R-Tree

May 31, 2016

Contents

1 功能介绍以及现实生活中的具体应用

$R-tree$可以很好的解决在高维空间的搜索等问题，在数据库等领域做出的功绩是非常显著的。$R-tree$中的R应该代表的是$Rectangle$[1]。举个R树在现实领域中能够解决的例子：查找20英里以内所有的餐厅。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，这种方法当然不可行。

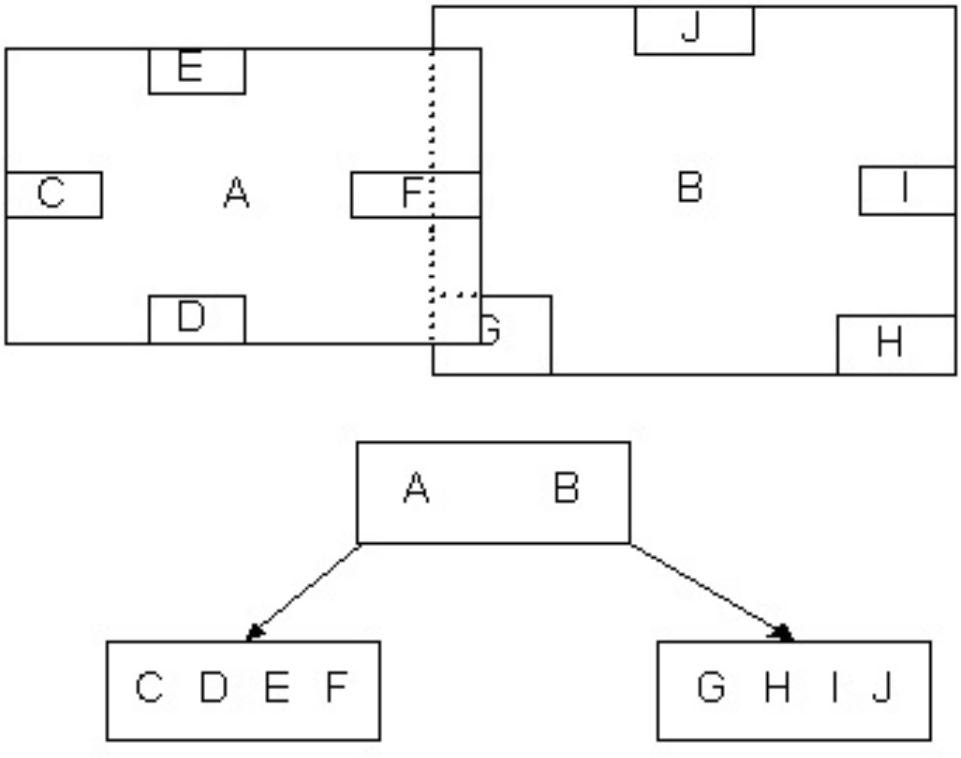
$R-tree$就很好的解决了这种高维空间搜索问题。它把B树的思想很好的扩展到了多维空间，采用了B树分割空间的思想，并在添加、删除操作时采用合并、分解结点的方法，保证树的平衡性。因此，R树就是一棵用来存储高维数据的平衡树。

2 原理说明

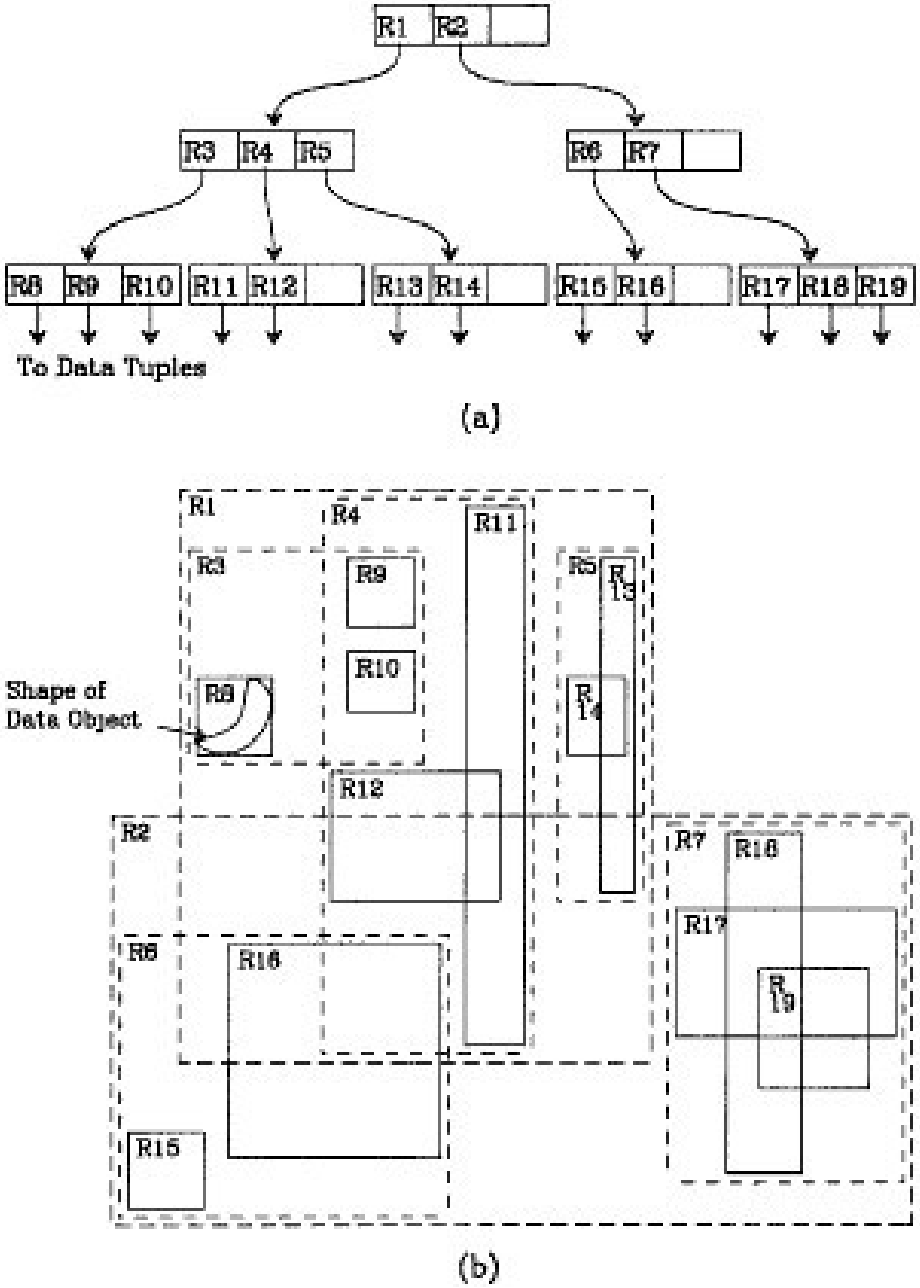
2.1 简要介绍

如功能介绍中所述，$R-tree$是B树在高维空间的扩展，是一棵平衡树。每个$R-tree$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$R-tree$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$R-tree$的一个简单实例：



$R-tree$采用了一种称为$MBR(MinimalBoundingRectangle)$的方法，在此将它译作“最小边界矩形”，来实现空间分割。从叶子结点开始用矩形将空间框起来，结点越往上，框住的空间就越大，以此对空间进行分割。以二维空间来举例，如下图[2]。



如上图（b）。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，也就是文中所示$shapeofdataobject$。我们把其中不规则区域看作是多个数据围成的一个区域。为了实现$R-tree$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，即恰好框住所有在此区域中的数据。其他实线包围住的区域，如$R9$，$R10$，$R12$等也都类似。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。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。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。

将这些矩形对应到$R-tree$中。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。

2.2 算法轮廓

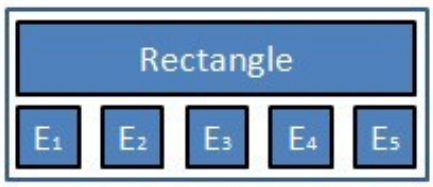
2.2.1 一棵$R-tree$满足的性质

- 除非它是根结点之外，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，$m=\frac{M}{2}$。
- 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）。
- 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。
- 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。
- 所有叶子结点都位于同一层，因此$R-tree$为平衡树。

2.2.2 叶子结点的结构

叶子结点所保存的数据形式为：$(I, tuple-identifier)$。

其中，$tuple-identifier$表示的是一个存放于数据库中的$tuple$，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。$l=(l_0, l_1, ..., l_{n-1})$。其结构如下图所示:



R树中结点的存储结构。E代表Entry，即指向孩子结点的条目

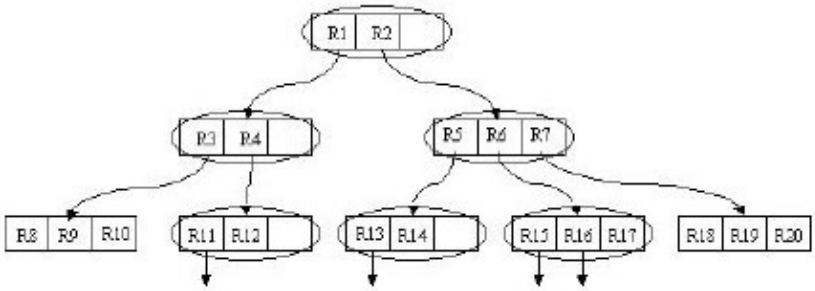
2.2.3 非叶子结点的结构

非叶子结点的结构其实与叶子结点非常类似。$R-tree$的非叶子结点存放的数据结构为：$l, child-pointer$。

其中，$child-pointer$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。

2.2.4 $Search$

$R-tree$的搜索操作很简单，它返回的结果是所有符合查找信息的记录条目。输入可以看成是一个空间中的矩形。从根结点开始搜索，如果当前结点与所查找的矩形（高维）有重合，那么就检察其所有子结点，直至叶子结点，返回所有符合条件的记录。



以上图为例，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。

2.2.5 $Insert$

当新的数据记录需要被添加入叶子结点时，先找出添加该记录后扩张最小的叶子结点，若有多个这样的结点，则选择面积最小的结点。若叶子结点溢出，那么我们需要对叶子结点进行分裂操作。最后对该结点进行$Adjust$,具体见下方$Adjust$操作以及$Split$操作说明。

2.2.6 $Adjust$

即将当前结点的改变渐渐上传至根结点以改变各个矩阵，这中间可能有$Split$操作，如果有，则假设分裂出的结点为N'，若父亲P也有足够空间放下N',则直接将N'合并进P中，否则将P分解出P',将N'并入P'，递归向上。

2.2.7 $Split$

先从要分裂的矩形中选出两个子矩形当作要分裂成的两部分的第一个元素，满足$area(J)-area(E1)-area(E2)$最大，其中$J$是覆盖$E1$和$E2$的矩形。对于剩下的元素，考虑进入哪个部分使得面积扩张更小，如果一个部分的元素已经达到$M-m+1$,则直接将剩下元素放入另一个部分。或者所有元素已经被分配，也停止。

2.2.8 $Delete$

将指定条目从叶子结点中删除后，若该叶子结点条目数过少，则将该叶子结点从树中删除，并将其剩余条目重新插入树中。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。最后，若根结点只包含一个孩子结点，将这个唯一的孩子结点设为根结点。

3 正确性分析

使用上述方法，可以在动态插入删除结点的过程中维护当前的$R-tree$，并使得它总满足$R-tree$的性质。查询正确性显然。

删除和插入过程中$R-tree$也总满足其性质，在原理中的算法轮廓部分已有说明。

4 时空复杂度证明

4.1 时间复杂度

$R-tree$兄弟结点对应的空间区域可以重叠，可以较容易地进行插入和删除操作。但正因为区域之间有重叠，空间索引可能要对多条路径进行搜索后才能得到最后的结果。当查找与给定的查询窗口相交的所有空间对象时，空间搜索算法是从根结点开始，向下搜索相应的子树．算法递归遍历所有约束矩形与查询窗口相交的子树，当到达叶结点时，边界矩形中的元素被取出并测试其是否与查询矩形相交，所有与查询窗口相交的叶结点即为要查找的空间对象。R树的查询效率会因重叠区域的增大而大大减弱，在最坏情况下，其时间复杂度甚至会由对数搜索退化成线性搜索。

对于一棵包含N个数据记录的$R-tree$, 其高度最多为$|log_m N|-1$， 因为每个叶子结点最少有m个记录，叶结点个数最多为$\lceil \frac{N}{m} \rceil + \lceil \frac{N}{m^2} \rceil + 1$。除了根结点，最坏情况下每个结点利用的空间为自身结点的$\frac{m}{M}$。当一个叶子结点包含更多数据记录时，不仅树的高度会减少，空间利用率也会提升。

由以下几幅图可知，$R-tree$的效率主要取决于$Split$操作的时间复杂度。

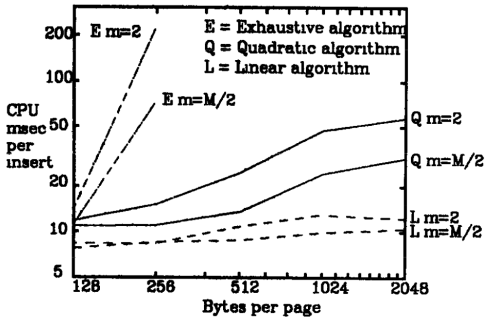


Figure 4 2
CPU cost of inserting records

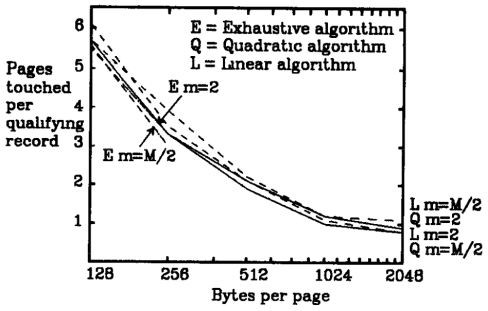


Figure 4 4
Search performance Pages touched

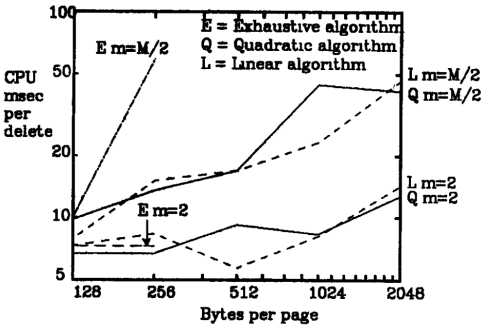


Figure 4 3
CPU cost of deleting records

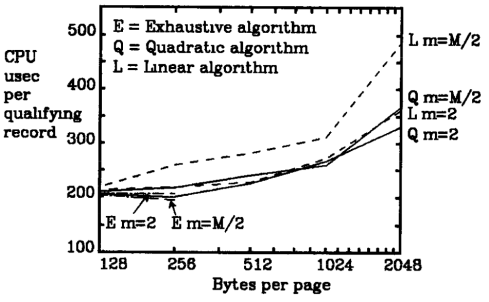


Figure 4 5
Search performance CPU cost

由于我们的$Split$算法每次需要挑选任意两个矩形计算覆盖它们的矩形大小，时间复杂度几乎接近于平方。

4.2 空间复杂度

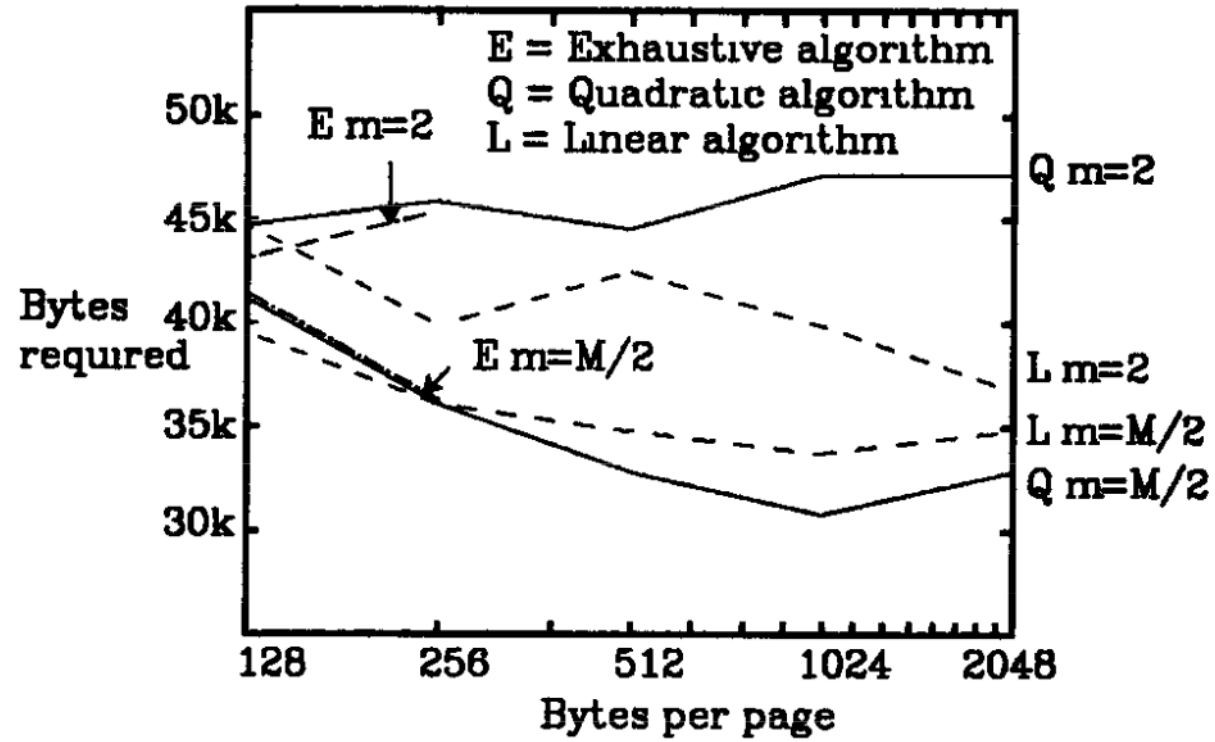


Figure 4 6
Space efficiency

所有叶子结点$O(N)$的空间，耗费空间最大的情况下，每个结点保存m个数据记录，令$r = \lceil \frac{N}{m} \rceil$, 需要$O(\lceil \frac{r}{m-1} \rceil)$的结点来存储。

5 具体实现方式及细节

以下面积、矩形所表示含义都可推广至高维。

- $Search$
  - 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）。
  - 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。
- $Insert$
  - 选择插入后面积扩张最小的叶子结点L以放置记录E，若有多个选择面积最小的。
  - 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$Split$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。

- 开始对结点L进行$Adjust$操作，如果进行了分裂操作，那么同时需要对LL进行$Adjust$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。

- $Adjust$
  - 将N设为L。
  - 如果N为根结点，则停止操作。
  - 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$EN.I$以保证所有在N中的矩形都被恰好包围。
  - 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$Split$操作以得到P和PP。
  - 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。

- $Delete$
  - 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。
  - 将E从L中删除。
  - 对L使用$Condense$操作。
  - 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。

- $Condense$
  - 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。
  - 如果N为根结点，那么直接跳转至第6步。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。
  - 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。
  - 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。
  - 令N等于P，从第二步开始重复操作。
  - 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。

6 参考文献

[1] R-tree WIKIPEDIA https://en.wikipedia.org/wiki/R-tree
[2] R-Trees: A Dynamic Index Structure for Spatial Searching Guttman, A. (1984) http://www-db.deis.unibo.it/courses/SI-LS/papers/Gut84.pdf