LOCK FREE MULTIDIMENSIONAL RANGE SEARCH

Aman Aggarwal
2018327
IIIT
Delhi
aman18327@iiitd.ac.in

Prasham Narayan
2018359
IIIT
Delhi
prasham18359@iiitd.ac.in

ABSTRACT

Multi-Dimensional Range Search is a fundamental problem in many application domains dealing with multidimensional data. R-tree is an ideal candidate for the multidimensional range search problem. However, there is no existing implementation of R-tree supporting concurrent operations. This paper introduces the lock-free concurrent R-tree, which implements an abstract data type (ADT) that provides the operations Add, Remove, RangeSearch. The operations in the Lock Free R-tree use single-word read and compare-and-swap (CAS) atomic primitives, which are readily supported on available multi-core processors. A lock-based implementation is also prepared as a benchmark.

Keywords Lock-Free · Lock-Based · R-tree · Range Search

1 Introduction

We propose using an external 2-way R-tree to implement the ADT = Add, Delete, RangeSearch in a 2-dimensional setting. An R-Tree is a tree-based data structure that can store spatial data indexes. Each entry in an internal node of an R-tree corresponds to MBR(Minimum bounding rectangle), which represents the smallest rectangle containing all its child nodes. R-trees use this concept of MBR to group nearby objects, which helps in spatial search. Each entry in the leaf nodes corresponds to a spatial point.
The main structure of our Lock free R tree is adapted from its sequential implementation. In the sequential implementation, we defined all the classes with its internal structure. Using the sequential implementation we also created a lock-based implementation of R tree. This lock-based implementation provided further insights on additional cases that are needed to be handled in concurrent setting.
Moreover, we will also compare the performance of lock-based implementation of R tree with its lock free implementation. Github Link - github.com/aman-agg/CLDS-Project

2 Abstract Data Type

Multi Dimensional Range Search Tree (MDRST) = {Add, Delete, RangeSearch}

- **Add** - adds point p to the tree
- **Delete** - deletes point p from the tree
- **RangeSearch** - Given a 2-dimensional range, return points lying inside that range.

2.1 Invariants

- Two way R tree i.e. it has two children (leftChild and rightChild).
- There will be atleast a single entry in a node.
- All internal nodes will have both the entries.
- MBR criteria i.e. parent node's MBR will envelop the MBR of its children.
- Tree is imbalanced.

3 Structure of Components

We will be using a 2-way R-tree. In a 2-way R-tree, each node consists of 2 entries (left entry and right entry). Each node contains a link to the parent pointer and its two children.

3.1 Node Class

Algorithm 1 Node Class

1: Entry $leftEntry, rightEntry$
2: Node $leftChild, rightChild, parent$

Explaination

- parent - a link to parent node of type "Node". It will be null, if the parent is not present i.e. the current node is the root itself.
- leftChild - stores the link to the left child is of type "Node". It will be null if leftChild is not present. This can happen if current node is leaf node or tree has some skewness which will be handled by compression()
- rightChild - stores the link to the right child of type "node". It will be null if rightChild is not present. This can happen if current node is leaf node or tree has some skewness which will be handled by compression()
- leftEntry - In the case of an internal node, leftEntry represents the minimum bounding rectangle containing all the nodes present in the left subtree of the current node. In the case of a leaf node, it will be a spatial point.
- rightEntry - In the case of an internal node, rightEntry represents the minimum bounding rectangle containing all the nodes present in the right subtree of the current node. In the case of a leaf node, it will be a spatial point.

3.2 Entry Class

Algorithm 2 Entry Class

1: Point $lowerBottom, upperTop$

Explaination

- lowerBottom - In case of leaf node, represents the spatial point. In case of an internal node, it represents the lower bottom coordinates of the minimum bounding rectangle.
- upperTop - In case of leaf node, it is null (to detect the difference between internal and leaf node). In case of an internal node, it represents the upper top coordinates of the minimum bounding rectangle.

3.3 Atomic Referencing

For Lock-free implementation, we created root with the atomic reference class as it provides a flexible way to update values without use of synchronization. Moreover, we used AtomicReferenceFeildUpdator on the leftChild and rightChild of Node class to update the child links between parent and its children as it provides AtomicReferenceFieldUpdater provides us the capability of performing an atomic Compare-and-swap on a particular field of an object.
We have created two different instances of atomicReferenceFieldUpdater, one for each of the child node links.
Also, root is the only class based reference that needs to support atomic operations. Hence, it is created using the AtomicReference class which provides the capability to perform an atomic Compare-and-swap operation on the root node.

Algorithm 3 AtomicReferencing

1: AtomicReference<Node> $root$ = new AtomicReference<Node>()
2: AtomicReferenceFieldUpdater<Node,Node> $leftChildUpdater$ = newUpdater(Node.class, Node.class, "leftChild")
3: AtomicReferenceFieldUpdater<Node,Node> $rightChildUpdater$ = newUpdater(Node.class, Node.class, "rightChild")

4 Pseudo Code for Lock free implementation of R-tree

Algorithm 4 Add ($newPoint$)

```
restartAddition ← true
while restartAddition do
    if newPoint in tree then
        return
    end if
    if root is null then
        Create newNode with newPoint as an entry
        if root.CAS(null, newNode) then
            restartAddition ← false
        else
            continue
        end if
    end if
    currNode ← root
    parent ← null
    Store the link of currNode to its parent i.e. whether it is left child or right child to its parent
    traversal ← true
    while traversal do
        if currNode is empty leaf then
            Make newNode a copy of currNode
            Add the newPoint in the empty slot of newNode
        end if
        if currNode is an internal node then
            Find minimum MBR expansion while adding the newPoint to left and right child
            if minimum MBR expansion is towards left then
                Traverse left
            else
                Traverse right
            end if
        end if
        if currNode is full leaf then
            Split the currNode into two entries
            Combine three entries i.e. two entries from currNode & one from newPoint into two diff nodes.
            Make newNode parent to these two different nodes.
        end if
        if currNode is full leaf or empty leaf then
            if currNode is root then
                if root.CAS(currNode, newNode) then
                    restartAddition ← false
                else
                    continue
                end if
            end if
            if currNode is its parent's leftChild then
                if leftChildUpdater.CAS(parent, currNode, newNode) then
                    restartAddition ← false
                else
                    if rightChildUpdater.CAS(parent, currNode, newNode) then
                        restartAddition ← false
                    end if
                end if
            end if
            traversal ← false
        end if
```

```
        if restartAddition is false then
            updateMBR(currNode)
        end if
    end while
end while
```

Algorithm 5 Delete ($delPoint$)

```
restartDeletion ← true
while restartDeletion do
    if delPoint not in tree then
        return
    end if
    currNode ← root
    newNode ← null
    didCompression ← false
    parentChildLink ← false
    foundPoint ← false
    parent ← null
    Make a queue Queue and add root to it, for traversal
    while Queue is not empty do
        currNode ← Queue.poll()
        if currNode is null then
            break;
        end if
        if currNode is Internal node then
            Update parentChildLink
            Update parent
            if checkAndCompressSkewed(currNode, parent, parentChildLink) then
                didCompression ← true
                break
            else
                if left child of currNode is not null then
                    Add left child of currNode to Queue
                end if
                if right child of currNode is not null then
                    Add right child of currNode to Queue
                end if
            end if
        else
            if If delpoint is present in currNode then
                foundPoint ← true
                break
            end if
        end if
    end while
    if didCompression then
        continue
    end if
    if foundPoint is false then
        break
    end if
    Check if the currNode is empty leaf or full leaf
    Make a newNode which will be used to replace the currNode using CAS
    if currNode is a full leaf then
        Make the newNode a copy of currNode
        Make the entry that contains the delPoint as null
    end if
```

```
        if currNode is a empty leaf then Assign the newNode as null
        end if
        if currNode is the root then
            if root.CAS(currNode, newNode) then
                restartDeletion ← false
                break
            else
                continue
            end if
        end if
        Check the link of currNode to its parent i.e. whether it is left child or right child
        if currNode is leftChild of its parent then
            if leftChildUpdater.CAS(parent, currNode, newNode) then
                restartDeletion ← false
                break
            else
                continue
            end if
        end if
        if currNode is rightChild of its parent then
            if rightChildUpdater.CAS(parent, currNode, newNode) then
                restartDeletion ← false
                break
            else
                continue
            end if
        end if
    end while
```

Algorithm 6 Compression ($currNode, parent, parentChildLink$)

```
if currNode is null then
    return true
end if
leftChild ← currNode.leftChild
rightChild ← currNode.rightChild
if leftChild and rightChild are not null then
    return false
end if
if leftChild or rightChild is null then
    if parent is null then
        root.CAS(currNode, null)
    else
        if currNode is left child of parent then
            leftChildUpdator.CAS(parent, currNode, null)
        end if
        if currNode is right child of parent then
            rightChildUpdator.CAS(parent, currNode, null)
        end if
    end if
    return true
end if
```

Algorithm 7 RangeSearch

To be added

5 Future Work

We are currently at the stage of integrating the compression algorithm into the add and delete algorithm of our R-tree. We are also working on implementing the lock free rangeSearch operation using wait free snapshot technique.

Acknowledgments

This implementation of lock free R-tree is proposed as part of course project towards our course "Concurrent and Learned Data Structures", taught by Dr. Bapi Chatterjee at IIIT Delhi. We had multiple discussions with him where he guided us and provided us multiple resources to tackle our problems. This project is only possible because of his continuous contributions.

References

- Chatterjee, Bapi. "Lock-Free Linearizable 1-Dimensional Range Queries." Proceedings of the 18th International Conference on Distributed Computing and Networking, Association for Computing Machinery, 2017, pp. 1–10. ACM Digital Library, https://doi.org/10.1145/3007748.3007771.

- "R-Tree." Wikipedia, 14 Mar. 2022. Wikipedia, https://en.wikipedia.org/w/index.php?title=R-treeoldid=1077176048.

- Rtree-Chap1.Pdf. https://tildesites.bowdoin.edu/ ltoma/teaching/cs340/spring08/Papers/Rtree-chap1.pdf. Accessed 22 Apr. 2022.