Lock-free Radial Spatial Index

CSC 490: Lock-free, GPU & vectorization

Assignment 1

Victor Kamel
vkamel@uvic.ca
University of Victoria
Victoria, BC, Canada

1 INTRODUCTION

In this report, we will present a thread-safe and lock-free spatial index for performing top-K queries based on cosine similarity in two dimensions. Unlike previous work on this subject [2], this index is not static, and allows for arbitrary insertions but not deletions. The index itself is implemented as a modified internal binary search tree.

2 DEFINITIONS

We define our target metric as follows.

Definition 1. *Cosine similarity [2]. Given $\vec{a}, \vec{b} \in \mathbb{R}^2$,*

$$S_{cos}(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}||\vec{b}|} = \frac{\sum_{i=1}^{2} a_i b_i}{\sqrt{\sum_{i=1}^{2} a_i^2}\sqrt{\sum_{i=1}^{2} b_i^2}}.$$

S_{cos} has a range of $[-1, 1]$ where vectors that point in the same direction have a similarity score of 1, orthogonal vectors have a similarity of 0, and vectors that point in opposite directions have a similarity score of -1.

Definition 2. *Top-K query [2]. Given a set $V \subseteq \mathbb{R}^2$, $|V| = n$ we define the top-K query based on a similarity function S to a given $\vec{q} \in \mathbb{R}^2$ with $k \in [0, n]$ as*

$$topK(V, \vec{q}, k) = argmax_{K \subseteq V, |K|=k} \sum_{\vec{p} \in K} S_{cos}(\vec{p}, \vec{q}).$$

We will make all assumptions consistent with Kamel [2], since the focus of this report is on the underlying data structure proposed and its properties rather than the problem we will be using it to solve. Of notable difference is that we will assume that V may grow, but will not shrink.

In terms of algorithmic analysis, we will assume the worst case runtime in the word RAM model, and we will use linearizability [1] as our notion of consistency.

3 DATA STRUCTURE

The index consists of a map, underpinned by an unbalanced internal binary search tree. That is to say, data is stored both in the internal nodes and the leaves. Each node contains a key, value and an insertion identifier. The insertion identifier is used to guarantee a form of consistency during `Find` and `FindNext` operations. In addition, each nodes has 3 pointers: one for the left child, one for the right child, and an additional pointer for the parent. The parent pointer can subsequently be used to iterate from any node to the next or previous in sorted sequence.

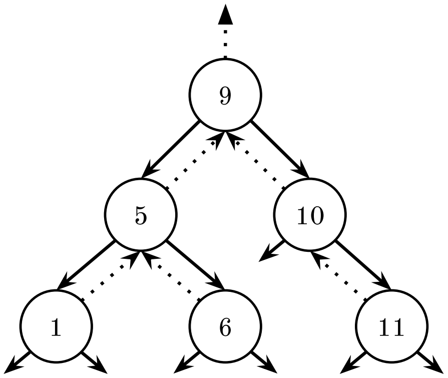

Figure 1: Simplified diagram of binary search tree data structure. Only keys are shown. Values and insertion identifiers are omitted.

We assume that keys are unique. Thus, given a node with key k, we will maintain that all nodes in the left subtree will have keys strictly less than k, and all nodes in the right subtree will have keys strictly greater than k. The tree class stores a pointer to the root that is initially `NULL`, and an insertion epoch counter that starts at 0. We assume that reads and writes to this counter are atomic operations, and that this number does not overflow.

4 TREE OPERATIONS

We will now elaborate on the interface provided by the map, demonstrate correctness, and provide asymptotic time complexities.

4.1 Insert

`Insert(key, value)` will insert a new key/value pair into the map. We begin by creating a node with the given key k, value v, and a placeholder insert epoch identifier. We also load the root pointer's value into the "current node" pointer.

Within the CAS loop, we search, starting from the current node pointer, in the direction of key k until we either arrive at a pointer to a leaf (or `NULL` pointer in the case of an empty tree). If the found node has the same key as the new key we are trying to add, we will deallocate the node and throw an exception. otherwise, we set our node's parent pointer to the found pointer and the node's epoch with an atomic fetch add.

We will now try to CAS either the appropriate left or right pointer of the leaf node (depending on k), or the tree's root pointer, from a `NULL` pointer to a pointer to the new node. If the CAS fails, we will re-enter the loop, with the current node pointer's value updated to the value loaded during the CAS. We can start searching again from this value rather than the root, because we know that the path we took to arrive to this node could not possibly have changed (since insertions only happen at the leaves, and deletions cannot occur), and thus the new correct insert location is either the left or right child pointer the node pointed to by the current node pointer.

Theorem 4.1. *The* `Insert` *operation is linearizable.*

Proof. We need to show that at some instant between the invocation and return of the function, there is a point, the "linearization point," at which the effect of the function appears to take place instantaneously.

Consider the moment at which the CAS succeeds. Prior to this, the node that we created is thread-local, and the path that we traversed to the leaf is static (since nodes cannot be removed). Until the CAS succeeds, the changes to the data structure are not visible. At the point it does, the node with the correct key, value, and insert identifier is available to all threads. We are guaranteed to insert into the correct location, since we either arrived at the correct leaf and succeeded in inserting the new node, or failed some number of times, but each time advanced to the new correct insert location (and updated the insert epoch counter). □

We observe that due to this, we can ensure that the tree remains valid (retains its sorted property).

Corollary 4.2. *At any given point in time, the tree is a valid binary search tree.*

Proof. The only operation that can change the tree is the `Insert` operation. In addition, an inserted node can never be removed. We have shown in Theorem 4.1 that `Insert` operations have a single point between their invocation and return at which the insert occurs from the point of view of other threads. Thinking inductively, in the base case, an insert into an empty tree produces a tree with a single node which is trivially a valid binary search tree. Given an arbitrary (valid) tree, an insert operation will first find the correct location, then insert the node, all as if in a single point of time. Thus, as insert operations are performed, the tree remains valid. □

In addition, due to the epoch counter, we can make the following guarantee:

Lemma 4.3. *At any given time, all nodes from root to leaf have strictly increasing insertion identifiers.*

Proof. By contradiction. Suppose that node a is the parent of node b, and b has a lower insertion identifier than that of node a. This must mean that node b's CAS succeeded before node a's CAS, since the epoch number increases (globally) every time the CAS fails. However, if this were the case, then node a cannot be b's parent, since it would not have been attached to the tree yet (and thus invisible to node b). This property holds for every adjacent pair of nodes in the path. □

We can also see that the `Insert` operation runs in $O(h)$ time, where h is the height of the tree (at the point where the insert operation succeeds). Since the tree is unbalanced, h may be as large as the total number of nodes.

4.2 Find

`Find(k)` returns a `NULL` pointer if the tree is empty, a pointer to a node with key k if such a node exists, or otherwise a pointer to the leaf node that would be the parent of a node with key k should it be inserted.

First, we atomically read the current epoch, say c. All inserts that occur after this moment are ignored during the find operation, since we will treat all node pointers with insertion identifiers greater than or equal to the current epoch as `NULL`. By Corollary 4.2 and Lemma 4.3, we know that at a given time, all nodes *successfully inserted* with an epoch $< c$ are reachable by Lemma 4.3 (if they were not, then they must have a node with a greater epoch between them and the root, which is a contradiction).

We then follow the pointers from the root until we reach a `NULL`, choosing the left or right child based on the key of the node.

Theorem 4.4. *The* `Find` *operation is linearizable.*

Proof. We shall determine a linearization point for the `Find` operation.

When reading the epoch counter atomically to get some value c, consider the set $I, |I| = \ell$ of insert operations with insert identifiers strictly less than c that have yet to complete (and, will complete before the `Find` operation has concluded), with keys $k_0, k_1, \ldots, k_\ell$. The `Find` operation either linearizes at the point where for every $k_i,\ i = 0, 1, \ldots \ell$, the search has come across a node with key k and proceeded down the subtree that does not contain k_i (that is to say $K_i < k$ and $k <$ search key, or $k_i > k$ and $k >$ search key). Or it reads a pointer to a node with key k, or it reads a `NULL` pointer (this could be a pointer to a node with insert identifier $\geq c$), whichever occurs first. □

In addition, since this function operates on static data and does not interact with other threads, we claim that it is wait-free in addition to lock-free. Trivially, we can say that the `Find` operation runs in $O(h)$ time.

4.3 Find Next

`FindNext(node_ptr, epoch, direction)` will find the next in-order node pointer in the given direction, with wrapping (modular) semantics. We will consider the case where we are moving to the next node in sequence, but the argument for moving in the direction of the previous node is symmetric. We consider pointers to nodes with an insert identifier greater than or equal to the epoch at `NULL` pointers. Given a non-`NULL` node pointer, we have two cases:

1. The current node has a right child. In this case, the left-most node of the right subtree is the next node.
2. The current node does not have a right child. Then, we must find the parent of the root of the subtree that the current node is the right-most child of. This is done using the parent pointers, by climbing the tree until we have traversed a left edge.
   (a) If there is no such node (we are the right-most node in the tree), we must find the left-most node in the tree (wrap around).

This algorithm is based on [3]. However, instead of using a stack to keep track of the path from the root to the current node in order to backtrack up the tree in the second case, we can use the parent pointers of the nodes themselves.

Theorem 4.5. *The* `FindNext` *operation is linearizable.*

Proof. We shall determine a linearization point for the `FindNext` operation.

Consider the first case, where we iterate down the tree towards a leaf. We can see that the linearization point is exactly analogous to `Find` in Theorem 4.4, except that we are not searching for a target node (or, we are searching for the node's key in the right subtree). This is also the case for when we are at an extreme end of the tree. We first ascend to the root, then descend again, linearizing in the same manner.

Consider the second case, where we iteration up the tree towards the root. Since the path from the root to any node is immutable, we know that an insertion into the path from the current node to the root is not possible. Thus, we linearize immediately at the invocation. □

In the worst case, when we are at an extreme of the tree, we require about $O(2h) = O(h)$ steps to find the next node (h to get to the root from the leaf, then h to get from the root to the leaf). However, an amortized analysis would provide a more favourable runtime, since it is clear that if iterating over an entire tree from the smallest to largest element, each edge is traversed only twice. We will note that a complete traversal will not necessarily be consistent, as the elements visited as we traverse using this method will depend on pending insert operations in the same epoch. However, we can be certain that the relative order of elements in the traversal will be correct, as at each step, the tree is valid by Corollary 4.2.

This operation is also wait-free.

5 INDEX OPERATIONS

The index class instantiates a map as defined above. In the preprocessing step, vectors are linearized according to [2], denoted here as $L(\vec{q})$, and these linearized values are used as the keys, with the vectors themselves being stored as the values.

All operations described here are implemented using `Insert`, `Find` and `FindNext`.

Theorem 5.1. `contains`*,* `knn`*,* `nearest` *and* `insert` *are linearizable.*

Proof. Since we have shown that all operations on our lock-free map are linearizable, and it is known that compositions of linearizable functions are themselves linearizable, all of our index operations will be linearizable as well. □

5.1 Contains

`contains`$(\vec{q})$ will invoke `Find`$(L(\vec{q}))$ and compare the result with $\vec{q}$. We see this this is wait-free since `Find` is wait-free, and runs in $O(h)$ time.

5.2 KNN

`knn`$(\vec{q},$ `n`$)$ will return the the top-K closest vectors to in $\vec{q}$ in the index. If there are insufficient vectors in the index, the result is undefined. Although no vectors that are inserted after this operation has run will be included (due to the epoch counter), if some vectors in the current epoch have not finished being inserted by the time this function is called, it is possible that these vectors are not present in the output if they are added after their position in the index has been iterated past. The algorithm itself is identical to *RadialIndex* presented in Kamel [2], with iterators into the linear array that use modular arithmetic replaced by iterators into the binary tree that use `FindNext`.

The runtime complexity of the query is $O(h + kh) = O(hk)$ in order to first find the nearest node to k, then to iterate k times. This operation is wait-free.

5.3 Nearest

`nearest`$(\vec{q})$ will invoke `knn`$(\vec{q},$ `1`$)$, and thus run in $O(h)$ time. This operation is wait-free.

5.4 Insert

`insert`$(\vec{q})$ will invoke $Insert(L(\vec{q}), q)$, which is lock-free and runs in $O(h)$.

6 CONCLUSION

We have presented a thread-safe, lock-free algorithm for performing spatial queries. Future work should address the inherent limitations of this design, namely lack of balance, absent support for deletion operations, and epoch counter rollover.

REFERENCES

[1] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (jul 1990), 463–492. https://doi.org/10.1145/78969.78972

[2] Victor Kamel. 2023. Optimizing Index Structures to Support Semantic Queries in Relational Databases. http://hdl.handle.net/1828/14914

[3] Ben Pfaff. 2004. An Introduction to Binary Search Trees and Balanced Trees.