Oyewale Oyediran Follow
Software Engineer. Life-Long-Learner.
Sep 6, 2017 · 5 min read

Spatial Indexing with Quadtrees

> *"Hello World" is in order, as this is my first article on Medium. I hope to keep this up and write more.*

I'm going straight to the point here. Traditional DBMSs and their standard index implementations are able to index data based on various natural orderings ($1 < 2 < 3$ and some other similar ordering). An important ordering that occurs in many software applications is the **proximity ordering** in a 2-D space (we'd see later in the article how our spherical earth gets abstracted to a 2-D space).

> *In comes Spatial Indexing*

Spatial indexing is an increasingly important area of geo-spatial application design. The use-cases abound around us. Ride-sharing apps (Uber) need to be able to track the location of cars in near real-time, and provide user updates via extremely fast geo-queries. Facebook wants to let you know when your friends are nearby. The entities (Uber rides, Facebook users) and associated meta-data are stored in traditional tables and a spatial index is built separately with the coordinates of the entities. The spatial index allows special geometric queries to be executed efficiently.

With this index, these apps can efficiently query a database based on the spatial relationship of its entities. Some important queries that can be run from spatial indexes are listed.

Nearest neighbour (NN) queries are important for apps that provide nearby points of interests. Having a 'nearby friends' feature seems to be mainstream in social network applications, which can be easily powered by NN queries.

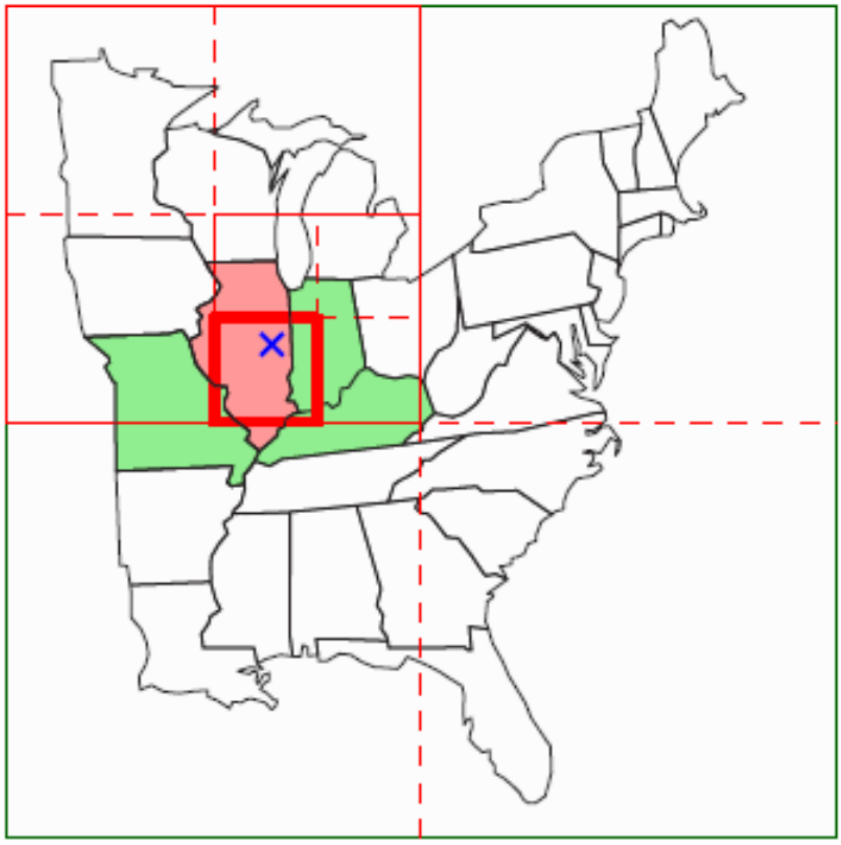
Geo-distance Range Queries: These queries help retrieve entities within a specified range. Some use-cases include finding all cabs within a 2km radius from a user's location. Yeah, Uber comes to mind here.

A properly designed spatial indexing scheme is a central part of building high performance geo-apps.

The beauty called a quadtree

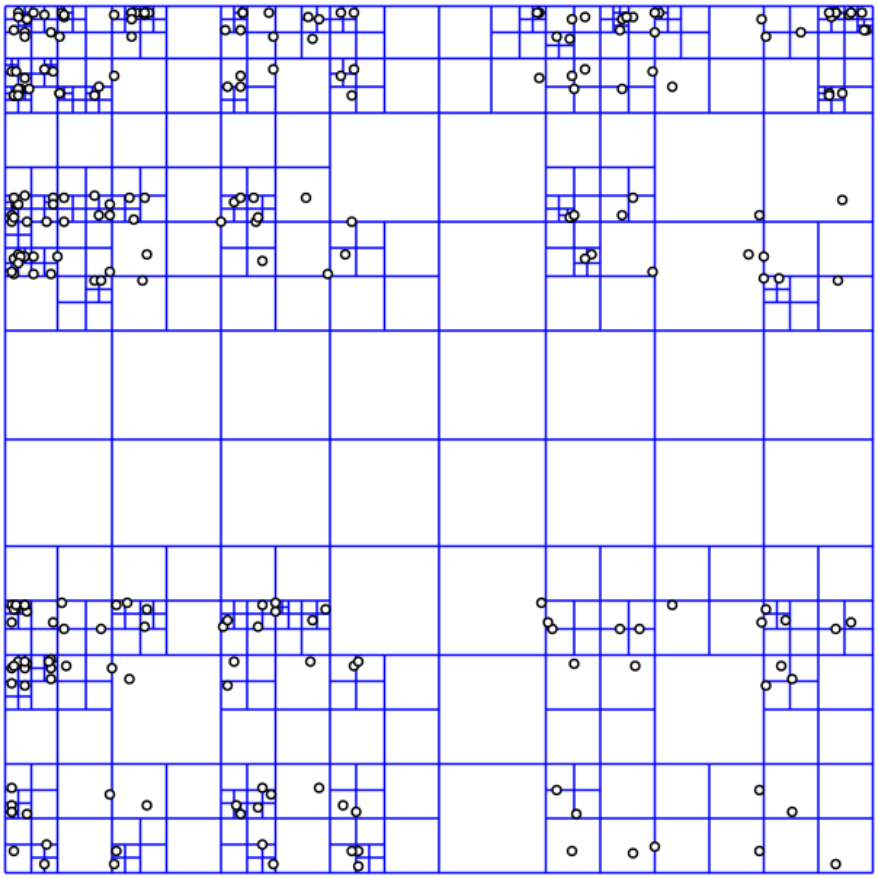
The Quadtree is one of my favorite data structures. As the name implies, it is a variant of the Tree data structure. Trees generally have internal nodes (nodes that have at least one child) and leaf nodes which have no children. These nodes holds data that are ordered in a hierarchical order.

A quadtree is a tree data structure in which each node has zero or four children. Its main peculiarity is its way of recursively dividing a flat 2-D space into four quadrants.



Quadtree representation of USA. http://www.cs.tau.ac.il/~haimk/seminar12b/Quadtrees.pdf

Some quadtree use-cases includes Image processing, sparse data storage, spatial indexing etc. This article focuses on the spatial-indexing use-case.



A quadtree representation with Bucket size of 1. Source: Wikipedia

Inserting data into a Quadtree

Inserting data into a quadtree involves recursively determining which of the four children (quadrants) the data-point should occupy, until you reach a leaf-node (quadrant). If the elements in the leaf-nodes exceeds a pre-specified bucket size, the leaf node is split into four equal quadrants and the points are rearranged into their correct quadrants.

```
def insertInTree(root, data):
    """Pseudo code for inserting a point in a Quadtree"""
    if not root
        createLeafAndInsertNode(root, data)
    elif root.isLeaf() and root.size() < BUCKET_SIZE:
        root.addNode(data)
    elif root.isLeaf(): # Leaf node must be full
        root.decomposeLeafNodeAndInsert(data)
```

```
    # Find the appropriate sub-tree to insert node
    elif root.northwest.isValidParent(data)
        insertInTree(root.northwest, data)
    elif root.southwest.isValidParent(data)
        insertInTree(root.southwest, data)
    elif root.southeast.isValidParent(data)
        insertInTree(root.southeast, data)
```

```
        else
            insertInTree(root.northeast, data)
```

The recursive algorithm above can be adapted to implement a Delete operation for the quadtree. Thus in a **reasonably balanced quadtree, we would have insert and delete operations implemented in logarithmic time.

***Worst case scenario is generally O(n) time, when the tree is extremely unbalanced.*

Range Query

A range query is an important function you could do with a Quadtree. It is generally of the form: retrieve all points within a spatial range. In real life applications, A ride-sharing service wants to expose an API to provide the cars available within 1km of a user's location, or Facebook's "nearby friends" feature wants to expose a list of friends in a user's town.

```
def getPointsInRange(root, range):
    points = []

    # If there is no intersection with the area, return
    if not root.intersect(range):
        return points

    # Return all data points on a leaf node
    if root.isLeaf():
        points.append(root.getNodes())
        return points


    # Recursively append the points from the 4 quadrants
    points.append(getPointsInRange(root.northwest, range))
    points.append(getPointsInRange(root.northeast, range))
    points.append(getPointsInRange(root.southeast, range))
    points.append(getPointsInRange(root.southwest, range))

    return points
```

Important Quadtree Spatial Indexing Implementation Notes

> *The earth is not flat*

The earth is not a flat 2-D surface, but it is an abstraction we make (like most flat maps) to help us divide the earth in a form that makes for easy representation. A proper quadtree implementation requires an optimal approach to representing the near spherical earth to a flat surface. GIS (and other geography) specialists generally make use of projections for these representations. I have found that a decent number of quadtree implementations tend to favour the Sinusoidal projection.

> *Tile Size, Bucket Size makes a whole lot of difference*

Getting optimal performance out of your Quadtrees involves choosing the right tile-size (quadrant dimensions) depending on your particular use case. In some use-case, having the 2-D surface pre-divided into its composing grids may provide better performance at the expense of memory.

The Bucket size, the maximum number of data points in each leaf node, is also another variable that can be tuned to get better performance from the Quadtree. The ideal bucket-size depends on the use-case of the Quadtree. A quadtree with a bucket-size of 1 is quite easy to implement, but in real-world applications you get to have multiple entities in the same spatial region. The data points in the leaf node could be implemented with a LinkedList.

> *You are probably using Quadtrees already*

Quadtrees are actually in use in a couple of DBMS today, although R-Trees seem to have more support. Oracle's spatial index has the option to use a QuadTree. MongoDB's spatial appears to use GeoHashes instead. So Geohashes, R-Trees are Quadtree alternatives.

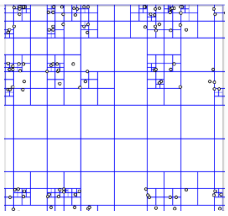
There are lots of decent quadtree implementations on Github. Here is one https://github.com/karimbahgat/Pyqtree

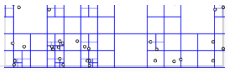
Please leave a comment to let me know what you think and perhaps hit the clap (recommend) button so more people can.

Further Readings

Quadtree - Wikipedia

A quadtree is a tree data structure in which each internal node has exactly four children. Quadtre...



en.wikipedia.org 

https://dzone.com/articles/algorithm-week-spatial
http://blog.notdot.net/2009/11/Damn-Cool-Algorithms-Spatial-indexing-with-Quadtrees-and-Hilbert-Curves
https://www.cs.umd.edu/class/spring2008/cmsc420/L17-18.QuadTrees.pdf
http://www.cs.tau.ac.il/~haimk/seminar12b/Quadtrees.pdf
http://www.cs.cmu.edu/afs/cs/academic/class/15456-s10/ClassNotes/nn.pdf
http://www.geomesa.org/assets/outreach/SpatioTemporalIndexing_IEEEcopyright.pdf