文章编号:1001 - 9081(2010)04 - 1103 - 04

基于 RBD-tree 的空间索引结构设计

马 绚
(华东政法大学 信息化办公室,上海 201620)
(maxuan@ecupl.edu.cn)

摘 要:在 R°-tree 索引结构的基础上提出了一种新的变种 RBD-tree 索引结构。RBD-tree 是一种基于节点密度的索引结构,以节点密度来衡量节点的性质,其核心思想是将密度相近的点组织在一起,实际上,这些密度相近的节点往往在物理上也是相近的,因此有效提高了查询效率,而且对索引结构的优化是独立于存储设备的。

关键词:地理信息系统;空间索引; R° 树;RBD 树
中图分类号: TP311 **文献标志码**:A

Design of spatial index structure based on RBD-tree

MA Xuan
(*Informatization Office, East China University of Political Science and Law, Shanghai* 201620, *China*)

Abstract: RBD-tree, a new variant index structure, derived from R°-tree was introduced in this paper. RBD-tree is an index structure based on node density which is important to measure the node quality. The core idea of RBD-tree is to organize the nodes with similar features together. In reality, the nodes with similar features are often adjacent with each other. Therefore, RBD-tree greatly improves the spatial query efficiency, and optimization of the index structure is independent of storage devices.

Key words: Geographic Information System (GIS); spatial index; R°-tree; RBD-tree

0 引言

地理信息系统(Geographic Information System,GIS)中海量空间数据的查询效率一直是 GIS 的瓶颈,因此空间数据的管理是新一代 GIS 亟待解决的重要技术问题。空间数据的快速索引是指依据空间对象的位置和形状或空间对象之间的某种空间关系,按一定顺序排列的一种数据结构。不同的空间索引结构和空间索引技术性能的优劣直接影响 GIS 系统的整体性能,它是空间数据库和 GIS 的一项关键技术。

近年来,国内外学者提出了许多不同的空间索引结构,其组织形式已经形成了一个从一维索引到多维索引的庞大体系,目前国内外主要的空间数据库广泛采用的是 R-tree[1] 索引结构和四叉树系列空间索引结构,还有很多使用了 R-tree 的变形如优先级树[2]、R*-tree[3]、R°-tree[4]、Hilbert R-tree (using fractals)[5] 或者四叉树 QuadTree、Grid、Cell-trees 等。现有的这些空间索引技术都是以磁盘作为存储设备,索引中节点信息以磁盘上的扇区为单位来存储,对节点的更新也是在对应扇区上进行操作,由于磁盘是以扇区为基本读写单位,因此这些索引结构的修改方式比较高效。

随着电信技术的快速发展,现在嵌入式设备的使用日益普遍,GIS 的发展和应用有了更广阔空间。但是,嵌入设备通常使用 Flash Memory 作为存储设备,其物理结构和读写方式与传统磁盘不同,因而嵌入设备中采用的空间索引结构必须根据 Flash Memory 的特性来对传统的空间索引结构进行优化[6]。所以本文提出了一种基于 R°-tree 的新变种 RBD-tree,旨在获得更高的查询效率,并尽力使对索引结构的优化独立于存储设备。

1 R-tree 及其变种

1.1 R-tree

R-tree 是一种高度平衡的动态索引结构,是类似 B + 树[7] 在 k 维空间的自然扩展。R-tree 包含叶节点和非叶节点,每个节点都对应一个区域和磁盘页,索引和到记录的指针都存储在叶节点中,当 R-tree 存储在磁盘上的时候,指针就是某个磁盘页的地址,在内存中就是到内存中记录首地址的指针。空间数据库用一个元组集合来标识空间实体,每一个元组具有一个唯一的标识符,叶节点包含多个形式为 (mbr, oi) 的实体,oi 是数据库中空间数据对象的标识,而 mbr 是包围该数据对象的最小边界矩形(Minimum Bound Rectangle,MBR);非叶节点则包含多个形式为 (mbr, cp) 的实体,cp 指向更低一级的孩子节点,mbr 是包围其孩子节点中所有 mbr 的最小边界矩形。由于 R-tree 的叶节点存储的是实际空间对象的最小边界矩形而不是空间对象本身,因此不可避免地导致边界矩形区域重叠。

假设 R-tree 的节点至少有 n、至多有 N 棵子树,一般 $n \leqslant N/2$,那么:每一个叶节点除非作为根节点否则含有 $n - N$ 个索引记录;对于叶节点里面的每一个(mbr, oi),每一个内部节点除非作为根节点否则含有 $n - N$ 棵子树;根节点至少含有两个孩子节点;所有叶子节点都在同一层上。图 1 给出了 R-tree 的空间结构和相应的存储结构。

R-tree 的每一个叶节点同磁盘的一页相对应,空间数据检索只需要搜索相当少的节点,而不需要遍历整棵树。R-tree 的查询、删除、插入等操作与 B 树类似,在插入或者删除的过程中索引的结构会被局部地调整和重构,而且不需要周期性的重建,所以 R-tree 是一种完全动态的空间索引数据结构。

收稿日期:2009 - 10 - 25;**修回日期**:2009 - 12 - 09。
作者简介:马绚(1976 -),女,重庆人,工程师,硕士,主要研究方向:嵌入式软件、网络。

R-tree 是一种空间利用率高、适合于外存存储的数据结构,和磁盘页相对应的节点能取得很好的性能,当节点变小作为内存索引的时候性能应该变化不大但 I/O 成本可能增加。而且由于 R-tree 允许同层节点间相互重叠,因此对精确匹配查询而言其搜索的性能取决于覆盖和交叠两个参数。

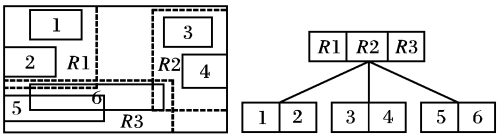


图 1 R-tree 结构

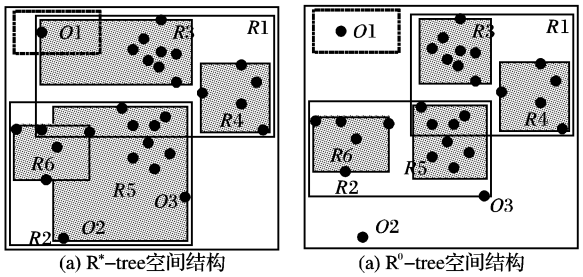
1.2 R*-tree

R*-tree 作为 R-tree 的变种继承了 R-tree 的结构,区别在于尽量减少非叶节点包围体中被包围体包含但是不被子树中任何元素包含的空闲区域;由于给定同样的面积,正方形的边界长度最短所以尽量让每一棵子树的包围体尽量接近正方形[8]以减少子树的边界长度;增加节点的空间利用率;使同级节点之间的包围体重叠应尽量小。

R*-tree 的宗旨是减少节点的覆盖区域、边界长度以及节点之间的重叠,以此来构造一棵健壮的树,而且由于采用了强制插入的方法,减少了分裂次数、动态调整树的结构并增加空间利用率,以少量增加 CPU 开销而提高了查询性能。R*-tree 在经过改良之后还衍生出了其他的版本[8-10]。

1.3 R*-tree 的变种 R°-tree

R°-tree 是基于 R*-tree 的变种,对 R*-tree 的主要改动是优化树中的孩子节点,将地理上与其他节点相距比较远的节点作为外部节点进行特殊处理,如果有一个孩子距离其他孩子节点都比较远,则把这个孩子节点放到父节点中去以减少节点包围体的空闲区域,提高空间利用率和查询效率,如图 2 所示。



(a) R*–tree空间结构 (a) R°–tree空间结构

图 2 R*-tree 和 R°-tree 空间结构组织对比

2 RBD-tree 索引结构的设计与实现

2.1 节点密度

RBD-tree 的设计思路与已有的 R-tree 索引结构不同,它是在 R°-tree 基础上提出的一种新的索引结构。R°-tree 提出了外部节点的概念,将节点中离其他孩子节点都比较远的孩子作为外部节点,然后放到上一级节点中,由此来优化节点的质量,减少节点之间的重叠区域。RBD-tree 认为在物理上相邻的节点性质类似,尽量将这些性质类似的节点划分为一个区域,因此引入节点密度的概念。

节点密度可以按照面积来计算,节点密度公式为:

$$D_R = \sum_{i=1}^{n} s_i / S_R$$

其中:D_R表示节点 R 的密度;S_R 表示节点 R 包围体的面积;n 是 R 的孩子节点个数,$\sum_{i=1}^{n} S_i$ 为 n 个孩子节点的包围体所占的面积总值。

节点密度也可以按照孩子个数计算,当一个节点其孩子个数多时节点密度就高,反之就低,节点密度公式为:

$$D_R = \frac{n}{S_R}$$

由于多个孩子节点之间包围体会有重叠,所以通常以孩子个数计算节点密度。

在基于节点密度的设计理念下,将某个空间区域中节点密度稠密区域的节点尽量划分在一起,节点密度稀疏的节点划分在一起,减少了子节点之间的重叠。这种划分节点的方法也恰好反应了现实世界空间区域的状态,例如同样大小的一个区域,与在郊区相比,在闹市区就会包含有更多的孩子节点数,也就意味着更高的节点密度。

2.2 算法描述

如前所述,R°-tree 是采用外部节点的方式来提高节点的质量,从而减少节点之间的重复区域[4],但这种将不在该包围体中的节点全部视为外部节点的方式,会在进行插入操作时导致树节点中含有大量的外部节点,增加了查询过程中的比较次数,而实际上这些所谓的外部节点和其他子节点可能非常邻近。

在算法上,RBD-tree 在以下几方面对 R°-tree 做了改进:1)改进了插入过程中对外部节点的识别算法,在 RBD-tree 中如果将一个孩子节点插入父节点后并不引起父节点密度的降低,则认为该节点并不是一个外部节点,该识别算法不仅从逻辑上更契合外部节点定义而且提高了节点的质量,减少了节点中的外部节点数量;2)优化了删除过程,当在删除过程中节点向下溢出时,通过从父节点借入一个外部节点来防止无意义的重新插入;3)提高了查询效率,由于减少了外部节点数量,因此在查询过程中需要比较的次数也会相应减少。

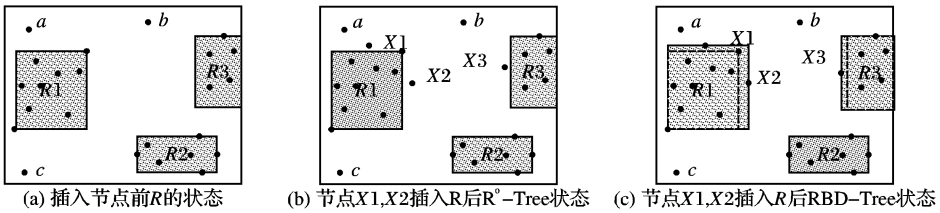
2.2.1 插入操作

RBD-tree 对 R°-tree 的优化就是在进行插入操作时将根据节点密度来区分外部节点。如图 3 所示,在 R°-tree 中 $X1$,$X2$ 是外部节点,但是对 RBD-tree 这两个节点不是外部节点而是节点 $R1$ 的子节点。如果一个待插入的记录和节点中的其他记录在物理上是相邻近的,亦就是新插入的记录并不会降低节点密度,那么认为该记录不是一个外部节点可以插入到节点中。

基于密度的插入方式减少了外部节点的数量,使得后来的插入过程减少了与子节点进行比较的次数,提高了查询效率,而这正是所有索引树的主要目标;根据密度进行划分,相同性质的点集中在一起,外部节点被很好地区分,减少了节点之间的包围体重叠,而这正是查询优化的主要瓶颈。算法的伪代码如下:

```
/*插入操作*/
Set S = ∅;
TreeNode * P = ChooseNode( R );
                    /*寻找一个可以容纳节点 R 的父节点 P */
while( pos = HasEntry( P ))
{
 InsertEntry( pos,R );
 If (! HasBorrowFlag( P ))
   Return;
 TreeNode * O = ChooseAnOutlier( P );
 ClearBorrowFlag( P );
 R = O;
 P = Parent( P );
}
```

```
ChooseOutlierNodes( P,S );
/*从父节点P中选出若干个孩子节点作为外部节点来重新插入*/
If(IsNull( S ))
{
 TreeNode * LL = NULL;
 SplitNode( P, &LL);
 While( LL )
 AdjustTree(Parent( P),LL );
 /*从节点R执行到根节点,调整每个节点的MBR*/
 /*如果在这个过程中发生了节点分裂,则要传播节点分裂带来的更新*/
}
AdjustMBR( P );
ReInsert( S );
```



(a) 插入节点前R的状态　(b) 节点$X1$,$X2$插入R后R°-Tree状态　(c) 节点$X1$,$X2$插入R后RBD-Tree状态

图3　RBD-tree 的外部节点

2.2.2　删除操作

要从索引树中删除一个记录,首先查找记录所在的节点,如果该节点是根节点则直接删除记录,否则将所有节点重新插入到合适的高度。在解决节点溢出问题时,如果节点向上溢出 RBD-tree 的处理方式和 R°-tree 相同;但当节点向下溢出时,RBD-tree 则向父节点借入一个外部节点,并为了表示从父节点借来一个外部节点引入一个标识位,当下次有新的元素插入时,将外部节点再重新移到父节点中,这样可以减少重新插入的节点数量。算法的伪代码如下:

```
/*删除操作 */
S =∅;
TreeNode * R = NULL, * P = NULL, * O = NULL;
If ( ! ( R =FindNode( )))
        /*查找记录C所在的节点R,如果没有找到C返回 */
  Return;
While( ! IsRoot( R ))
{
 If( R - >childrenNum > m )
 {
  DeleteEntry( R,C );
  Return;
 }
 P =Parent( R );
 If( BorrowOutlierFromParent( P,&O ))
 {
  InsertEntry( R,O );
  SetBorrowFlag( R );
  Return;
 }
 If( R - >childrenNum < m )
 {
  DeleteEntry( P,R );
  AddAllChildrenToSet( S,R );
  R = P,C = R;
 }
}
If ( C )
  DeleteEntry( R,C );
Else
  ReInsert( S );
```

在图4中,如果从子节点$R2$中删除节点$Y3$,假设最少孩子数目为7,那么节点$R2$会向下溢出。按照R°-tree 的算法,$R2$中的剩余节点会被重新插入,由于移除$R2$后R的MBR并没有发生变化,所以这些节点很有可能被重新插入到R中,并且这些节点在R中被当作外部节点来处理,造成进一步的节点分裂开销。RBD-tree 的处理过程如图5所示$R2$从父节点R中借入插入代价最小的外部节点d,将节点d插入到$R2$中,此时满足了最小孩子数目的要求,并设置标识位以表示该节点曾经借入一个外部节点,当插入另外的节点O时,插入过程发现该节点被设置了“借入”标识,然后调用外部节点鉴别算法选出一个外部节点归还给父节点R。

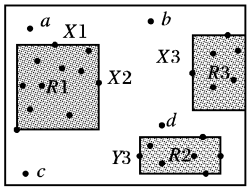
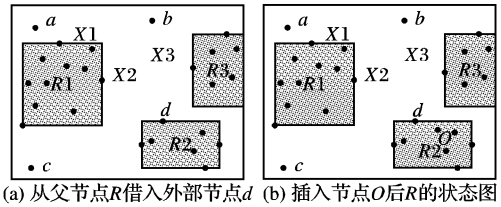


图4　删除$Y3$前节点R的状态



(a) 从父节点R借入外部节点d　(b) 插入节点O后R的状态图

(c) 将$R2$借入的节点归还

图5　RBD-tree 的删除操作

2.2.3　分裂操作

RBD-tree 的分裂算法与 R°-tree 相同,将节点R分裂为两个节点之后,在每个节点里面寻找外部节点并插入到父节点中。

3　实验

为了模拟现实中的地图,地图数据由应用程序随机生成,实验中生成了具有6000条记录的索引树,记录坐标均在经度位于(-180°, 180°)、纬度位于(-90°, 90°)的正常范围,并将不同的区域赋予不同的权值使得在生成随机数据的时候出

现在权值高的区域的节点个数会比较多。实验在 Linux 环境中利用 RAM 模拟 NAND Flash,在 Fedora9 操作系统下将文件系统 yaffs2 作为一个 module 挂载到内核中以充分利用 yaffs2 提供专门的模拟接口,将生成的记录数据用 R°-tree 和 RBD-tree 通过 yaffs2 的接口存储到模拟的 NAND 中。

将记录数据分别用 RBD-tree 和 R°-tree 在内存中进行组织,并对两种索引结构的性能进行比较:

1)RBD-tree 和 R°-tree 的区域查询比较[11];

2)RBD-tree 和 R°-tree 的最近邻居节点查询比较[12-13];

3)RBD-tree 和 R°-tree 的高度比较;

4)RBD-tree 和 R°-tree 的内部节点个数比较。

3.1 树的内部节点个数比较

分别随机生成了 2 000,4 000,6 000 个记录的数据集。针对相同的数据集,比较 R°-tree 和 RBD-tree 的内部节点个数,对比结果如图 6 所示。从图中可以看出,RBD-tree 的内部节点个数比 R°-tree 平均多出 5% ~10% 左右,这是由于 R°-tree 中外部节点总个数比 RBD-tree 要多,使得索引树的内部节点所需个数减少,因此这个结果是在预期当中。

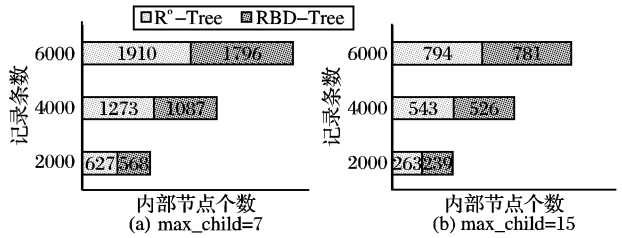


图 6 内部节点个数比较

3.2 树的高度比较

当记录的数量比较多时,两种树具有相同的高度。在多次实验中,6 000 个记录的树高度都为 6,4 000 个记录的高度为 6,2 000 个节点的高度为 5。

3.3 树的区域查询比较

实验中,采用相同的查询方式对相同的数据集进行查询,比较 R°-tree 和 RBD-tree 的查询效率。查询区域的范围是整个区域的 0.01% ~1%,每次实验在区间(0.01%, 1%)上随机生成 1 000 个这种查询区域,计算平均查询时间。然后重复该过程 1 000 次,将每次的平均查询时间相加再取平均数,减少由于样本随机带来的误差,查询结果如图 7 所示。由于查询过程中节点比较次数减少必然使得查询效率提高,数据表明,RBD-tree 的查询性能比 R°-tree 高出 20% 左右。

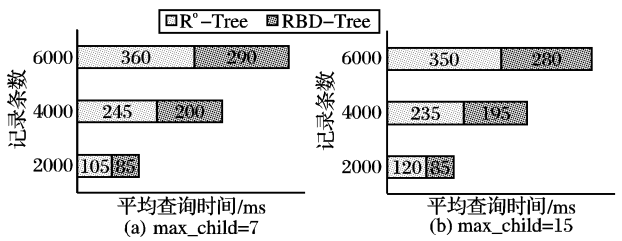


图 7 区域查询平均时间对比

3.4 树的最近邻居节点查询

从实验中还可以看出,在对最近邻居节点进行查询时,由于 R°-tree 的外部节点数量比 RBD-tree 要多,因此当最近邻居节点就在外部节点中时查询在树的顶部就终止的情形多一些;但当最近邻居节点在子节点中的时候,RBD-tree 的优势就要大一些,比较次数会减少,剪枝的过程也要快一些。

4 结语

本文在 R°-tree 的基础上提出了一种新的索引结构 RBD-tree。RBD-tree 比 R°-tree 具有更高的查询效率,但 RBD-tree 比 R°-tree 所占的空间要大一些。总的来看,RBD-tree 采取的是一种以空间换时间的机制,由于一般地图的数据量不是很大,因此 RBD-tree 带来的查询收益要大于空间增大的成本,而且 RBD-tree 是设备无关的,可以更好地、平滑地应用在新的嵌入式地图格式中。

致谢:本文是在上海交通大学蔡松露硕士和戚正伟博士的协助下完成的,在此对他们的大力支持和帮助表示最衷心的感谢。

参考文献:

[1] GUTTMAN A. R-tree: A dynamic index structure for spatial searching[C]// Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data. New York: ACM, 1984: 45-57.

[2] ARGE L, BERG M, HAVERKORT H J, *et al*. The priority R-tree: A practically efficient and worst-case optimal R-tree[C]// Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data. New York: ACM, 2004: 347-358.

[3] BECKMANN N, KRIEGEL H P, SCHNEIDER R, *et al*. The R*-tree: An efficient and robust access method for points and rectangles [C]// Proceedings of the 1990 ACM SIGMOD International Conference. New York: ACM, 1990: 142-153.

[4] ZHANG D, XIA T. Improving the R*-tree with outlier handling techniques[C]// Proceedings of the 13th Annual ACM International Workshop on Geographic Information Systems. New York: ACM, 2004: 204-213.

[5] KAMEL I, FALOUTSOS C. Hilbert R-tree: An improved R-tree using fractals[C]// Proceedings of the 20th International Conference on Very Large Data Bases. San Francisco: Morgan Kaufmann Publishers Inc., 1994: 500-509.

[6] 蔡松露. 基于 NAND Flash 的嵌入式 GIS 地图格式设计及优化[D]. 上海: 上海交通大学, 2009.

[7] VITTER J S. External memory algorithms and data structures[M]. Boston: American Mathematical Society, 1999: 1-38.

[8] ZHANG D, XIA T. A novel improvement to the R*-tree spatial index using gain/loss metrics[C]// Proceedings of the 12th Annual ACM International Workshop on Geographic Information Systems. New York: ACM, 2004: 204-213.

[9] RIJK A D. Improving the R*-tree storage allocation algorithm [D]. Netherlands: Delft University of Technology, 2000: 389-400.

[10] 陈敏, 王晶澎. R*-树空间索引的优化研究[J]. 计算机应用, 2007, 27(10): 2581-2583.

[11] THEODORIDIS Y, STEFANAKIS E, SELLIS T. Efficient cost models for spatial queries using R-tree[J]. IEEE Transactions on Knowledge and Data Engineering, 2000, 12(1): 19-32.

[12] ROUSSOPOULOS N, KELLEY S, VINCENT F. Nearest neighbor queries[C]// Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data. New York: ACM, 1995: 71-79.

[13] GUNTHER O. Efficient structures for geometric data management [M]. Berlin: Springer-Verlag, 1988.