R-tree

R-tree是用来做空间数据存储的树状数据结构。R-tree是B-tree向多维空间发展的另一种形式，并且R树也是平衡树。

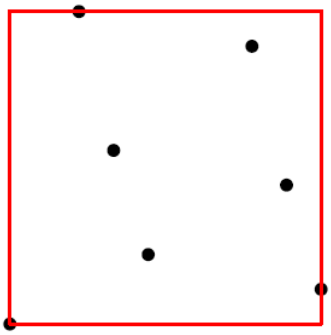
R树的核心思想是聚合距离相近的节点并在树结构的上一层将其表示为这些节点的最小外接矩形，这个最小外接矩形就成为上一层的一个节点。因为所有节点都在它们的最小外接矩形中，所以跟某个矩形不相交的查询就一定跟这个矩形中的所有节点都不相交。叶子节点上的每个矩形都代表一个对象，节点都是对象的聚合，并且越往上层聚合的对象就越多。

R树的主要难点在于构建一棵既能保持平衡（所有叶子节点在同一层），又能让树上的矩形既不包括太多空白区域也不过多相交（这样在搜索的时候可以处理尽量少的子树）的高效的树。

1 R-tree数据结构

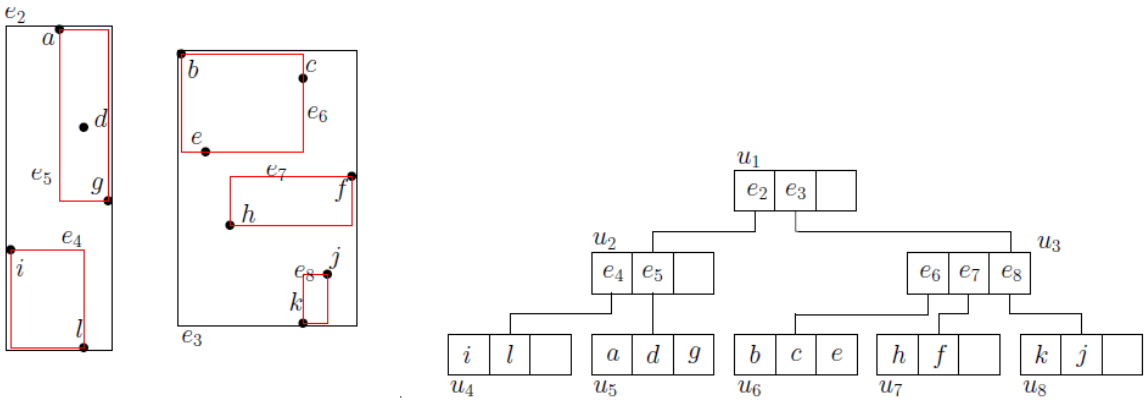
通常，我们不选择去索引几何物体本身，而是采用最小限定箱 MBB（minimum bounding box) 作为不规则几何图形的 key 来构建空间索引。在二维空间中，我们称之为最小边界矩形MBR(minimum bounding retangle)。叶子节点（leaf node）则存储每个对象需要的数据，一般是一个外接矩形和指向数据的标识符(Index，Obj_ID)。如果是点数据，叶子节点只需要存储这些点本身。如果是多边形数据，一般的做法是在叶子节点中存储多边形的最小外接矩形和指向这个多边形的数据的唯一标识符。而非叶子节点（non-leaf node）上的每一条数据由指向子节点的标识符和该子节点的外接矩形组成(Index，Child_Pointer)。在这里Index表示包围空间数据对象的最小外接矩形MBR。

下图是一个MBR的实例，包含了7个点。



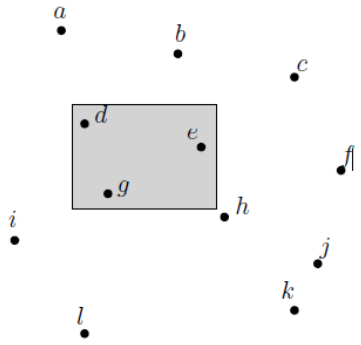
通常，只需要两个点就可限定一个矩形，也就是矩形某个对角线的两个点就可以决定一个唯一的矩形。通常使用左下，右上两个点表示或者使用右上，左下两个点来代表这个矩形。判断两个MBR是否相交即判断一个MBR的某个顶点是否处在另一个MBR所代表的范围内。

下图是R-tree构造的实例，并假定每个节点限定子节点个数为3。



2 R-tree的搜索

R-tree中的搜索的最常见情况如下，令$S = \{p_1, p_2, ..., p_n\}$为$R^2$中的一组点。给定一个与坐标轴平行的矩形Q，范围搜索将返回在S中被Q覆盖的所有点，即$S \cap Q$。如下图所示。该定义可以扩展到任何维度。



2.1 范围搜索

在范围搜索（Range query）中，输入为搜索矩形（查询框），返回与搜索矩形相重叠的实体。

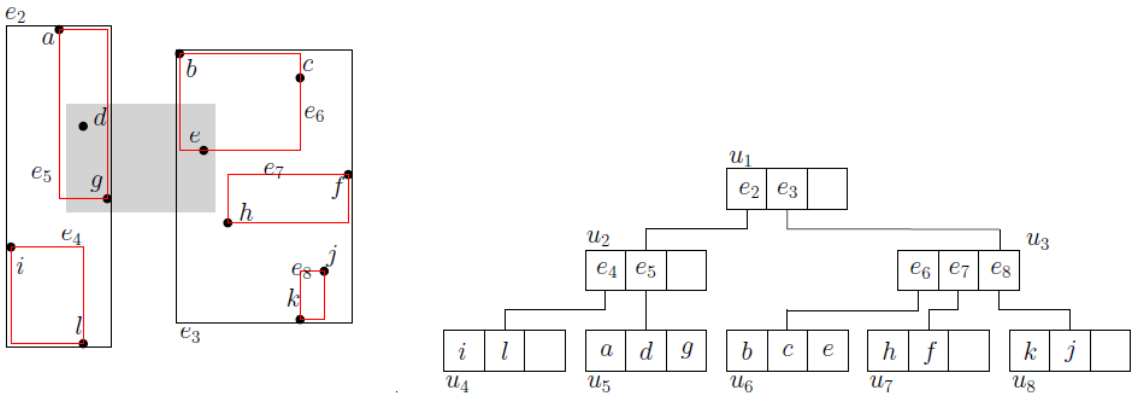
从根节点开始，每个非叶子节点包含一系列外接矩形和指向对应子节点的指针，而叶子节点则包含空间对象的外接矩形以及这些空间对象（或者指向它们的指针）。对于搜索路径上的每个节点，遍历其中的外接矩形，如果与搜索框相交就深入对应的子节点继续搜索。这样递归地搜索下去，直到所有相交的矩形都被访问过为止。如果搜索到一个叶子节点，则尝试比较搜索框与它的外接矩形和数据（如果有的话），如果该叶子节点在搜索框中，则把它加入搜索结果。

假定Q为我们的搜索范围，R-tree的根节点为root。我们查询的语句为range-query(root; q)。

Algorithm `range-query`(u, r)

1. **if** u is a leaf **then**
2. report all points stored at u that are covered by r
3. **else**
4. **for** each child v of u **do**
5. **if** $MBR(v)$ intersects r **then**
6. `range-query`(v, r)

如下灰色区域是进行的查询Q。节点u_1, u_2, u_3, u_5, u_6是查询访问过的节点。



2.2 最近邻搜索

对于最近邻搜索(nearest neighbor search)来说，可以查询一个矩形或者一个点。 先把根节点加入优先队列，然后查询优先队列中离查询的矩形或者点最近的项，直到优先队列被清空或者已经得到要求的邻居数。从优先队列顶端取出每个节点时先将其展开，然后把它的子节点插回优先队列。如果取出的是子节点就直接加入搜索结果。

3 R-tree的构造

通常，R树的构造算法旨在最小化所有MBR的周长或面积总和。

插入节点时，算法从树的根节点开始递归地向下遍历。检查当前节点中所有外接矩形，并启发式地选择在哪个子节点中插入（例如选择插入后外接矩形扩张最小的那个子节点），然后进入选择的那个子节点继续检查，直到到达叶子节点。满的叶子节点应该在插入之前分割，所以插入时到达的叶子节点一定有空位来写数据。由于把整棵树遍历一遍代价太大，在分割叶子节点时应该使用启发式算法。把新分割的节点添加进上一层节点，如果这个操作填满了上一层，就继续分割，直到到达根节点。如果根节点也被填满，就分割根节点并创建一个新的根节点，这样树就多了一层。

Algorithm `insert`(u, p)

1. **if** u is a leaf node **then**
2. add p to u
3. **if** u overflows **then**
   /* namely, u has $B+1$ points */
4. `handle-overflow`(u)
5. **else**
6. $v \leftarrow$ `choose-subtree`(u, p)
   /* which subtree under u should we insert p into? */
7. `insert`(v, p)

插入算法在树的每一层都要决定把新的数据插入到哪个子树中。经典R树的实现会把数据插入到外接矩形需要增长面积最小的那个子树。

4 STR R-tree的构造

在上面所示的R-tree的数据结构可以看出，R-tree存在着以下的缺点：
（1）建立索引花费的代价高昂：使用传统的插入函数来建立索引树，会涉及到一系列节点的分裂，子节点的重新分布，以及矩形的求交等。
（2）内存空间利用率不高：由于R树索引中每个节点不一定被子节点填满，这导致它的树高有时过大，这在数据量很大时表现的比较明显。
（3）当对索引多次进行插入或者删除的操作后，同层节点间的交叠大。

针对如上的这些缺点，研究者针对数据极少变动的静态环境提出了Packing的思想。如果事先知道被索引的全部数据，可以针对空间对象的空间分布特性按照某些规则进行分组，凡是在同一组的数据节点作为同一父节点的孩子，这样就可以减少节点之间的交叠，优化查询的性能。

Packing-R-tree 的建立算法存在着通式，可以归纳为：
（1）对输入的r个矩形进行预处理，根据某种规则对r个矩形进行分组，使每个分组包含的矩形数目为节点的最大的容量m。
（2）将每个分组作为一个父节点，于是这些分组便可产生$\lceil r/m \rceil$个父节点。
（3）将这些父节点作为输入，递归地调用索引建立过程，向上生成下一层节点。

Sort-Tile-Recusive（STR R-tree)是Packing-R-tree的一种形成方式，其主要思想是将数据通过垂直和水平切片（slice）进行分组。

假设数据空间中存在着r个物体，每个节点最多可以容纳的矩形个数是m。首先用大约$\sqrt{r/m}$个垂直的切片分割整个数据，每个切片大概含有$\sqrt{r/m}$数据矩形，其中最后一个切片可能含有的矩形数目少于$\sqrt{r/m}$个，然后对每个切片在横向分割$\sqrt{r/m}$份，使每份大概含有m个数据矩形，每大概m数据矩形的集合并生成一个父节点。当数据矩形的上层父节点全部生成好后，作为新的数据矩形输入，如此递归地生成索引结构，直到根节点。由于STR 索引从全局上根据空间的邻接性对空间数据进行了分组处理，这使它的检索性能是目前空间索引中比较优秀的。

可以看出，当输入矩形为 r 个时它的 STR 索引高度为$\lceil log_m r \rceil$, 除了最后一个slice，所有的节点都被完全填满，这样最大化地利用了空间。

这里给出一份改良过的STR R-tree构建方式。

```
    public RTree createRTree(List<Point> points, int maxPointPerEntry) throws Exception {
    // calculate number of nodes we have
    int totalLeaf = (int) Math.ceil(points.size() * 1.0 / maxPointPerEntry);
    int[] dim = new int[2];
    dim[0] = (int)Math.ceil(Math.sqrt(totalLeaf));
    dim[1] = (int)totalLeaf/dim[0];

    // Pack leaf nodes into non-leaf nodes. Pack bottom - up until we get only one root node
    List<RTreeNode> nodes = (List)packPointLeafNodes(points, dim,0);
    for(RTreeNode node : nodes){
        System.out.println(node.toString());
        System.out.println(node.getMbr().toString());
    }
    do {
        int totalEntry = (int) Math.ceil(nodes.size() *1.0 / maxPointPerEntry);
        int[] dim2 = new int[2];
        dim2[0] = (int)Math.ceil(Math.sqrt(totalEntry));
        dim2[1] = (int)totalEntry/dim2[0];

        nodes = packRTreeNodes(nodes, dim2,0);
        System.out.println("nodes size: " + nodes.size());
        for(RTreeNode node : nodes){
            System.out.println(node.getMbr().toString());
        }

    } while (nodes.size() != 1);
    return new RTree(nodes.get(0));
}
    public List<RTreeNode> packRTreeNodes(List<RTreeNode> data, int[] nbMBRs, final int currentDimension) {
    // sort data first
    Collections.sort(data, new Comparator<RTreeNode>() {
        @Override
        public int compare(RTreeNode o1, RTreeNode o2) {
            return o1.getMbr().compare(o2.getMbr(), currentDimension);
        }
    });

    int nbPointPerMBR = (int) Math.ceil(data.size() * 1.0 / nbMBRs[currentDimension]);
    int currentNbMBR = (int) Math.ceil(data.size() * 1.0 / nbPointPerMBR);

    List<RTreeNode> result = new ArrayList<RTreeNode>();
    if (currentDimension == 1) {
        // For each group i, we create a MBR
        for (int i = 0; i < currentNbMBR; i++) {
            // calculate index of group in data
            int startIndex = i * nbPointPerMBR;
            int endIndex = (i + 1) * nbPointPerMBR;
            if(i == currentNbMBR - 1){
                endIndex = data.size();
            }
```

```
            NonLeafNode node = new NonLeafNode();
            for(int j = startIndex; j< endIndex; j++){
                try {
                    node.insert(data.get(j));
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
            result.add(node);
        }
    } else {

        // For each group i, we create a MBR
        for (int i = 0; i < currentNbMBR; i++) {
            // calculate index of group in data
            int startIndex = i * nbPointPerMBR;
            int endIndex = (i + 1) * nbPointPerMBR;

            if(i == currentNbMBR -1){
                endIndex = data.size();
            }

            //  we use endIndex +1 because endIndex is exclusive in sublist
            List<RTreeNode> group = data.subList(startIndex, endIndex);
            List<RTreeNode> leafNodes = packRTreeNodes(group, nbMBRs, currentDimension + 1);
            result.addAll(leafNodes);
        }
    }
    return result;
}

public List<PointLeafNode> packPointLeafNodes(List<Point> data, int[] totalLeaf, final int currentDimension)
    System.out.println("currentDimension: " + currentDimension);
    // sort data first
    Collections.sort(data, new PointComparator(currentDimension));
    int maxPointPerEntry = (int) Math.ceil(data.size() * 1.0 / totalLeaf[currentDimension]);
    int curTotalLeaf = (int) Math.ceil(data.size() * 1.0 / maxPointPerEntry);
    System.out.println("maxPointPerEntry: " + maxPointPerEntry);
    System.out.println("curTotalLeaf: " + curTotalLeaf);

    List<PointLeafNode> result = new ArrayList<PointLeafNode>();
    if (currentDimension == 1) {
        // For each group i, we create a MBR
        for (int i = 0; i < curTotalLeaf; i++) {
            // calculate index of group in data
            int startIndex = i * maxPointPerEntry;
            int endIndex = (i + 1) * maxPointPerEntry;
            if(i == curTotalLeaf - 1){
                endIndex = data.size();
            }
```

```
            PointLeafNode leafNode = new PointLeafNode();
            for(int j = startIndex; j< endIndex; j++){
                leafNode.addPoint(data.get(j));
            }
            result.add(leafNode);
        }
    } else {

        // For each group i, we create a MBR
        for (int i = 0; i < curTotalLeaf; i++) {
            // calculate index of group in data
            int startIndex = i * maxPointPerEntry;
            int endIndex = (i + 1) * maxPointPerEntry;

            if(i == curTotalLeaf -1){
                endIndex = data.size();
            }

            //  we use endIndex +1 because endIndex is exclusive in sublist
            List<Point> group = data.subList(startIndex, endIndex);

            List<PointLeafNode> leafNodes = packPointLeafNodes(group, totalLeaf,currentDimension + 1);
            result.addAll(leafNodes);
        }
    }

    return result;
}
```