E3D R-Tree：一种处理移动对象数据库历史查询的索引结构

张文杰　李建中　张　炜
（哈尔滨工业大学计算机科学与技术学院　哈尔滨 150001）

摘　要　历史查询是移动对象数据库管理的一个重要方面。为提高历史查询效率，在3D R-Tree基础上实现了优化的索引结构E3D R-Tree。在E3D R-Tree中，结合移动对象数据特征引入空白区域作为新的插入代价参数，同时，在插入算法中利用最小代价优先搜索算法确定全局最优插入路径，并给出算法正确性证明。实验结果表明，E3D R-Tree查询效率高于3D R-Tree。

关键词　移动对象数据库，历史查询，索引结构，插入算法，查询效率

E3D R-Tree: An Index Structure for Indexing the Histories in Moving Object Database

ZHANG Wen-Jie　LI Jian-Zhong　ZHANG Wei
(Department of Computer Science and Technology, Harbin Institute of Technology, Harbin 150001)

Abstract　Efficient index of the histories is an important aspect in moving object database management. This paper implements an optimizing index structure called E3D R-Tree based on 3D R-Tree. E3D R-Tree takes into account the features of moving object data and took advantage of new cost parameters. In particular, least-cost-first search algorithm is used in the insertion algorithm to find the overall best way to insert a new record in E3D R-Tree. The proof of the vidility of the algorithm is given. The result of experiments illustrates that E3D R-Tree outperforms 3D R-Tree in query processes.

Keywords　Moving object database, History query, Intey structure, Insertion algorithm, Query efficiency

1　引言

随着无线通讯技术与定位技术的迅速发展，对移动对象的跟踪和定位变得越来越可行与必要。移动对象数据库管理大量移动对象不断更新的动态信息。典型的移动对象例子是在D维空间中运动的车辆。移动对象通过位置探测装置（如GPS）获得自身的位置和速度信息，并通过通讯网络（如无线网络）向服务器传送，服务器接收这些不断更新的信息并为每一个移动对象保留信息的历史记录。对于移动对象位置信息的查询主要分为两个方面：对于历史信息的查询和对于现在以及未来信息的查询[1]。移动对象数据库历史信息管理在交通监测、城市规划、移动计算等领域有着广泛的应用。在服务器端所支持的典型历史查询是区域查询，包括时刻查询（如“找到在时刻t经过区域S的所有移动对象”），以及窗口查询（如“找到在时间范围内经过区域S的所有移动对象”）。

为了支持对移动对象数据库历史信息的管理，提出了很多索引结构。在这些索引结构中，3D R-Tree能够有效管理移动对象历史信息，并且在处理长时间窗口查询时效率高于其它结构。然而3D R-Tree中的插入代价参数未能体现出移动对象数据的特征，并且在插入新的记录时，3D R-Tree在每一步上利用贪心策略选取当前局部最优节点插入，无法得到全局最优路径，因而影响查询效率。

为进一步提高处理移动对象数据库历史查询的效率，本文提出了以3D R-Tree为基础进行优化的索引结构E3D R-Tree。优化主要体现在以下两个方面：第一，在3D R-Tree基础上，以移动对象数据特征作为指导，引入空白区域作为新的插入代价参数；第二，E3D R-Tree在插入新的移动对象记录时，利用最小代价优先算法确定全局最优插入路径。为确定全局最优插入路径，E3D R-Tree在插入过程中与3D R-Tree相比将访问更多的节点。但对于历史信息的索引来说，更新的频率远小于查询的频率，因此降低更新效率，以得到更优的索引结构，从而提高查询效率，是合理的。在实验部分将对两种索引结构在查询以及更新时所需的I/O代价进行比较。

2　相关工作

支持移动对象数据库历史信息查询的索引结构可以归纳为以下三类[2]：第一类基于现有的空间索引结构，在空间索引方法上加入时间要素来实现，如3D R-Tree和STR-Tree等。3D R-Tree管理移动对象信息的主要思想是将时间作为单独的一维来处理，不区分时间维和空间维[3]；STR-Tree是对R-Tree的扩展，具有与R-Tree不同的插入/分裂算法，主要思想是保持同一对象轨迹的各个部分的空间聚集性[4]。第二类是基于重叠多版本结构的索引，主要思想是将时间和空间分别处理，如HR-Tree、HR$^+$-Tree以及MV3R-Tree等。HR-Tree的主要思想是为每一个更新时刻均构建一棵独立的R-Tree，这样时刻查询退化为利用R-Tree处理的空间查询[5]，因而可以有效支持时刻查询，但支持窗口查询效率很低。HR+-Tree是在HR-Tree基础上的改进[6]。这两种方法均需要大量的存储空间，且存在数据冗余问题。在MV3R-Tree中构建两棵树，一棵MVR-Tree（Multi-Version R-Tree）用来处

张文杰　硕士研究生，主要研究方向是移动对象数据库；李建中　博士生导师，主要研究方向是数据库及相关技术；张　炜　博士研究生，主要研究方向是移动对象数据库。

理时刻查询，一棵 3D R-Tree 用来处理窗口查询[7]。第三类是支持面向轨迹(trajectory)信息查询的索引，主要思想是优先考虑同一对象的轨迹属性，移动对象的空间聚集性则被放在次要的位置。典型的轨迹查询如“找到在时间段范围内经过区域 S 的移动对象在下一个小时内的位置信息”，索引结构包括 TB-Tree[4]、SEB-Tree[8]、SETI[9] 等。在这些处理历史查询的索引结构中，3D R-Tree 结构简单无冗余，在处理长时间窗口查询时效率高于其它结构，而且 3D R-Tree 本身是一些索引结构(如 MV3R-Tree)用于处理窗口查询的一部分。因此，对 3D R-Tree 索引结构的优化以提高历史查询效率有着现实的意义。

本文中所指 3D R-Tree(3-Dimensional R-Tree)实现基于 R*-Tree 的构造算法，R*-Tree 的结构是在 R-Tree 基础上的进一步优化[10]。在利用 3D R-Tree 对移动对象信息进行管理时，时间维被当作空间维之外单独的一维来处理，在对时间维和空间维的处理上不存在差别，如图 1 所示。R-Tree 是管理高维数据的最有效的索引结构之一，树形结构中包括中间节点以及叶节点。3D R-Tree 中移动对象信息保存在叶节点中，中间节点通过聚集低层节点而建立。移动对象信息包括时间和空间两个方面，常见信息格式为(id，t，bounces，velocity)，其中 id 为对象标号，t 为移动对象向服务器发送信息的时刻，bounces、velocity 为对象在 t 时刻的空间位置和速度矢量。叶节点的最小范围矩形(Minimum Bounding Rectangles：MBR)最小覆盖了其中包含的所有移动对象，中间节点的 MBR 则最小覆盖了其所有子节点的 MBR 的范围，如图 1 所示。

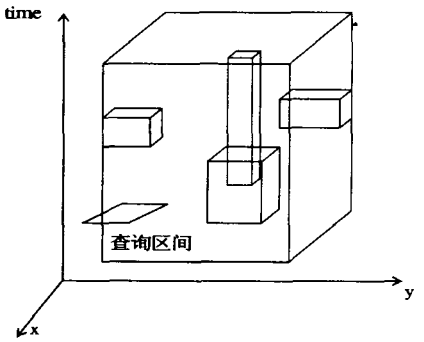

图 1　3D R-Tree 中的时刻查询

基于 R*-Tree 的构造算法，3D R-Tree 在构造时最小化如下代价参数：(1)MBR 的面积；(2)MBR 的周长；(3)同一节点内的 MBR 之间的覆盖；(4)一个 MBR 的中心与包含它的节点中心之间的距离。具体来说，每当插入一个新的记录时，从根节点开始，在每一层上插入算法根据贪心策略选择包含新的记录后面积扩大最少的子节点。特别地，在选择叶节点插入新记录时，选择面积扩大后引起兄弟节点之间覆盖最小的叶节点来包含新记录。如果要插入新记录的叶节点已满，首先会从叶节点中删除中心点距离叶节点 MBR 中心点最远的一部分记录，然后将这些记录重新插入到 3D R-Tree 中去。在这些记录都可以插入到其它叶节点的前提下，尽可能避免节点分裂。如果在重新插入后仍有溢出，则执行节点分裂算法。节点分裂分为两个步骤，首先选择分裂坐标轴，选择的依据是按此坐标轴分裂得到的两个节点具有最小的周长之和；然后将节点中的记录依据选定的坐标轴进行排序，考虑每一种可能的分界。最后选择的分界可以保证得到的两个新节点的 MBR 之间的覆盖最小。

3　E3D R-Tree 中的代价参数

在 3D R-Tree 中，大量空白部分的存在会影响查询效率。如图 1 所示，查询区间可能会与节点的 MBR 相交，但相交的部分并没有实际对象，访问无用节点。而对于 3D R-Tree 中所处理的移动对象数据来说，其中的时间维随着移动对象信息中更新时刻 t 的增加而不断向上扩展，MBR 中的空白部分会占 MBR 的很大比例。由此我们考虑使用空白区域作为一个插入代价参数，在 E3DR-Tree 的构造算法中控制节点中空白区域的增加，使节点更加紧凑，减少查询区域与节点 MBR 的空白部分相交的可能性，提高查询效率。

定义 1(空白区域)　在 E3D R-Tree 中，一个节点的空白区域定义为节点中没有被实际对象覆盖区域的面积，表示为 Dead_space(Node O)=Coverage(O)−Area(O)。其中，Area 为节点的面积，而 Coverage 为节点中被实际覆盖区域的面积。

同时，当有一个新的记录要插入时，3D R-Tree 在非叶节点层次上选择插入后面积增长最小的节点作为下一步插入的节点，当面积增长相同时，选择节点面积较小的一个。这种方法在多个节点具有相同或相似的面积增长时效率会降低。如图 2 所示(在这里，我们以二维数据为例进行说明，三维形式同二维相似)，h,i,j 为根节点的子节点，$a,b,\cdots,g$ 为 h,i,j 相应的子节点且为叶节点的上层节点，以 level 表示节点所在的层数。在 E3D R-tree 中，定义叶节点 level=0。假设要插入记录 p，位置如图。按 3D R-Tree 中算法，在从根节点的子节点开始选择时，p 插入 h，j 节点都不会引起该节点面积的扩大。在这种情况下，选择面积较小的节点，即 h 节点。然后，h 节点选择插入引起面积扩大最小的子节点为下一步插入的节点，即节点 e。然而，插入记录 p 的最佳选择是节点 a，因为它引起的面积增大远小于节点 e。但是，3D R-Tree 中所使用的插入算法每一步只能得到下一步的最佳插入选择，无法得到全局最优结果。

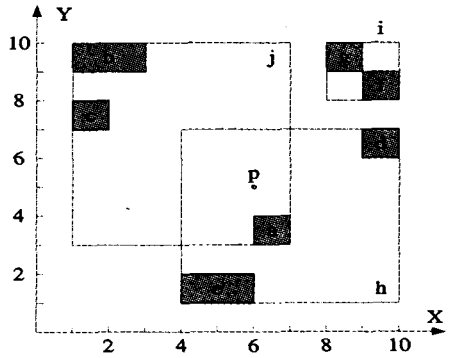

图 2　R*-Tree 中的插入

由此，在 E3D R-Tree 中插入一个新的记录时，考虑使用最小代价优先搜索算法，代价参数设为从根节点至 level=1 节点的插入路径上经过的所有节点插入记录后扩大的面积之和，这样得到从根节点到叶节点的代价参数。

定义 2(插入代价参数)　设叶节点层数 level=0，root_level 表示根节点层数。E3D R-Tree 在插入过程中的代价参数定义如下：

$$\text{Cost}=\sum\Delta\text{Area(O)}\quad 1\leqslant\text{node level}\leqslant\text{root\_level}$$

$$\text{Cost}=\Delta\ \text{dead\_space}\quad \text{node level}=0$$

根据插入代价参数，插入新记录时选择一条从根节点到level=1节点的路径作为全局最优插入路径，这条路径满足沿该路径插入记录后路径中的所有节点面积增长之和最小。然后根据这条路径，选择level=1节点的子节点中插入记录后dead_space增大最小的叶节点，完成插入过程。

4 E3D R-Tree的插入算法

4.1 插入算法

E3D R-Tree中的插入算法总体描述如下：

```
Algorithm Insert(e)
/* Input: e is the entry to be inserted */
1. re-inserted_i=false for all levels of the tree
2. initialize an empty re-insertion list L_reinsert
     //用于保存在叶节点溢出时删除的记录
3. invoke Choose Path to find the path to insert e
     //路径选择算法，以选择最优插入路径
4. invoke Node Insert(Root, e)
   //节点插入算法，从根节点开始插入
5. for each entry in the L_reinsert
   //将L_reinsert中的记录重新插入到树中
6.   similar to lines 3-5
End Insert
```

其中，在Node Insert算法中，从根节点开始在每一步选取子节点进行插入时，根据Choose Path算法得出的路径进行，并选择路径中level=1节点的子节点插入记录后Cost=Δdead_space最小的叶节点。以下分别介绍Choose Path算法和节点分裂算法Node Split。

4.2 路径选择算法 Choose Path

在插入时使用基于最小代价优先搜索的Choose Path算法，给定一个新的记录，算法得到从根节点到level=1的节点的所有路径中使得代价参数中Cost=$\sum_{everynode}$·ΔArea(O)取值最小的一条。算法的基本思想是在当前所有可以展开的节点中选择具有最小参数值的节点进行扩展，具有全局化的观念。在Choose Path算法中，利用优先队列PQ来记录候选路径以及相应的代价，PQ中的各项按照代价参数取值由小到大的顺序排列。以图2为例，设r表示根节点，利用Choose Path算法寻找最优插入路径的步骤可以描述为：

1. 初始时，优先队列PQ为{⟨(r),0⟩}，展开r的子节点后将得到的路径插入到PQ中，并删除PQ中的第一个记录⟨(r), 0⟩，得到{⟨(h, r),0⟩,⟨(j, r), 0⟩,⟨(i, r), 16⟩}，其中的每个值代表了p插入该路径所得到的代价参数的取值。p插入节点h，j并不会引起节点面积的增加，因此代价参数值为0，而插入节点i后其面积增大16。
2. 访问PQ中的第一项中节点h，向PQ中插入另外两条路径(e, h, r)和(d, h, r)后将⟨(h, r), 0⟩从PQ中删除，PQ={⟨(j, r), 0⟩, ⟨(e, h, r), 6⟩, ⟨(d, h, r),7⟩, ⟨(i, r), 16⟩}。路径(e, h, r), (d, h, r)已到达level=1的节点，不再继续。
3. 展开第一项中的路径(j, r)，将路径(a, j, r),(b, j, r)和(c, j, r)插入到优先队列中，PQ={⟨(a, j, r), 1⟩, ⟨(e, h, r),6⟩, ⟨(d, h, r), 7⟩, ⟨(c, j, r), 14⟩, ⟨(i, r), 16⟩, ⟨(b, j, r), 23⟩}。这时，找到了一条路径(a, g, r)至level=1的节点且代价最小，(a, j, r)即是最终选择的路径。
4. 在选择叶节点时，选择节点a的子节点中插入p后dead_space增加最小的一个作为最后一步插入的节点，从而确定从根节点到叶节点的全部插入路径。

实现时，利用一个堆的实例path_heap来实现优先队列，path_heap中的每一项heap entry中包含属性：son，path，cost，其中son表示下一步要访问的节点的id，path中记录当前访问过的路径，cost=$\sum_{everynode(O)}$ ΔArea(O)，即路径中各个节点包含新记录后面积增加之和，O是路径中的节点。这样，从根节点到level=1节点的利用最小代价优先算法的Choose Path算法描述如下：

```
Algorithm Choose Path (e)
/* Input: e is the new entry to be inserted; */
1 initialize a heap path_heap, insert root node into it
    //初始化堆，插入根节点信息作为堆的第一项
2 while (path_heap→cont [0]. cost<mincost)
        //检查是否已经找到目标路径
3     get first entry hpe in path_heap, get the son node N of hpe,
      delete hpe //取第一项子节点后将其删除
4     if N→level=2
5         for each entry e'in N
6             compute the area A of e'
7             compute the area A' of e after contains e
8             cost=A'-A+hpe→cost //计算代价参数
9             if(cost <mincost)
10                Copy from hpe→path to path
              //将代价参数小的项中路径拷贝到path中
11                path [N→level-1]=e'. son//level=子节点
12                mincost=cost //更新mincost
13    else if N→level>2
14      for every entry e' in N
15          Get parameter cost similar to line 5~7
16          if(cost <mincost)
17              initialize a new heap entry new_hpe
                    //创建新的向插入到堆中
18            new_hpe→path [N→evel-1]=e'. son
19            Copy from hpe→path to new_hpe→path
20            Insert new_hpe into path_heap
                  //向path_heap中插入新的候选路径
21 Return Path //将path作为最终结果返回
End Choose_Path
```

定义3（状态） 在E3D R-Tree中，状态定义为在优先队列PQ的项中所保存的从根节点到中间节点的路径。形式表示为⟨N_1, N_2,…,N_k⟩，其中N_1为根节点，N_{i+1}是N_i($1\leqslant i\leqslant k-1$)的子节点，$1\leqslant N_k$. level$\leqslant$root_level。

定义4（后代、直接后代） 一个状态的后代定义为由这个状态生成的后续路径，形式定义为：设⟨N_1, N_2,…,N_k⟩为一个状态，则其后代为⟨N_1, N_2,…,N_k, N_{k+1},…,N_h⟩，N_{i+1}是N_i的子节点($k+1\leqslant i\leqslant h-1$)，且$1<N_k$. level$\leqslant$root_level。如果h=k+1，则称这个后代为直接后代。

定理 Choose Path算法可以在有限步骤内终止并找到一条满足代价参数Cost=$\sum_{everynode\,O}$ ΔArea(O)最小的最优路径。

证明：首先，必存在从初始节点（根节点）到目标节点（level=1的节点）的最优路径。这是因为E3DR-Tree高度有限，从初始节点到目标节点的路径至多为有限条，其中必然存在一条或几条路径使代价参数取值最小。其次，在算法结束前，每条最优路径均有一个状态在优先队列PQ中。事实上，每条从初始节点到目标节点的路径都会有一个状态在PQ中。这是因为从初始节点开始，每个由当前状态生成的后代都被插入PQ中，只有在当前状态生成所有后代后，才将其从PQ中删除。现假设PQ中状态N=⟨N_1, N_2,…,N_k⟩是最优路径上的一个状态，则对于任意选定的时刻，PQ中位于N之前的状态只能为有限个，称这有限个状态为第一代状态，其中代价参数取值最小者为a_1。第一代状态的直接后代称为第二代状态，其中代价参数取值最小者为a_2，设一个状态每生成一个后代代价参数增加值为e，则$a_2\geqslant a_1+e$。一般地，有$a_j\geqslant a_1+(j-1)e$。设最优路径代价参数为C，当j足够大时，有$a_j\geqslant C$。这说明，由第一代状态生成的后代只能有有限代位于N之前。由于每个状态只能有有限多个直接后代，在此之后它就要被从PQ中删除，因此经过有限步骤后S必然成为PQ

中的第一个状态(如果在此之前算法尚未结束的话)。又由于最优路径上只有有限多个状态,因此经过有限步后算法必然因到达目标状态 T 而停止,这就是最优解。若算法在尚未执行到 T 之前就停止了,也即找到了另一个目标状态或另一条路径,则由于 PQ 中的状态是按代价参数取值大小排列的,它必然也是最优解。

Choose Path 在节点 level>0 时利用最小代价优先算法找到最佳插入路径,与原 3D R-Tree 算法相比,需要更多的节点访问,然而 Choose Path 算法可以产生更好的树结构,最终提高查询效率。而且大部分路径在上层节点就已终止,如图 2 中的路径(i , r),无需进一步的节点访问。在实验部分会对查询与插入时访问的节点与 3DR-Tree 进行对比。

4.3 节点分裂算法 Node Split

与 3D R-Tree 相似,E3D R-Tree 中节点分裂算法首先把节点中所有记录沿着各个坐标轴排序,考虑每一种可能的分裂方式后,选择分裂后节点周长之和最小的坐标轴为分裂坐标轴。接下来在沿着选定的坐标轴分裂节点时,对每一种分裂方式计算分裂后的两个节点空白区域 dead_space 之和,取 dead_space 之和最小的为最终分裂方式。

5 实验结果分析

由于移动对象的实际运动数据难以获得,本文使用的数据基于实际地图数据通过模拟方法产生。实验中移动对象的总数为 10 万个,每个移动对象每隔 1min 更新其位置和速度信息,共更新 50 次,因此共插入记录 500 万条。移动对象运动空间范围为 1000km × 1000km,最大速度在 90km/h 到 120km/h 之间均匀分布。在实验数据中,移动对象的初始位置均匀分布在空间范围内。它们初始速度的大小和方向在其可能的速度范围内随机设定,每次更新位置信息时随机更新速度矢量。在所有的实验中,3D R-Tree 与 E3D R-Tree 的页面大小均被设定为 8kB。

5.1 窗口查询

图 3 中比较了 3D R-Tree 与 E3D R-Tree 在处理窗口查询时的 I/O 代价。在窗口查询中时间范围保持不变而改变空间范围大小。其中所有查询的时间范围均为 10min,空间范围为正方形,图中横坐标值为正方形边长。由图中可以看出,随着空间范围的扩大,两种索引结构需访问的节点数都在增多,但与 3D R-Tree 相比,E3D R-Tree 需要更少的 I/O 次数,查询效率更高。

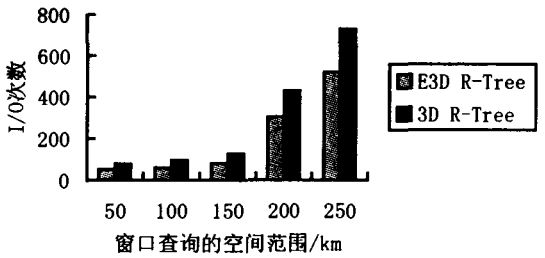

图 3 查询 I/O 与窗口查询空间范围

在图 4 中,3D R-Tree 与 E3D R-Tree 所处理的窗口查询空间范围保持不变而改变时间范围。其中查询的空间范围被设置为边长为 100km 的正方形,而查询的时间范围大小为横坐标值。如图所示,随时间范围的不断增大,查询所需的 I/O 次数增多,E3D R-Tree 与 3D R-Tree 相比,访问更少的节点。与图 3 相比较可以看出,I/O 次数时间范围增大的程度小于随空间范围增大的程度,这是因为大多数的移动对象在一段时间内空间范围的改变并不大。

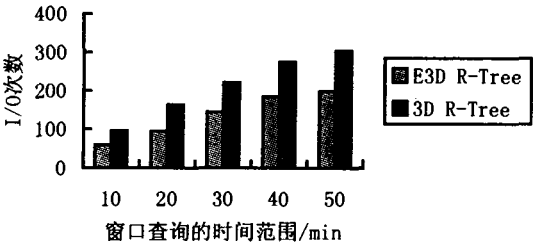

图 4 查询 I/O 代价与窗口查询时间范围

5.2 时刻查询

在一些应用中,需要移动对象在过去某一时刻的位置信息[3],时刻查询是窗口查询在时间范围为 0 情况下的特例。图 5 描述了查询 I/O 随查询时刻的变化,在实验中查询的空间范围设定为边长为 100km 的正方形,横坐标代表了查询的时刻。可以看出,在时刻查询中,E3D R-Tree 的效率同样高于 3D R-Tree。

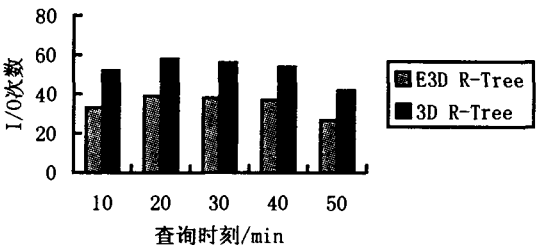

图 5 查询 I/O 代价与查询时刻

5.3 更新代价比较

图 6 比较了 3D R-Tree 与 E3D R-Tree 在插入记录时所需的 I/O 代价。对 10 万条数据,分别记录更新 10,20,30,40,50 次之后所需总的 I/O 次数,然后除以插入记录条数,得到插入一条记录所需的平均 I/O 次数,如纵坐标所示。可以看出,随着更新次数的增加,两种索引结构在插入过程中要访问的节点数目都在增加,这是因为随插入记录条数的增加,3D R-Tree 与 E3D R-Tree 的高度都在增长。从结果可以看出,由于大部分路径在上层终止,无需进一步展开,E3D R-Tree 插入时所访问的节点个数与 3D R-Tree 相比增大并不显著。但最小代价优先搜索算法可以产生更优的树结构,从而提高查询的效率,这对于移动对象历史信息的处理是更有价值的。

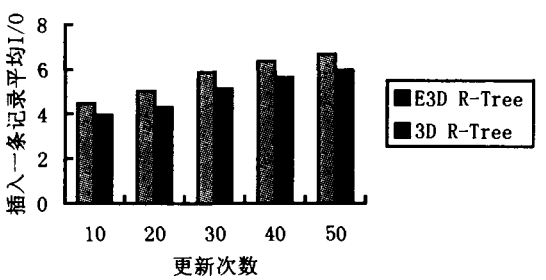

图 6 插入一条记录平均 I/O 次数与更新次数

结论以及进一步工作 对于移动对象历史信息的查询是

移动对象数据库管理的重要方面，3D R-Tree 是可以提供对于高维数据有效管理的数据结构。本文基于 3D R-Tree 结合移动对象数据特点提出了新的插入代价参数，并利用最小代价优先搜索算法确定从根节点到 level=1 的节点的全局最优插入路径。在选择叶节点插入时，保证叶节点的空白区域增加最小，以减小在查询时访问无效节点的可能性。这样，虽然在插入记录时需要额外的节点访问，但与 3DR-Tree 相比提高了查询效率。移动对象数据库领域目前的大部分研究工作集中于区域查询处理，未来工作是研究移动对象数据库环境下的连接以及最邻近算法。

参 考 文 献

1 Satenis S, Jensen C S, Leutenegger S T. Indexing the positions of continuously moving objects. In: Proc. of the 202 ACM SIGMOD Int'l Conf. on Management of Data. New York: ACM Press, 2000. 331～342

2 Mokbel M F, Ghanem T M, Aref W G. Spatio-Temporal Access Methodes. IEEE, 2003

3 Vazirgiannis T M, Sellis T. Spatio-Temporal Indexing for Large Multimedia Applications. In: Proc. of the IEEE Conf. on Multimedia Computing and Systems, ICMCS, 1996

4 Pfoser D, Jensen C S, Theodoridis Y. Novel Approaches in Query Processing for Moving Object Trajectories. In: Proc. of the Intl Conf. on Very Large Data Bases(VLDB), 2000. 395～406

5 Nascimento M A, Silva J R O. Towards historical R-trees. In: Proc. of the ACM Symp on Applied Computing(SAC), 1998. 235～240

6 Tao Y, Papadias D. Efficient Historical R-trees. In: Proc of. the Intl Conf. on Scientific and Statistical Database Management(SSDBM), 2001. 223～232

7 Tao Y, Papadias D. MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. In: Proc. of the Intl. Conf. on Very Large Data Bases, (VLDB), 2001. 431～440

8 Song Z, Roussopoulos N. SEB-tree: An Approach to Index Continuously Moving Objects. In Mobile Data Management(MDM), 2003. 340～344

9 Chakka V P, Everspaugh A, Patel J M. Indexing Large Trajectory Data Sets with SETI. In: Proc. of the Conf. on Innovative Data Systems Research(CIDR), 2003

10 Beckmann N, Kriegel H, Schneider R, et al. The R*-Tree: An efficient and robust access method for points and rectangles. In: Proc. of the 1990 ACM -SIGMODE Conf.

(上接第 99 页)

验结果可以看出，利用立方体计算中的分割策略大大提高了算法的效率，维度越大，效果越显著。

此外，由于我们在算法中使用了插入排序方法来记录当前幅度最大的前 k 个梯度，对算法效率的提高也起到了相当大的作用。插入排序在最好情况下的时间复杂度是 $O(n)$，最坏情况下的时间复杂度是 $O(n^2)$，其中 n 表示数据立方体格中的总边数。采用插入排序的算法与直接比较的方法相比，优势显而易见。此处限于篇幅，不再详细列出实验结果。

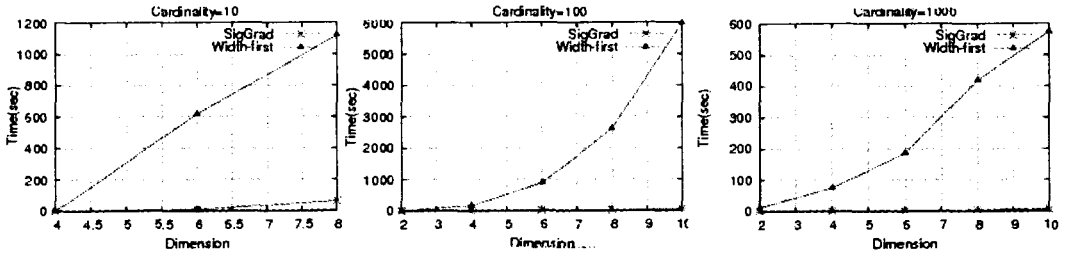

图 7 分割策略对算法效率的影响

小结 本文提出了一种与梯度阈值无关的关键梯度分析方法。我们通过在立方体计算过程中添加补充路径，实现了深度优先方式的关键梯度挖掘方法。由于利用了计数排序、分割策略以及插入排序，使得算法的效率得到很大提高。同时，由于返回的结果是最关键的梯度特征，使得用户不仅在最大程度上不会遗漏重要有价值的信息，也使得分析的结果更加简洁易懂。实验结果证明了算法的高效性和适用性。下一步的工作是针对其他两种梯度特征进行研究，提出合适的分析策略和相应的高效算法。

参 考 文 献

1 Imielinski T, Khachiyan L, Abdulghani A. Cubegrades: Generalizing association rules. In: Proc. of the 8th Int Conf. on Data Mining and Knowledge Discovery. ACM, 2002. 219～257

2 Dong G, Han J, Lam J, et al. Mining multi-dimensional constrained gradients in data cubes. In: Proc. 2001 Int Conf on Very Large Data Bases (VLDB'01), Roma, Italy, 2001

3 Zhao Y, Deshpande P M, Naughton J F. An array-based algorithm for simultaneous multidimensional aggregates. In: Proc. 1997 ACM-SIGMOD Int Conf. Management of Data (SIGMOD'97), 1997. 159～170

4 Han J, Pei J, Dong G, et al. Efficient computation of iceberg cubes with complex measures. In: Proc. 2001 ACM-SIGMOD Int Conf. Management of Data (SIGMOD'01), 2001. 1～12

5 Wang K, Jiang Y, Yu J X, et al. Pushing aggregate constraints by divide-and approximate. In: Proc. 2003 Int Conf. Data Engineering (ICDE'03), 2003. 291～302

6 Xin D, Han J, Li X, et al. Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. In: Proc. 2003 Int Conf. on Very Large Data Bases (VLDB'03), 2003. 476～487

7 Ng R T, Wagner A S, Yin Y. Iceberg-cube computation with PC clusters. In: Proc. 2001 ACM-SIGMOD Int Conf. Management of Data (SIGMOG'01), 2001

8 Ross K A, Zaman K A. Optimizing selections over datacubes. In: Statistical and Scientific Database Management, 2000. 139～152

9 Beyer K, Ramakrishnan R. Bottom-up computation of sparse and iceberg cubes. In: Proc. 1999 ACM-SIGMOD Int Conf. Management of Data (SIGMOD'99), 1999. 359～370