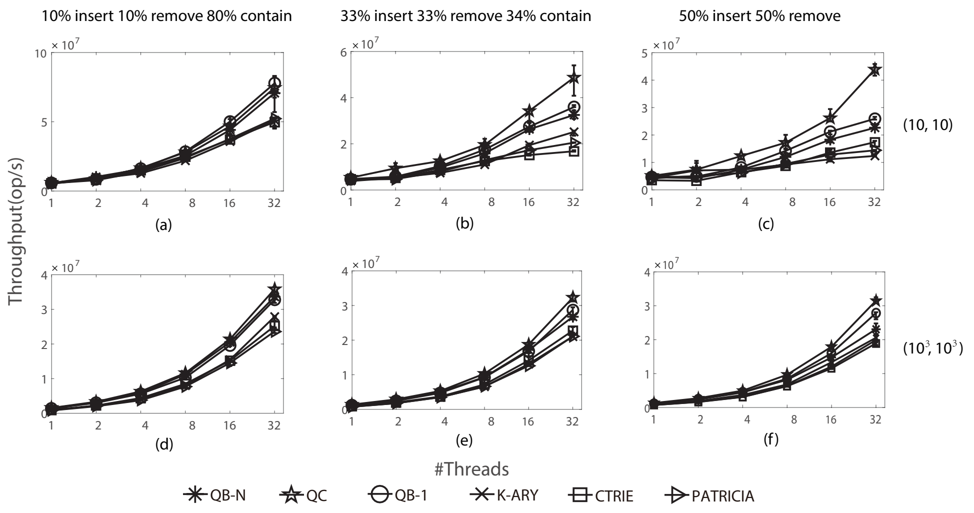
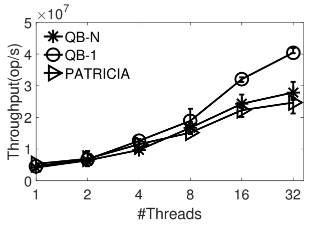


Fig. 15: Throughput of different concurrent trees under both high and low contention


(a) 10^2 keys

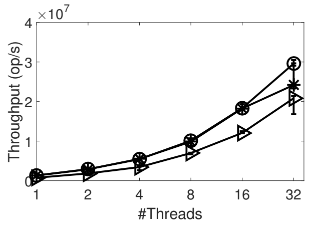
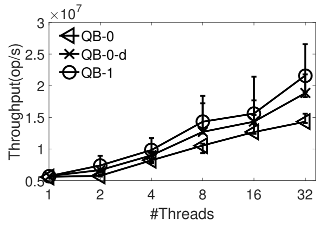

(b) 10^6 keys

Fig. 16: Comparison of the move operation's throughput between quadboost and patricia in both small and large ranges (10% *insert*, 10% *remove*, 80% *move*).


(a) 10% *insert*, 90% *remove*

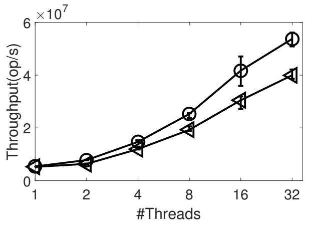
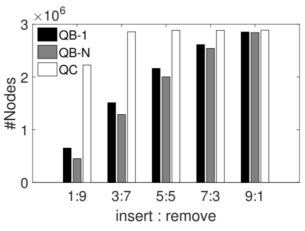

(b) 90% *insert*, 10% *remove*

Fig. 17: Throughput comparison in insert dominated and remove dominated cases (10^2 keys).

since it has to maintain a stack and recursively compress nodes in a quadtree. Figure 16b illustrates that QB-N and QB-1 have a similar throughput for the large key set. QB-1 outperforms PATRICIA by 31% at 32 threads. When the key set is large, the depth becomes a more significant factor due to less contention. Since each *Internal* node in a quadtree maintains four children while PATRICIA maintains two, the depth of PATRICIA is deeper than quadboost. Further, the combination of the LCA node and the continuous find mechanism ensures that QB-1 and QB-N do not need to restart from the root even if flags on two different nodes fail.

6.2 Analysis

$range$ here was set to $2^{32}-1$, and both key_x and key_y could be floating numbers. We used an insert dominated experiment and a remove dominated experiment to demonstrate the effects of different techniques. We used a remove dominated experiment to show the effect of decoupling, where *insert:remove* ratio was 1:9. In the insert dominated experiment, the *insert:remove* ratio was 9:1; hence, there were far more insert operations. Since fewer compress operations were induced, the experiment showed the effect of the continuous find mechanism. Figure 17a illustrates that quadtrees with decoupling exhibit a higher throughput than QB-0, the basic concurrent quadtree. Besides, QB-1 which incorporates the continuous find is more efficient than QB-0-D. Specifically, at 32 threads, QB-1 performs 15% better than QB-0-D and 51% better than QB-0. From Figure 17b, we notice that QB-1 outperforms QB-0 by up to 35%. Therefore, it demonstrates that the continuous find mechanism and the decoupling approach play a significant role in our algorithm.


(a) Nodes

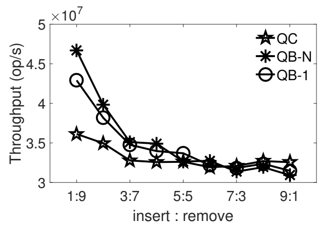

(b) Throughput

Fig. 18: Number of nodes left and throughput under different ratios of insert:remove from 9:1 to 1:9, with fixed 90% contain, under 32 threads and 10^6 keys.

Another advantage of quadboost results from the compression technique, which reduces the search path for each operation and the memory consumption. Figure 18[2] plots the number of nodes left and the throughput of each

2. Unlike previous experiments, we run eight 3-second cases in the experiment to ensure stable amount of modifications

quadtree at different *insert:remove* ratios. As QC only replaces a terminal node with an *Empty* node without compression, it results in the greatest number of nodes in the memory (Figure 18a). In contrast, QB-N and QB-1 compress a quadtree if necessary. When QC contains much more nodes than other quadtrees, the remove operation dominates (the first groups bars to the left) and has three times the amount of nodes of QB-N. The result also indicates that QB-1 contains a similar number of nodes to QB-N despite it only compresses one layer of nodes. Figure 18b illustrates the effectiveness of compression in the face of tremendous contain operations. QB-N outperforms QC by 30% at 9:1 *insert:remove* ratio because QB-N adjusts the quadtree's structure by compression to reduce the length of the search path. With the increment of the insert ratio, QB-N performs similar to QC due to the extra cost of maintaining a stack and the recursive compression. However, QB-1 achieves good balance between QB-N and QC, which compresses one layer of nodes without recording the whole traverse path.

7 Related Works

Because there are limited works related to concurrent quadtrees, we present a roadmap to show the development of state-of-the-art concurrent trees.

Ellen et al. [4] provided the first non-blocking BST by a cooperative method and proved it correct. Brown et al [18] used a similar approach for the concurrent k-ary tree. Shafiei [20] applied the method for the concurrent patricia trie. It also showed how to design a concurrent operation where two pointers need to be changed. Ellen et al. [21] exhibited how to incorporate a stack to reduce the original complexity from O(ch) to O(h + c). The above concurrent trees have an external structure, where only leaf nodes contain actual keys. Our quadboost is a hybrid of these techniques, using the cooperative method for concurrent coordination, changing two different positions with atomicity, and devising a continuous find mechanism to reduce restart cost.

Different from the previously mentioned methods that applied flags on nodes, Natarajan et al. [6] illustrated how to apply flags on edges for a non-blocking external BST. Chatterjee et al. [7] provided a threaded-BST with edge flags and claimed it had a low theoretical complexity. Unlike the above trees that have to flag their edges before removal, our CAS quadtree uses a single CAS in both the insert operation and the remove operation.

The first balanced concurrent BST was proposed by Bronson et al. [16] in which they used an optimistic and relaxed balance method to build an AVL tree. Besa et al. [22] employed a similar method for a red-black tree. Both works were constructed on fine-grained locks and were deadlock free. Based on the special properties of quadtrees, we also decoupled physical adjustment from logical removal and achieved a high throughput.

8 Conclusions

In this paper, we presented a non-blocking quadtree–*quadboost*, which supports concurrent insert, remove, contain, and move operations. We decouple physical updates from the logical removal to improve concurrency, analyze flags on nodes to decide whether to move down or up, and modify two pointers atomically. The experimental results demonstrate that quadboost algorithms are scalable with a variety of workloads and thread counts.

References

[1] M. Moir and N. Shavit, "Concurrent data structures," *Handbook of Data Structures and Applications*, pp. 47–14, 2007.

[2] N. Shavit, "Data structures in the multicore age," *Communications of the ACM*, vol. 54, no. 3, pp. 76–84, 2011.

[3] T. David, R. Guerraoui, and V. Trigonakis, "Asynchronized concurrency: The secret to scaling concurrent search data structures," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2015, pp. 631–644.

[4] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel, "Non-blocking binary search trees," in *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*. ACM, 2010, pp. 131–140.

[5] S. V. Howley and J. Jones, "A non-blocking internal binary search tree," in *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2012, pp. 161–171.

[6] A. Natarajan and N. Mittal, "Fast concurrent lock-free binary search trees," in *ACM SIGPLAN Notices*, vol. 49, no. 8. ACM, 2014, pp. 317–328.

[7] B. Chatterjee, N. Nguyen, and P. Tsigas, "Efficient lock-free binary search trees," in *Proceedings of the 2014 ACM symposium on Principles of distributed computing*. ACM, 2014, pp. 322–331.

[8] V. Ng and T. Kameda, "Concurrent accesses to r-trees," in *Advances in Spatial Databases*. Springer, 1993, pp. 142–161.

[9] J. Chen, Y.-F. Huang, and Y.-H. Chin, "A study of concurrent operations on r-trees," *Information Sciences*, vol. 98, no. 1, pp. 263–300, 1997.

[10] R. Obe and L. Hsu, *PostGIS in action*. Manning Publications Co., 2011.

[11] A. C. Tassio Knop, "Qollide - Quadtrees and Collisions," https://graphics.ethz.ch/~achapiro/gc.html, 2010, [Online; accessed 29-August-2015].

[12] G. J. Sullivan and R. L. Baker, "Efficient quadtree coding of images and video," *Image Processing, IEEE Transactions on*, vol. 3, no. 3, pp. 327–331, 1994.

[13] H. Samet, "The design and analysis of spatial data structures," *Addison-Wesley Reading, MA*, vol. 199, 1990.

[14] D. P. Mehta and S. Sahni, *Handbook of data structures and applications*. CRC Press, 2004.

[15] T. Brown, F. Ellen, and E. Ruppert, "A general technique for non-blocking trees," in *ACM SIGPLAN Notices*, vol. 49, no. 8. ACM, 2014, pp. 329–342.

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

[17] Region Quadtree Operations. http://donar.umiacs.umd.edu/quadtree/regions/regionquad.html, [Online; accessed 20-March-2017].

[18] T. Brown and J. Helga, "Non-blocking k-ary search trees," in *Principles of Distributed Systems*. Springer, 2011, pp. 207–221.

[19] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky, "Concurrent tries with efficient non-blocking snapshots," in *ACM Sigplan Notices*, vol. 47, no. 8. ACM, 2012, pp. 151–160.

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

[21] F. Ellen, P. Fatourou, J. Helga, and E. Ruppert, "The amortized complexity of non-blocking binary search trees," in *Proceedings of the 2014 ACM symposium on Principles of distributed computing*. ACM, 2014, pp. 332–340.

[22] J. Besa and Y. Eterovic, "A concurrent red–black tree," *Journal of Parallel and Distributed Computing*, vol. 73, no. 4, pp. 434–449, 2013.