R Tree

杨嘉成 林伟鸿 谭博文

June 1, 2016

Contents

1 简介

R 树 [1,2] 是一种可以很好地解决高维空间中搜索问题的数据结构。

举个例子，查找 20 公里以内所有的餐厅。一般情况下我们会把餐厅的坐标 (x, y) 分为两个字段存放在数据库中，一个字段记录经度，另一个字段记录纬度。这样的话我们就需要遍历所有的餐厅获取其位置信息，然后计算是否满足要求。如果一个地区有 100 家餐厅的话，我们就要进行 100 次位置计算操作了，如果应用到谷歌地图这种超大数据库中，这种方法便必定不可行了。

R 树就很好的解决了这种高维空间搜索问题。它把 B 树的思想很好的扩展到了多维空间，采用了 B 树分割空间的思想，并在添加、删除操作时采用合并、分解结点的方法，保证树的平衡性。因此，R 树就是一棵用来存储高维数据的平衡树。建议读者阅读本篇报告之前先了解 B 树。

2 基本形态

如上所述，R 树是 B 树在高维空间的扩展，是一棵平衡树。每个 R 树的叶子结点包含了多个指向不同数据的指针，这些数据可以是存放在硬盘中的，也可以是存在内存中。根据 R 树的这种数据结构，当我们需要进行一个高维空间查询时，我们只需要遍历少数几个叶子结点所包含的指针，查看这些指针指向的数据是否满足要求即可。这种方式使我们不必遍历所有数据即可获得答案，效率显著提高。下图是 R 树的一个简单实例：

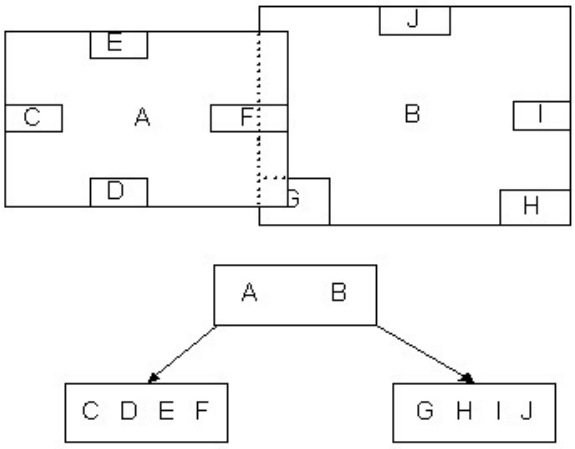
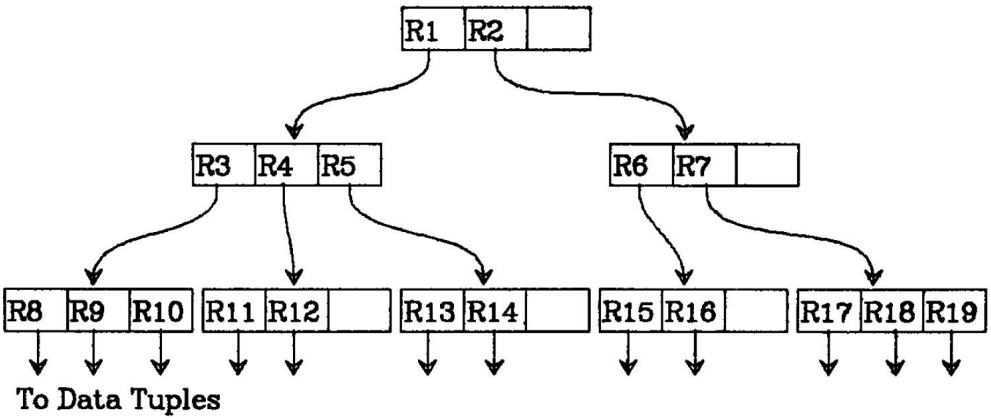
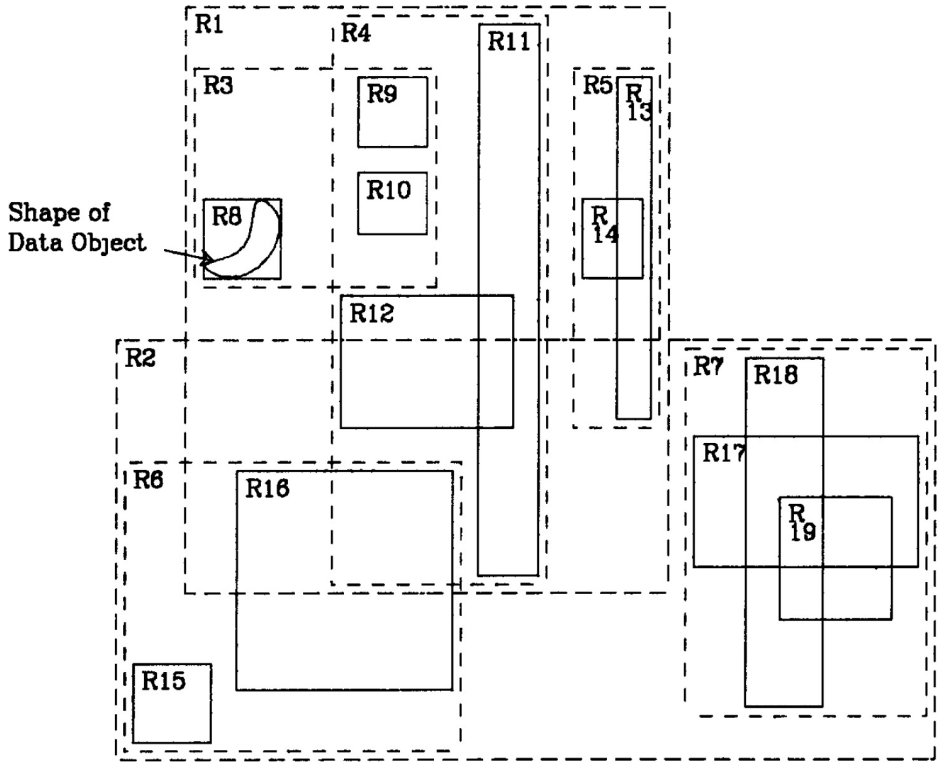


图 1: 一个 R 树的实例

我们在上面说过，R 树运用了空间分割的理念，这种理念是如何实现的呢？R 树采用了一种称为 MBR(Minimal Bounding Rectangle) 的方法，在此我把它译作“最小边界矩形”（这里的矩形指高维意义下的矩形）。从叶子结点开始用矩形 (rectangle) 将空间框起来，结点越往上，框住的空间就越大，以此对空间进行分割。我们就拿二维空间来举例。下图是 Guttman 论文 [2] 中的一幅图：



(a)



(b)

图 2: 一个较为复杂的实例及其对应的 R 树

详细解释一下这张图。先来看图2(b)：

1. 首先我们假设所有数据都是二维空间下的点，图中仅仅标志了 $R8$ 区域中的数据，也就是那个 shape of data object。别把那一块不规则图形看成一个数据，我们把它看作是多个数据围成的一个区域。为了实现 R 树结构，我们用一个

最小边界矩形恰好框住这个不规则区域，这样，我们就构造出了一个区域：$R8$。$R8$ 的特点很明显，就是正正好好框住所有在此区域中的数据。其他实线包围住的区域，如 $R9$，$R10$，$R12$ 等都是同样的道理。这样一来，我们一共得到了 12 个最最基本的最小矩形。这些矩形都将被存储在子结点中。

2. 下一步操作就是进行高一层次的处理。我们发现 $R8$，$R9$，$R10$ 三个矩形距离最为靠近，因此就可以用一个更大的矩形 $R3$ 恰好框住这 3 个矩形。

3. 同样道理，$R15$，$R16$ 被 $R6$ 恰好框住，$R11$，$R12$ 被 $R4$ 恰好框住，等等。所有最基本的最小边界矩形被框入更大的矩形中之后，再次迭代，用更大的框去框住这些矩形。

用地图的例子来解释，就是所有的数据都是餐厅所对应的地点，先把相邻的餐厅划分到同一块区域，划分好所有餐厅之后，再把邻近的区域划分到更大的区域，划分完毕后再次进行更高层次的划分，直到划分到只剩下两个最大的区域为止。要查找的时候就方便了。

下面就可以把这些大大小小的矩形存入我们的 R 树中去了。根结点存放的是两个最大的矩形，这两个最大的矩形框住了所有的剩余的矩形，当然也就框住了所有的数据。下一层的结点存放了次大的矩形，这些矩形缩小了范围。每个叶子结点都是存放的最小的矩形，这些矩形中可能包含有 n 个数据。

2.1 R 树的性质

1. 除非它是根结点，所有叶子结点包含有 m 至 M 个记录索引（条目）。作为根结点的叶子结点所具有的记录个数可以少于 m。通常，$m = M/2$；

2. 对于所有在叶子中存储的记录（条目），I 是最小的可以在空间中完全覆盖这些记录所代表的点的矩形（注意：此处所说的“矩形”是可以扩展到高维空间的）；

3. 每一个非叶子结点拥有 m 至 M 个孩子结点，除非它是根结点；

4. 对于在非叶子结点上的每一个条目，i 是最小的可以在空间上完全覆盖这些条目所代表的店的矩形（同性质 2）；

5. 所有叶子结点都位于同一层，因此 R 树为平衡树。

2.2 叶子结点的结构

叶子结点所保存的数据形式为：(I, tId) 其中，tId 表示的是一个存放于数据库中的 Tuple，即高维空间中的点，也就是一条记录，它是 n 维的。I 是一个 n 维空间的矩形，并可以恰好框住这个叶子结点中所有记录代表的 n 维空间中的点。$I = (I_0, I_1, \ldots, I_{n-1})$。其结构如下图所示：

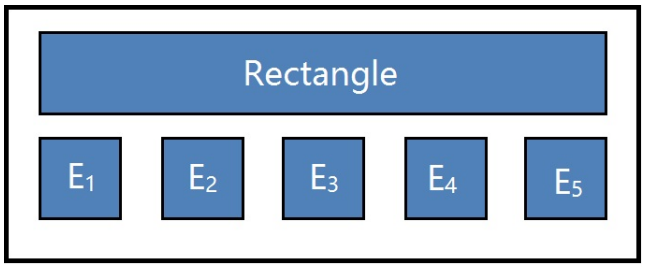


图 3: 存储结构

下图描述的就是在二维空间中的叶子结点所要存储的信息：

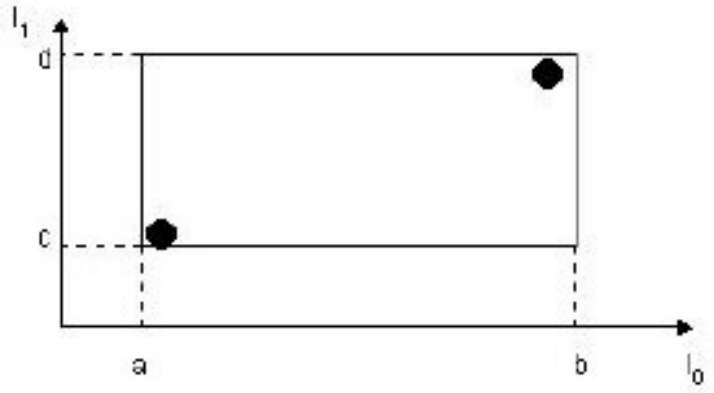


图 4: 存储信息

在这张图中，I 所代表的就是图中的矩形，其范围是 $a \leq I_0 \leq b\, c \leq I_1 \leq d$。有两个 tId，在图中即表示为那两个点。这种形式完全可以推广到高维空间。大家简单想想三维空间中的样子就可以了。

2.3 非叶子结点

非叶子结点的结构其实与叶子结点非常类似。想象一下 B 树就知道了，B 树的叶子结点存放的是真实存在的数据，而非叶子结点存放的是这些数据的“边界”，或者说也算是一种索引。

同样道理，R 树的非叶子结点存放的数据结构为：(I, P)。

其中，P 是指向孩子结点的指针，I 是覆盖所有孩子结点对应矩形的矩形。

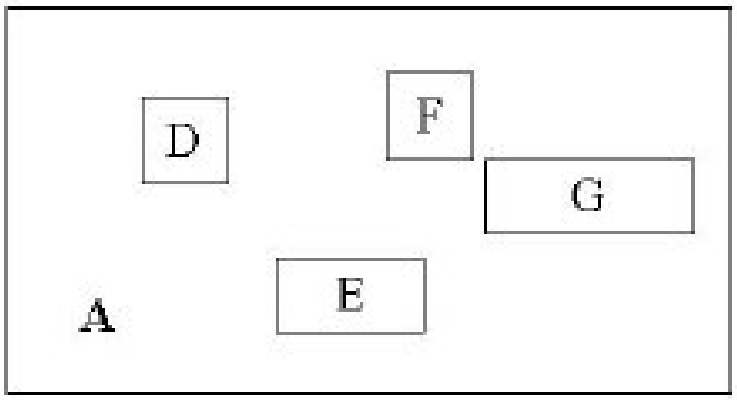


图 5: 一个示例

在上图中，D, E, F, G 为孩子结点所对应的矩形。A 为能够覆盖这些矩形的更大的矩形。这个 A 就是这个非叶子结点所对应的矩形。无论是叶子结点还是非叶子结点，它们都对应着一个矩形。树形结构上层的结点所对应的矩形能够完全覆盖它的孩子结点所对应的矩形。根结点也唯一对应一个矩形，而这个矩形是可以覆盖所有我们拥有的数据信息在空间中代表的点的。

3 具体实现

3.1 查找

R 树的搜索操作很简单，跟 B 树上的搜索十分相似。它返回的结果是所有符合查找信息的记录条目。而输入是什么？通常而言，输入不仅仅是一个范围了，它更可以看成是一个空间中的矩形。也就是说，我们输入的是一个搜索矩形。先给出伪代码：

Algorithm 1 查找函数

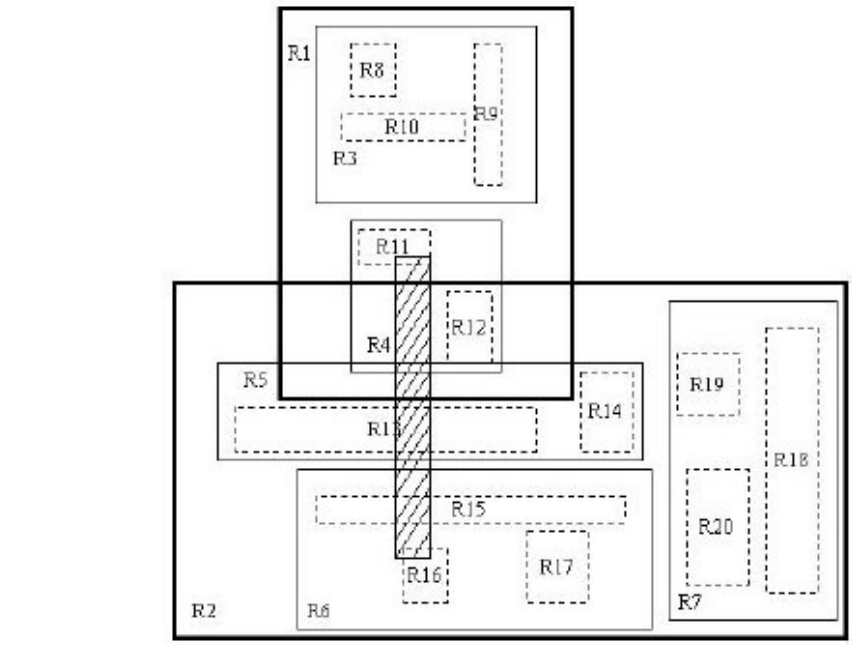
Input: x, Rectangle
Output: PointSet

```
1: function SEARCH(x, Rectangle)
2:     if Rectangle ∈ LeftChild(x) then
3:         left ← SEARCH(LeftChild(x), Rectangle)
4:     end if
5:     if Rectangle ∈ RightChild(x) then
6:         right ← SEARCH(RightChild(x), Rectangle)
7:     end if
8:     return left + right
9: end function
```

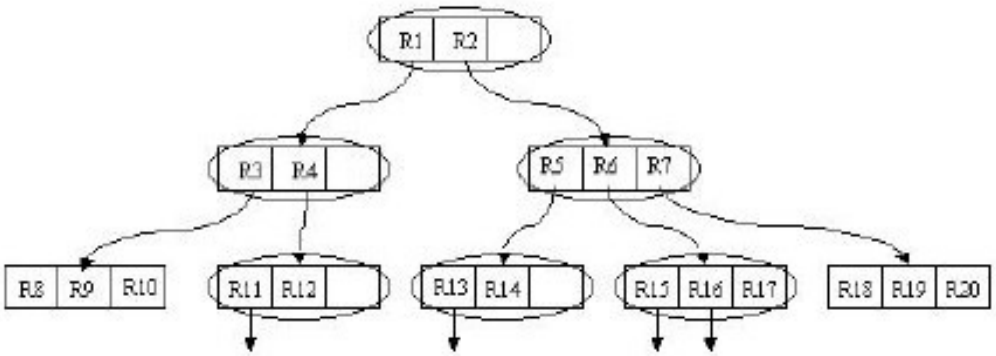
其步骤大致为：

1. 如果 T 是非叶子结点，如果 T 所对应的矩形与 S 有重合，那么检查所有 T 中存储的条目，对于所有这些条目，使用 Search 操作作用在每一个条目所指向的子树的根结点上（即 T 结点的孩子结点）。
2. 如果 T 是叶子结点，如果 T 所对应的矩形与 S 有重合，那么直接检查 S 所指向的所有记录条目。返回符合条件的记录。

我们通过下图来理解这个 Search 操作。



(a)



(b)

图 6: 一个查询的实例及其对应的 R 树

阴影部分所对应的矩形为搜索矩形。它与根结点对应的最大的矩形（未画出）有重叠。这样将 Search 操作作用在其两个子树上。两个子树对应的矩形分别为 R1 与 R2。搜索 R1，发现与 R1 中的 R4 矩形有重叠，继续搜索 R4。最终在 R4 所包含的 R11 与 R12 两个矩形中查找是否有符合条件的记录。搜索 R2 的过程同样如此。很显然，该算法进行的是一个迭代操作。

3.2 维护

3.2.1 插入

R 树的插入操作也同 B 树的插入操作类似。当新的数据记录需要被添加入叶子结点时，若叶子结点溢出，那么我们需要对叶子结点进行分裂操作。显然，叶子结点的插入操作会比搜索操作要复杂。插入操作需要一些辅助方法才能够完成。

来看一下伪代码：

Algorithm 2 插入函数

Input: x, value

```
1:  procedure INSERT(x, value)
2:      if x is Leaf then
3:          INSERTINTOLEAF(x, value)
4:          if x has no space then
5:              SPLITLEAF(x)
6:          end if
7:      end if
8:      direction ← CHOOSELEAF(x, value) ADJUSTTREE(x)
9:      if x.child[direction] has been splitted then
10:         ADJUSTTREE(x.child[direction])
11:     end if
12: end procedure
```

其步骤大致为：

1. 为新记录找到合适插入的叶子结点，开始 ChooseLeaf 方法选择叶子结点 L 以放置记录 E。
2. 添加新记录至叶子结点, 如果 L 有足够的空间来放置新的记录条目，则向 L 中添加 E。如果没有足够的空间，则进行 SplitNode 方法以获得两个结点 L 与 LL，这两个结点包含了所有原来叶子结点 L 中的条目与新条目 E。
3. 将变换向上传递，开始对结点 L 进行 AdjustTree 操作，如果进行了分裂操作，那么同时需要对 LL 进行 AdjustTree 操作。
4. 对树进行增高操作，如果结点分裂，且该分裂向上传播导致了根结点的分裂，那么需要创建一个新的根结点，并且让它的两个孩子结点分别为原来那个根结点分裂后的两个结点。

接下来是函数 ChooseLeaf，这个函数是用来选择往哪一个孩子插入的函数，其伪代码为：

Algorithm 3 选择儿子函数

Input: x, value

```
1: function ChooseLeaf(x, value)
2:     N ← x
3:     while N is not a leaf do
4:         LDelta ← GetDelta(LeftChild(N), value)
5:         RDelta ← GetDelta(RightChild(N), value)
6:         if LDelta < RDelta then
7:             Swap(N, LeftChild(N))
8:         else
9:             Swap(N, RightChild(N))
10:        end if
11:    end while
12:    return N
13: end function
```

其步骤大致为：

1. 初始化，设置 N 为根结点；
2. 叶子结点的检查，如果 N 为叶子结点，则直接返回 N；
3. 选择子树，如果 N 不是叶子结点，则遍历 N 中的结点，找出添加时扩张最小的结点，并把该结点定义为 F。如果有多个这样的结点，那么选择面积最小的结点；
4. 下降至叶子结点，将 N 设为 F，从第二步开始重复操作。

最后一个函数是调整树，其伪代码为：

Algorithm 4 树调整函数

Input: x

```
1: procedure ChooseLeaf(x)
2:     N ← x
3:     P ← Parent(x)
4:     while N is not a root do
5:         LDelta ← GetDelta(LeftChild(N))
6:         RDelta ← GetDelta(RightChild(N))
7:         if LDelta < RDelta then
8:             Swap(N, LeftChild(N))
9:         else
10:            Swap(N, RightChild(N))
11:        end if
12:    end while
13: end procedure
```

其步骤大致为：

1. 初始化，将 N 设为 x；

2. 检验是否完成，如果 N 为根结点，则停止操作；

3. 调整父结点条目的最小边界矩形，设 P 为 N 的父节点，EN 为指向在父节点 P 中指向 N 的条目，调整 $EN.I$ 保证所有在 N 中的矩形都被恰好包围；

4. 向上传递结点分裂，如果 N 有一个刚刚被分裂产生的结点 NN，则创建一个指向 NN 的条目 ENN。如果 P 有空间来存放 ENN，则将 ENN 添加到 P 中。如果没有，则对 P 进行 SplitNode 操作以得到 P 和 PP；

5. 升高至下一级，如果 N 等于 L 且发生了分裂，则把 NN 置为 PP，从第二步开始重复操作。

同样，我们用图来更加直观的理解这个插入操作：

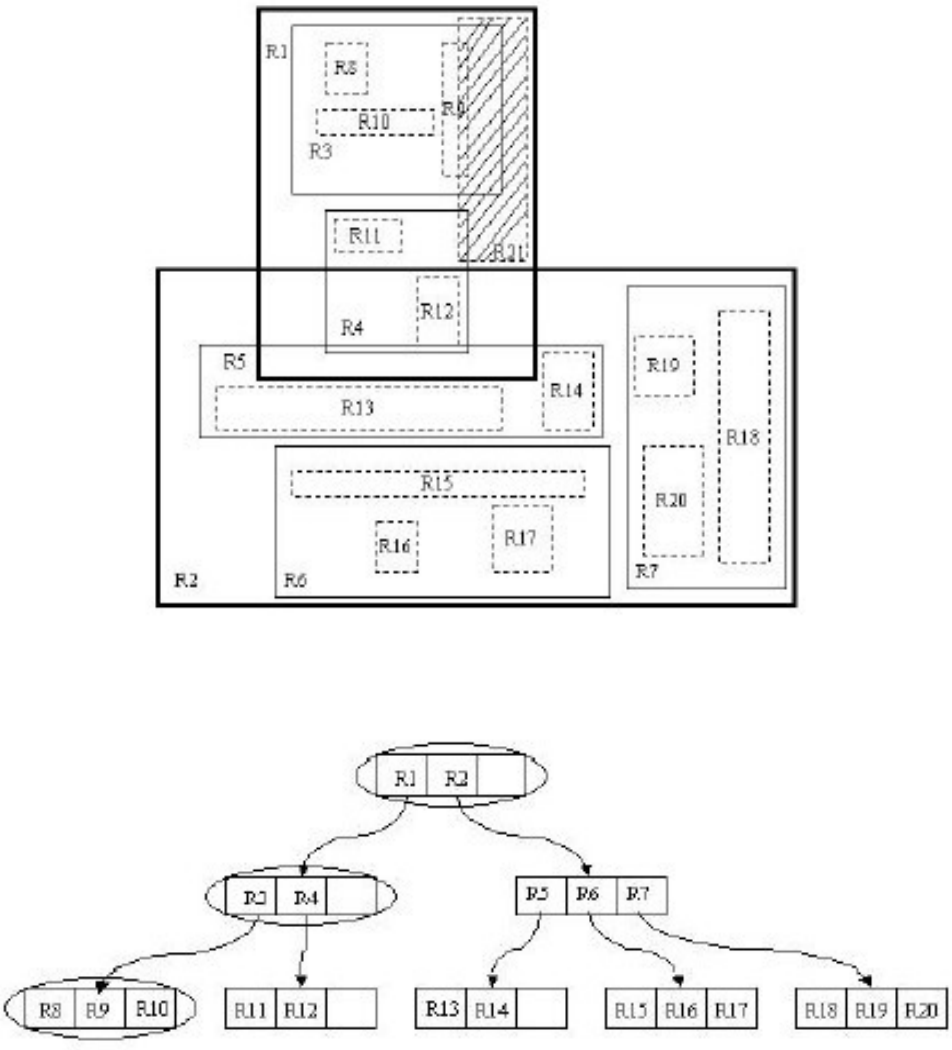


图 7: 一个查询的实例及其对应的 R 树

我们来通过图分析一下插入操作。现在我们需要插入 R21 这个矩形。开始时我们进行 ChooseLeaf 操作。在根结点中有两个条目，分别为 R1，R2。其实 R1 已经完全覆盖了 R21，而若向 R2 中添加 R21，则会使 R2.I 增大很多。显然我们选择 R1 插入。然后进行下一级的操作。相比于 R4，向 R3 中添加 R21 会更合适，因为 R3 覆盖 R21 所需增大的面积相对较小。这样就在 B8，B9，B10 所在的叶子结点中插入 R21。由于叶子结点没有足够空间，则要进行分裂操作。

插入操作如下图所示：

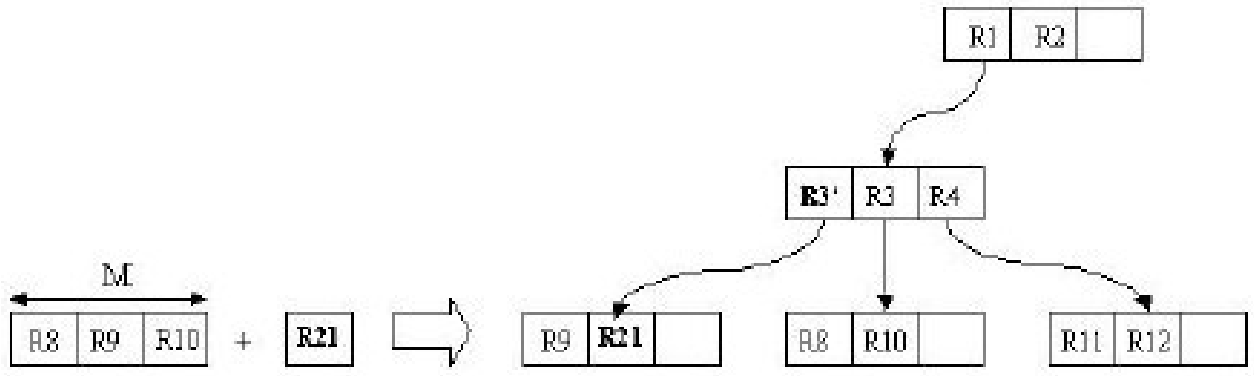


图 8: 插入操作示意图

这个插入操作其实类似于第一节中 B 树的插入操作，这里不再具体介绍，不过想必看过上面的伪代码大家应该也清楚了。

3.2.2 删除

R 树的删除操作与 B 树的删除操作会有所不同，不过同 B 树一样，会涉及到压缩等操作。相信读者看完以下的伪代码之后会有所体会。R 树的删除同样是比较复杂的，需要用到一些辅助函数来完成整个操作。

Delete 函数的步骤大致如下：

1. 找到含有记录的叶子结点，使用 FindLeaf 方法找到包含有记录 E 的叶子结点 L。如果搜索失败，则直接终止；
2. 删除记录，将 E 从 L 中删除；
3. 传递记录，对 L 使用 CondenseTree 操作；
4. 缩减树，当经过以上调整后，如果根结点只包含有一个孩子结点，则将这个唯一的孩子结点设为根结点。

下面是 FindLeaf 操作，其大致步骤为：

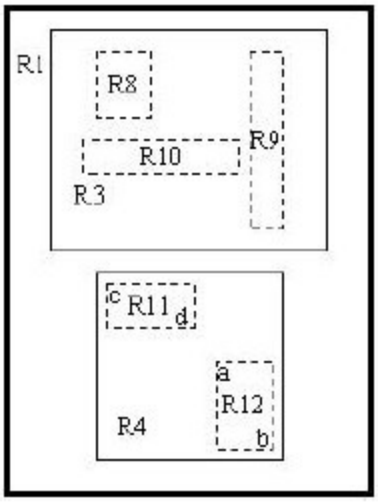
1. 搜索子树，如果 T 不是叶子结点，则检查每一条 T 中的条目 F，找出与 E 所对应的矩形相重合的 F（不必完全覆盖）。对于所有满足条件的 F，对其指向的孩子结点进行 FindLeaf 操作，直到寻找到 E 或者所有条目均以被检查过；
2. 搜索叶子结点以找到记录，如果 T 是叶子结点，那么检查每一个条目是否有 E 存在，如果有则返回 T；

下面是 CondenseTree 操作，L 为包含有被删除条目的叶子结点。如果 L 的条目数过少（小于要求的最小值 m），则必须将该叶子结点 L 从树中删除。经过这一删除操作，L 中的剩余条目必须重新插入树中。此操作将一直重复直至到达根结点。同样，调整在此修改树的过程所经过的路径上的所有结点对应的矩形大小。

其大致步骤为：

1. 初始化，令 N 为 L。初始化一个用于存储被删除结点包含的条目的链表 Q；
2. 找到父条目，如果 N 为根结点，那么直接跳转至 CT6。否则令 P 为 N 的父结点，令 EN 为 P 结点中存储的指向 N 的条目；
3. 删除下溢结点，如果 N 含有条目数少于 m，则从 P 中删除 EN，并把结点 N 中的条目添加入链表 Q 中；
4. 调整覆盖矩形，如果 N 没有被删除，则调整 EN.I 使得其对应矩形能够恰好覆盖 N 中的所有条目所对应的矩形；
5. 向上一层结点进行操作，令 N 等于 P，从 CT2 开始重复操作；
6. 重新插入孤立的条目，所有在 Q 中的结点中的条目需要被重新插入。原来属于叶子结点的条目可以使用 Insert 操作进行重新插入，而那些属于非叶子结点的条目必须插入删除之前所在层的结点，以确保它们所指向的子树还处于相同的层；
7. R 树删除记录过程中的 CondenseTree 操作是不同于 B 树的。我们知道，B 树删除过程中，如果出现结点的记录数少于半满（即下溢）的情况，则直接把这些记录与其他叶子的记录“融合”，也就是说两个相邻结点合并。然而 R 树却是直接重新插入；

同样，我们用图直观的说明这个操作。



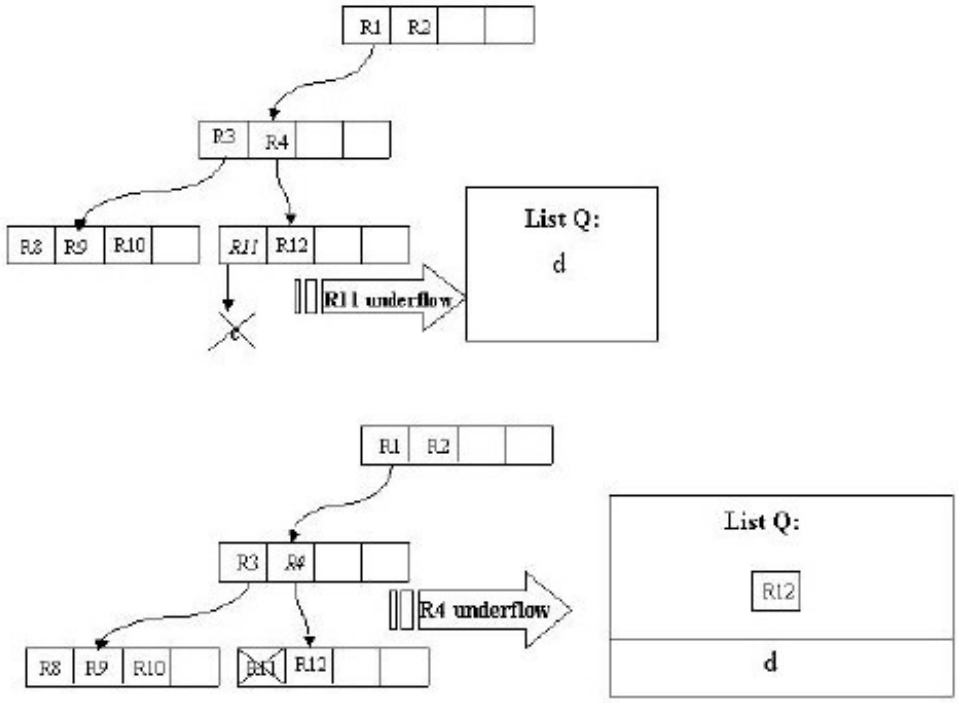


图 9: 删除操作示意图

假设结点最大条目数为 4，最小条目数为 2。在这张图中，我们的目标是删除记录 c。首先使用 FindLeaf 操作找到 c 所处在的叶子结点的位置——R11。当 c 从 R11 删除时，R11 就只有一条记录了，少于最小条目数 2，出现下溢，此时要调用 CondenseTree 操作。这样，c 被删除，R11 剩余的条目——指向记录 d 的指针——被插入链表 Q。然后向更高一层的结点进行此操作。这样 R12 会被插入链表中。原理是一样的，在这里就不再赘述。

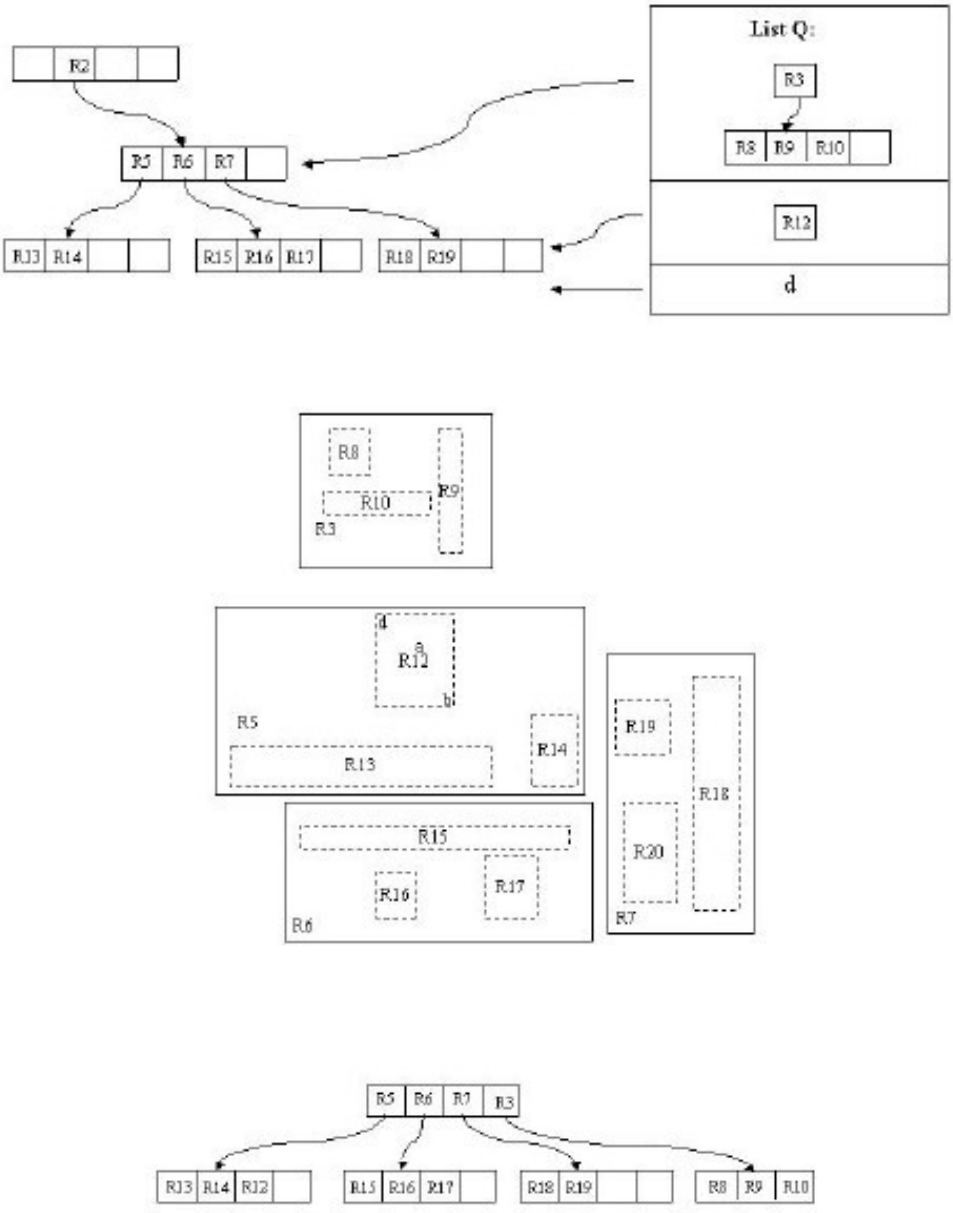


图 10: 一个删除的实例

有一点需要解释的是，我们发现这个删除操作向上传递之后，根结点的条目 R1 也被插入了 Q 中，这样根结点只剩下了 R2。别着急，重新插入操作会有效的解决这个问题。我们插入 R3，R12，d 至它原来所处的层。这样，我们发现根结点只有一个条目了，此时根据 Inert 中的操作，我们把这个根结点删除，它的孩子结点，即 R5，R6，R7，R3 所在的结点被置为根结点。至此，删除操作结束。

4 复杂度

显然包含 N 个节点的 R 树的高度是 $\lceil \log_m N \rceil - 1$ 的，并且节点的最大数量是 $\sum_{i=1} \lceil \frac{N}{m^i} \rceil + 1$，因此在查找的最坏复杂度是 $O(N \log N)$ 的、删除的复杂度是 $O(\log N)$ 的，而由于插入操作的特殊性（可能会引起其他节点分裂），不难看出插入操作的最坏复杂度为 $O(N)$，但是在平均意义下，插入操作的复杂度是 $O(\log N)$ 的。关于空间复杂度，不难看出这和树高还有插入数据量的乘积成正比，因此 R 树的空间复杂度是 $O(N \log N)$。

5 用途

R 树是一种用途广泛的数据结构，除了可以对空间中的点集进行查询外，这种数据结构还在大型数据库索引中起着至关重要的作用。

References

[1] csdn 博客, http://blog.csdn.net/zhouxuguang236/article/details/7898272. *R 树空间索引*.

[2] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. *Proceedings of the 1984 ACM SIGMOD international conference on Management of data – SIGMOD '84*, 1984.