CSC6210: Advanced multidimensional search CSE department, Chinese University of Hong Kong Fall 2009

Lecture 12: R-trees

By Yufei Tao (taoyf@cse.cuhk.edu.hk)

Last revision: Nov 22, 2009

## Forewords

We have learned many structures *dedicated* to specific problems. For example, *the persistent B-tree* is good for orthogonal segment-intersection queries, *the external interval tree* is good for stabbing queries, *locality sensitive hashing* for nearest neighbor search, and so on. In practice, however, it is unrealistic to create so many different indexes on the same dataset, because doing so will incur prohibitive space consumption and update overhead. Therefore, it would be nice to have a single, all-around, structure, which occupies small space, can be updated efficiently, and most importantly, supports a large variety of queries.

In this lecture, we will discuss such an all-around structure called *the R-tree*. Unlike the structures mentioned earlier, the R-tree is heuristic in nature, because it does not have any attractive theoretical guarantees on the search performance. Nevertheless, the practical efficiency of this structure has been widely established for many problems, especially if the dimensionality is low. Interestingly, for realistic datasets, there has been evidence [1] that R-trees even outperform some theoretical worst-case efficient structures. This is not as surprising as it may appear. The design of a theoretical structure aims at handling the most adverse datasets. Much of the design is not really needed for "good" datasets, and thus, may actually cause unnecessary overhead on such data.

# 1 The structure

There are many variations of the R-tree in the literature [2, 3, 5, 6, 7]. The version to be described below is similar to the  $R^*$ -tree [2], with some simplification in order to focus on the most crucial ideas. Also, our discussion is based on 2d point data, but the extensions to rectangle data and higher-dimensionalities are straightforward.

Let P be a set of points. An R-tree stores all these points in leaf nodes, each of which contains  $\Theta(B)$  points, where B is the size of a disk page. Each non-leaf node u has  $\Theta(B)$  children, except the root which must have 2 children at minimum unless it is the only node in the tree. For each child v, u stores a *minimum bounding rectangle* (MBR), which is the smallest rectangle that *tightly* encloses all the data points in the subtree of v. Note that there is no constraint on how points should be grouped into leaf nodes, and in general, how non-leaf nodes should be grouped into nodes of higher levels. Since each point is stored only once, the entire tree consumes linear space O(N/B), where N is the cardinality of P.

Figure 1 shows an example where P has 13 points  $p_1, p_2, ..., p_{13}$ . Points  $p_1, p_2, p_3$ , for example, are grouped into leaf node  $u_1$ . This leaf is a child of of non-leaf node  $u_6$ , which stores an MBR  $r_1$  for  $u_1$ . Note that  $r_1$  tightly bounds all the points in  $u_1$ .



Figure 1: An R-tree

#### 2 Insertions and deletions

Intuitively, in a good R-tree, nodes should have small MBRs. To see this easily, think about how to use the tree in Figure 1 to answer a range query. Namely, given a query rectangle q, we want to find all the points in P that are covered by q. It is easy to see that we only have to visit those nodes whose MBRs intersect q. Therefore, reducing the extents of the MBRs benefits query efficiency as fewer MBRs are expected to intersect q.

What do we mean, however, by a small MBR? Or in other words, what should the update algorithms of an R-tree minimize about the MBRs? A quick answer to think of is the area, but there exists a better answer. Figure 2 shows two MBRs, which actually have the same area. It turns out that the right (square) MBR leads to better performance in practice. Implication? We should minimize the *perimeter* of an MBR. This may look fairly reasonable in retrospect: a rectangle with a small perimeter almost always has a small area, but the opposite is not true.



Figure 2: The right MBR is preferred

Next, we describe the insertion and deletion algorithms, both of which attempt to create square-like MBRs by reducing their perimeters as much as possible.

#### 2.1 Insertions

To insert a point p, we use a strategy similar to that of a B-tree. Specifically, we add p to a leaf node u by following a single root-to-leaf path. If u overflows, it is split, which creates a new child of parent(u). In case parent(u) overflows, it is also split, which propagates upwards in the same manner. Finally, if the root is split, then a new root is created.

While all these sound familiar, there are, however, two important differences. First, although in

the B-tree the insertion path is unique (i.e., the leaf supposed to accommodate the new item is unambiguous), this is not true at all for the R-tree. In fact, the new point p can be inserted into any leaf, which always results in a legal structure. If, however, a bad leaf is chosen (to contain p), its MBR may need to be enlarged substantially, thus harming the efficiency of the tree. Second, the split algorithm is not as trivial as in a B-tree because now we have multiple dimensions to tackle. Next, we will deal with the two issues separately. Note that the (heuristic) strategies to be introduced are not the only ones. In fact, this is why there are so many variants of R-trees – each of them has its own strategies.

**Choosing a subtree to insert.** We are essentially facing the following problem. Given a non-leaf node u with children  $v_1, v_2, ..., v_f$  for some  $f = \Theta(B)$ , we need to pick the best child  $v^*$  such that the new point p is best inserted into the subtree of  $v^*$ . An approach that seems to work well in practice is a greedy one. Specifically,  $v^*$  can simply be the child  $v_i$  whose MBR requires the *least increase* of perimeter in order to cover p. For example, in Figure 3, both MBRs  $r_1$  and  $r_2$  must be expanded to enclose p, but  $r_2$  incurs smaller perimeter increase, and hence, is a better choice.



Figure 3: MBR  $r_2$  requires smaller perimeter increase to cover p

It is possible that p falls into the overlapping region of multiple MBRs. All those MBRs have a *tie* because none of them needs any perimeter increase to cover p. In this case, the winner can be decided according to other factors such as picking the MBR having the smallest area.

**Node split.** The node split problem can be phrased as follows. Given a set S of B + 1 points, split it into disjoint subsets  $S_1$  and  $S_2$  with  $S_1 \cup S_2 = S$  such that

- $|S_1| \ge \lambda B$ ,  $|S_2| \ge \lambda B$ , where constant  $\lambda$  is the minimum utilization rate of a node, and
- the sum of the perimeters of  $MBR(S_1)$  and  $MBR(S_2)$  is small.

In the sequel, for simplicity we assume that |S| is an even number, and  $|S_1| = |S_2| = |S|/2$ , i.e., we always aim at an even split. The extensions to uneven splits are straightforward.

Ideally, we would like to find the *optimal* split that minimizes the perimeter sum of  $MBR(S_1)$  and  $MBR(S_2)$ . Since an MBR is decided by 4 coordinates (i.e., a pair of opposite corners), it is easy to find the optimal split in  $O(B^4)$  time. This can be significantly improved to  $O(B^2)$  time using a trick in [4]. Unfortunately, even a quadratic split time is usually excessively long in practice. Therefore, we turn our attention to heuristics that do not guarantee optimality, but usually produce fairly good splits. Next, we will describe a split algorithm that runs in  $O(B \log B)$  time, or  $O(dB \log B)$  time in general d-dimensional space.



(a) Split by cutting the x-dimension (b) Split by cutting the y-dimension

Figure 4: Splitting a node

The idea of our algorithm is to always split S using an axis-orthogonal cut. Consider, for example, a cut along the x-axis. For this purpose, we sort the points of S in ascending order of their xcoordinates. Then, we put the first B/2 points in the sorted order into  $S_1$ , and the rest into  $S_2$ . The split along the y-axis can be obtained in the same way. See Figure 4 (where S has 8 points). The final split is the better one of the two splits.

The above applies to splitting a leaf node. The case of non-leaf node is a bit different because the items to be split are MBRs, as opposed to points. Nevertheless, similar heuristics can still be applied by, for example, sorting the MBRs by their centrioids along each dimension.

## 2.2 Deletions

Deleting a point from an R-tree is carried out in an interesting manner. In particular, node underflows are handled in a way that differs considerably from the conventional merging approach as in a B-tree.

Specifically, let p be the point to be deleted. First, we need to find the leaf node u where p is stored. This can be achieved with a special range query using p itself as the search region. Then, p is removed from u. The deletion finishes if u still has  $\lambda B$  items, where  $\lambda$  denotes the minimum node utilization. Otherwise, u underflows, which is handled by first removing u from its parent, and then re-inserting all the remaining points in u (using exactly the insertion algorithm mentioned earlier). See Figure 5.

Note that removing u from parent(u) may cause parent(u) to underflow too. In general, the underflow of a non-leaf node u' is also handled by re-insertions, with the only difference that the items re-inserted are MBRs, and each MBR is re-inserted to the same level of u'.

It is worth mentioning that while we can also design merging-based algorithms to handle node underflows, re-insertion actually gives better search performance [2]. This is because the structure of an R-tree is sensitive to the insertion order of the data points. Re-insertion gives the the earlyinserted points to be inserted in other (better) branches, thus improving the overall structure.

# References

 L. Arge, M. de Berg, H. J. Haverkort, and K. Yi. The priority R-tree: A practically efficient and worst-case optimal R-tree. In Proc. of ACM Management of Data (SIGMOD), pages 347–358, 2004.



Figure 5: Handling a node underflow

- [2] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R\*-tree: An efficient and robust access method for points and rectangles. In Proc. of ACM Management of Data (SIGMOD), pages 322–331, 1990.
- [3] N. Beckmann and B. Seeger. A revised R\*-tree in comparison with related index structures. In Proc. of ACM Management of Data (SIGMOD), pages 799–812, 2009.
- [4] Y. J. García, M. A. Lopez, and S. T. Leutenegger. On optimal node splitting for R-trees. In Proc. of Very Large Data Bases (VLDB), pages 334–344, 1998.
- [5] A. Guttman. R-trees: a dynamic index structure for spatial searching. In Proc. of ACM Management of Data (SIGMOD), pages 47–57, 1984.
- [6] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In Proc. of Very Large Data Bases (VLDB), pages 500–509, 1994.
- [7] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-tree: A dynamic index for multidimensional objects. In Proc. of Very Large Data Bases (VLDB), pages 507–518, 1987.