Selections from R-Trees: Theory and Applications

R-Trees are a relatively recent development in the field of data structures, building off of the foundation provided by B-Trees. To the uninitiated, B-Trees are balanced trees with nodes that contain a range of values. They usually have small heights and do not need to be rebalanced often. They are useful for storing large objects because of cache locality.

R-Trees are similar except that they are used for multi-dimensional data and correlate objects together within the same node that are geometrically close. To do so a minimized bounding rectangle (of any dimension) is drawn around nodes in order to organize them. The leaves are pointers to the actual objects which may be stored on disk pages. The R-Tree allows for a spatial search for objects within a defined location. Tree levels are searched to find which MBR contains the data, recursively searching finer grained MBRs.

An R-Tree supports three basic operations: searching, insertions and deletions. Searching, although not bounded, generally does well because most data is not touched by ignoring MBRs not within the search constraint. Insertion is straightforward by finding the correct MBR and placing it within a node. A difficulty arises though when a node becomes full. At this point a splitting of the node must occur which minimizes the total area of the new MBRs. There are a number of ways for finding the new split. The original R-Tree used a quadratic split which maximizes the dead space inside MBRs and then assigns nodes so that it requires the smallest MBR expansion. Deletion is interesting in that it does not just merge under-filled nodes, but instead uses the insertion algorithm to reinsert them. This helps to maintain the balanced property.

The R-Tree had a few disadvantages which made its performance suboptimal. The R+ tree was an attempt to make searching faster by avoiding multiple paths and by eliminating overlapping MBRs on the same level. These results mean that some objects may be duplicated within nodes. Additionally, on insertion, a level above or below may need to be changed (contrasted with the original R-Tree which only changed upper levels) in order to avoid overlapping.

The R*-tree is accepted as the best performing R-Tree due to its consideration of a number of factors which affect query time. 1) dead space is minimized in MBRs 2) overlaps are minimized, 3) higher level MBRs are smaller, and 4) storage utilization is maximized. Upon an insertion, in the case of an overfull node, nodes are not split automatically. Instead a portion of the node is reinserted. This helps to improve performance by keeping a better balance.

Another type mentioned is the Hilbert R-Tree, which uses the Hilbert value to ensure the nodes on a level are ordered. It cannot handle large objects though in terms of performance, which is due to increased overlaps as size grows.
The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles

The R-Tree is a popular way to perform spatial access methods on multi-dimensional data. The original creates the data structure based on minimizing the area of the bounding rectangles. Whether this is the best method was not explored in the original paper. The R*-Tree aims to encompass a larger set of criteria when deciding where to place nodes. The R*-Tree outperforms other implementations and works well for point access and spatial joins across files.

The heuristic for building a bounding box is affected by a number of criteria, any of which can change the others. An engineering approach is taken towards finding the best mix of: 1) area covered by a directory rectangle, 2) overlap between rectangles, 3) margins of rectangles (the sum of the lengths of a rectangle) and 4) the storage utilization. One such trade off is that when area is minimized, storage utilization drops.

Two main problems are identified with the R-Tree algorithm: bad splits may be made depending on the choice of seeds for splitting and when nodes overflow, the extra nodes are assigned without consideration of their geometry. To solve this problem, the R*-Tree chooses a different subtree from the R-Tree on descent that minimizes overlaps on leaf nodes. On splits, a number of methods are used to choose the final split: area-value, margin-value, and overlap value. The R-Tree method of handling deletes, by re-inserting nodes instead of merging, does admirably well and is responsible for decreasing overlaps, improving storage utilization and splitting less frequently.

Extensive testing was done in order to validate the improvements made. Using a set of six varied data sources, different types of queries were performed that tested rectangle intersection, point location and rectangle enclosure. The relevant statistic was the number of disk accesses made (the R-Tree is a structure designed for large objects that are too big to fit in main memory). In all tests, the R*-Tree performed much better than other variants in disk accesses and storage utilization. The linear splitting R-Tree, although popular, was the worst performer. A special case was made in order to investigate how R*-Trees function with degenerate rectangles (ie. points). It had a very favorable query time, but insertion time was higher than a popular point storage system referred to as the 2-level grid file method.
Indexing for Dynamic Abstract Regions

The authors present a new type of spatial representation tree, the RC-Tree, which actually stores the original geometry of objects. Its aim is to store information completely in memory, an idea contrary to the original motivations of R-Trees. It employs three main ideas in order to save space, keep better balanced trees and improve search characteristics: domain reduction, object clipping and re-balancing.

An RC-Tree is actually a binary tree where each node partitions its space based on a discriminator, or a rule that can determine which of two sets a node belongs in. All original objects are stored in the leaf nodes, which have a predetermined size. The algorithm differs from previous work such that on an insert, one either goes left or right down the tree, rebuilding lazily along the way, and splitting when a node is overfull. Splitting involves taking a discriminator, which can partition the nodes, and moving each group to the new location. On a rebuild, balancing is done (much like older R-Trees) by removing all the nodes in the unbalanced subtree and re-inserting.

Clipping is the process by which bounded boxes are drawn around the actual shape. These boxes may change depending on the discriminator criteria. By using sub-objects that have been clipped, the area of the MBR becomes reduces, a method referred to as domain reduction. Re-balancing is done when the height of the tree passes some threshold (a scale of \( \lg n \)). It is performed on insertion, and although it doesn't happen every time, it slows the average performance on an insert.

In order to evaluate the effectiveness of the RC-Tree, the number of comparisons are used, as execution time is an artifact of the implementations. They found that although insertion was slightly higher than other methods, due to occasional re-balancing, querying was significantly better. The space requirements are roughly the same, even though the original object geometry is not being discarded. This space requirement can be changed though, and it is possible to tune the algorithm for faster performance, but with higher memory costs. This trade off originates from the choice of the discriminator splitting, which can be done slowly for better results or quickly for an approximation (RC-SWEEP or RC-MID).