國 立 交 通 大 學

電機資訊學院 資訊學程

碩士論文

空間資料庫有效索引改善方式及應用

An Efficient Enhanced Method for Indexing with Implementation in Spatial Database

研 究 生：李韋毅

指導教授：李素瑛　教授

中 華 民 國 九 十 五 年 八 月

空間資料庫有效索引改善方式及應用

An Efficient Enhanced Method for Indexing with Implementation in Spatial Database

研 究 生：李韋毅　　　　Student： WEI-YI LEE

指導教授：李素瑛　　　　Advisor：Dr. SUH-YIN LEE

國 立 交 通 大 學

電機資訊學院 資訊學程

碩 士 論 文

A Thesis
Submitted to Degree Program of Electrical Engineering and Computer Science
College of Electrical Engineering and Computer Science
National Chiao Tung University
in Partial Fulfillment of the Requirements
for the Degree of
Master of Science
In
Computer Science
August 2006
Hsinchu, Taiwan, Republic of China

中 華 民 國 九 十 五 年 八 月

空間資料庫有效索引改善方式及應用

學生：李韋毅　　　　　　　　指導教授:李素瑛博士

國立交通大學電機資訊學院 資訊學程（研究所）碩士班

An Efficient Enhanced Method for Indexing with

Implementation in Spatial Database

Student: WEI-YI LEE　　　　Advisor: Dr. SUH-YIN LEE

Degree Program of Electrical Engineering Computer Science
National Chiao Tung University

中文摘要

在本論文中我們探討在空間資料庫搜尋大量物件資料所面臨耗時與效率的問題. 因此運用新式的儲存與索引架構透過有效提高索引的方法以及經驗與實驗為根據藉此能夠提升搜尋時的效能. 論文中演算法提供了一種利用 R-Tree 與雜湊法(Hashing)相結合的精簡方式來導引空間資料的搜尋並且探討如何利用這方法來強化存取大型空間資料庫. R-Tree 運用方形邊界範圍來決定是否要進行搜尋其中的節點. 透過這方式絕大部分樹的節點在搜尋時會過濾,也正因為如此 R-Tree 適合在資料庫中運用分開索引與資料方式來處理運作. 同時我們也審視和分析現今常用樹狀結構的演算法並且也確實察覺論文中提出的新方式能夠在大型資料庫中利用現有的架構來縮短搜尋時間. 我們採用了大量載入(bulk-loading)與雜湊(Hashing)資料的方式並且在實驗中證明新的觀點能夠在空間資料庫做搜尋時更省時更有效率.

Abstract

In this thesis we assess the efficiency issue when retrieving sets of objects from a very large spatial database. Thus enhanced performance will be empirically shown here through the new storing and indexing structure. The algorithm provides a condensed method to guide a spatial search and to enhance large data access operations by integrating hashing and R-Tree together. R-tree uses the bounding boxes to decide whether or not to search inside of a child node. In this way most of the nodes in the tree are proved during a search which makes R-trees become more suitable for database operations. We analyze current tree-based algorithms and verify that the new approach in the thesis improves the efficiency in the current architecture. To accomplish this, we use the bulk loading data with hashing into database together with experiments showing that the new algorithm supports spatial queries on spatial database efficiently.

誌　　謝

能夠順利完成論文，首先要謝謝指導教授 李素瑛老師，在研習過程中給予我各方面的教導和容忍學生繁重的工作所帶來諸多的不方便。謝謝擔任口試委員的 許芳榮 教授與 陳正 教授教授於口試期間給予詳盡的指導，讓這份論文亦趨嚴謹更完整的呈現。還要感謝我的女朋友佩慧犧牲了很多時間靜靜的在我身旁鼓勵我，忍受我新竹台北兩地奔波，就算舟車勞頓也願意陪在我身邊一起堅持下去，以及我的家人背後默默的打氣和支持。 還要感謝那群一起奮鬥的專班同學在患難中有著革命真情一起為著希望共同打拼。當初考上在職專班由於剛好工作轉換的關係並沒有直接銜續課業，反而因為新職務的需要允諾主管延後一年學業。曾經在多方壓力和時間配合不易下想要放棄，但是一直到今天一切的努力和汗水都是值得的 – 而且我以成為交大的一份子為榮。要感謝的人太多，無法一一答謝，對於所有一路走來幫助我的家人，師長和朋友致上十二萬分的感謝。

中 華 民 國　九 十 五 年 八 月

Table of Contents

List of Figures

Chapter 1 Introduction

With the rapid progress of modern information technology, recent studies of indexing in spatial database present new challenges due to the complexity, large size of data type which may determine the access methods of query processing on the different architecture. Most commercial databases consider the searching and indexing the requisite data structure of design. And they are providing a way of storing and retrieving the collections of objects which allows efficient access, insertion and deletion by key value. Various search methods have been defined for indexing in spatial database in past years.

Of course one methodology does not fit all of spatial data. Therefore, the growing need for computational power and storage capacity caused by the construction and operation of data makes the use different structure as appropriate choice. However, classic architectures of spatial database fail to improve performance significantly when it comes to scenarios with numerous simultaneously insertion with unbalanced structure because it will generate a skew tree and re-construct the whole tree several time. So in my approach, we use the R-Tree [1] to illustrate original query result and present the performance difference after applying the new methodology here. Commercial database widely deployed with R-Tree [1] for diagnosing the operation of indexing in spatial database. These databases also defined different level of objects to reflect the behavior on storing the objects for minimum boundary rectangle that enclosed the face for the geometries and boundaries. Most databases pre-defined spatial objects to retrieve more efficient during queries. In our approach we enhance the existed methodology and create the hashing with bulk-loading to prove and refer the differentiation in the algorithm with original design.

Furthermore, we start from investigating several index structures for efficient index update and query evaluation. Also we demonstrate the proper issues using each algorithm to see the improvement that are referred to the following methods of the indexing.

In the chapter two, we see the possible problems if data is inserted improper, construct without optimization, or splits with wrong direction, put nodes in wrong rectangles... that may lead the tree to skew or unbalanced. The possible problems will show up and many studies tried to fix it. It creates the motivation on the approach in this chapter.

The chapter three brings a detailed description of the R-tree family. Each of the most important members (R-Tree, R*, R+) [1] [2] [3] will be investigated, starting from data structures, and with the algorithms for new/insertion, retrieval and deletion and finishing with possible efficiency problems. And also we will compare the advantage and disadvantage of these methods that may have in special cases by constructing the proper indexing in R-Tree.

Following in the chapter four, we use bulk loading data with HASHING to illustrate how we solve the issues with this indexing structure to separate from “zones” and propose a filter by building up architecture recursively with R-Tree. It can easily accommodate existing R-Tree based algorithms with minor modification. The hashing R-tree is initiated by indexing minimum bonding rectangles (MBR) when summarizing data in the same cluster at coarse level. It is light-weight, easy to build and maintain, with full capability of an R-Tree. Also with batch loading before hashing function of the R-Tree, we can facilitate the processing of complex spatial data in a timely manner by executing before constructing the R-tree.

In the Chapter five, we prepare the environment of an Oracle database and elaborate the details of our enhancement during large data operations with our approach. We can show the simulations that considerable improvement is possible in terms of decreased response time, if we optimize the packing individually as proposed. The experiments and matrixes of improvement are described in the bottom.

Finally, chapter six gives a brief summary and conclusion.

Chapter 2 Problem Definition

2.1 Motivation

According to a published IDC study at Nov. 2005[22], the spatial information management (SIM) industry has experienced sweeping changes over the last 18 months, involving fundamental shifts in platforms, vendors, and users. The study finds that spatial information management has transformed from a specialist application to a technology with broad relevance within many IT ecosystems. "IDC Reveals Radical changes in spatial information management that will impact most IT companies."

The reason I choice Spatial as the subject on my approach is because most of commercial spatial solutions implement R-Tree as its indexing organization. R-tree [1] demonstrates easy to use, easy to construct, and quick to manage spatial objects nowadays. R-trees [1] are very useful in storing very large amounts of data because of high-balanced structure. Our problem consists of computing results of time consumption when try to insert and retrieve from R-tree [1].

2.2 Possible problems

I would like to raise the potential problems that we may face during R-Tree operations here:

1) Overlap: R-tree [1] uses Minimum Bounding Rectangles (MBR) to associate with different nodes which may overlap. If the geometry of the object is overlapped then we have to search all sub-trees rooted within the node. Thus only the latter will return a qualifying rectangle which takes a lot of time.
2) Algorithm will determine the performance: most of the R-tree operation is revised by each one who implemented in R-tree architecture. So the result of performance on operations will be diverse.
3) Wrong split or wrong insertion/deletion will cause the tree to skew or unbalanced.
4) Searching efficiency with object-oriented spatial data. Computer world is 0 and 1 digitalize for every single components. So when we perform the spatial query, it will rely on the spatial relationships of entities geometrically defined and located in space without regard to the nature of the coordinate system. Thus it will impact the performance with a weak design of query algorithm.

Chapter 3 Overview of previous indexing approaches

First at all we introduce the concept and components of spatial database and then move to the R-tree family to illustrate the popular studies in past years. Let’s move to the terms of spatial objects:

Geometry: ordered sequence of vertices that are connected by straight line segments or circular arcs etc. consisted of points, line strings (including compound), *n*-point polygons, arc line strings, arc polygons (including compound) and etc. Besides, the data objects are approximated by simple objects, i.e. rectangles so that we can easily distinguish out the “real” geometry and description using rectangles.

Furthermore, since we have illustrated that index structure helps to build up on the same objects, so we can query with this mechanism and get quick response. Data model also plays an important role in spatial database. It is divided into elements (sets of physical data), geometry assembled with single or multiple elements, layer (collections of geometries), and the coordinate system to fully integrate to demonstrate features of the spatial database.

In spatial database, we can use three methods to query the object from it: direct query, within distance, nearest neighbor. It will use filter to sort out smaller candidate set and then base on the sets to query with relationships, and identify the final target with database operators (or relationships):

Disjoint, Touch, Overlapbdydisjoint, Overlapbdyintersect, Equal, Contains, Covers, Inside, Coveredby , ON, Anyinteract, or directly use coordinates to show out the destination.

In the following sub-paragraphs, we will have a quick overview on the indexing structures that are wildly discussed in past years. This will help to derive the approach in this thesis.

3.1 R-Tree

Most commercial databases set the R-tree [1] as default data structure for spatial fundamentals, so we will take some pages to describe the operation of R-tree in details in this section

.

The R-tree is a height-balanced tree similar to the B-tree. In the R-tree, leaf nodes contain index records of the format (*Data*, *tuple*) where *tuple* uniquely determines a tuple in the database and determines a bounding rectangle of the indexed spatial object. The actual data objects of the node can have arbitrary shapes. Non-leaf nodes contain entries of the form (*Data*, *child*) where *child* refers to the address of a lower node in the R-tree and *Data* is the smallest bounding rectangle that contains the bounding rectangles of all of its children nodes. Assuming every leaf or non-leaf node contains between m ($m = M/2$) and M index records unless it is the root. See Figure. 1 for example of (1, 3) R-tree. Figure. 2 shows different (1, 3) R-tree on the same data set. When we have R-tree operation, it can fit into secondary storage (i.e. hard disk). Each node of the R-tree is placed on a separate disk page. This makes the R-tree particularly useful for applications involving very large object data where the index is too large to fit in memory.

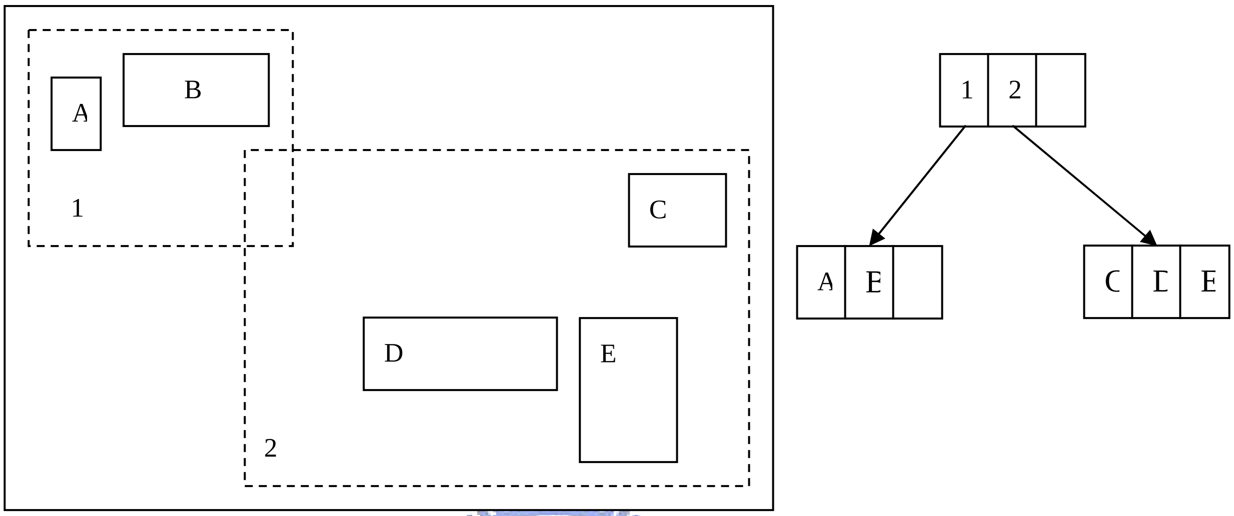


Figure 1. R-Tree with data set

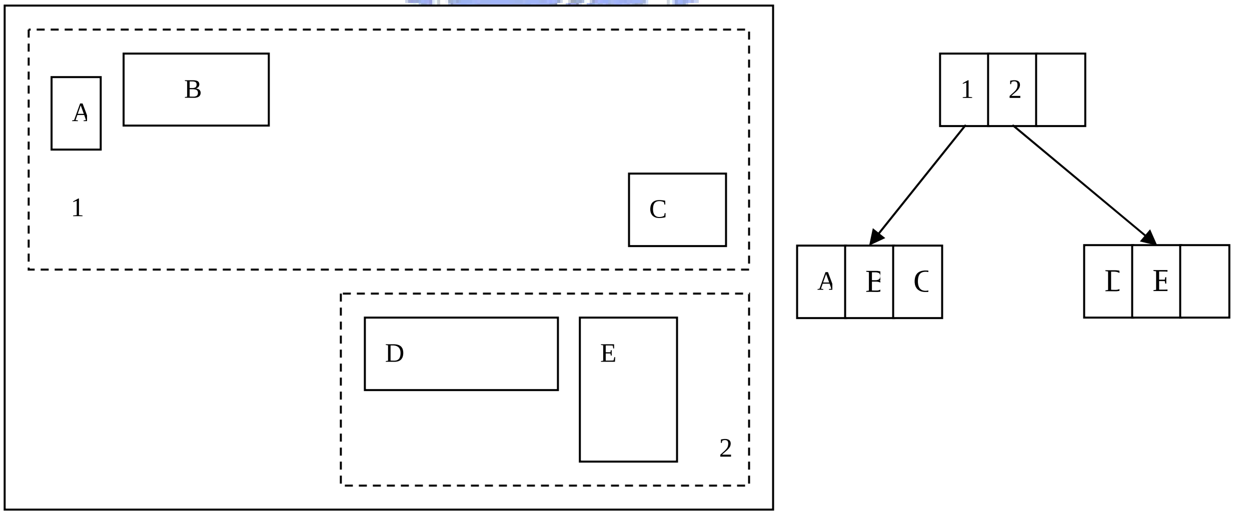


Figure 2. R-Tree with same data in Figure 1.

The R-tree [1] satisfies the following properties:

- Every non-leaf node has between m and M children unless it is the root.
- Every leaf node contains between m and M index records unless

it is the root.

- The root node has at least two children unless it is a leaf.
- All leaves appear at the same level (the tree is balanced).

Andrei Radu Popescu [23] introduced a good approach to calculate the degree of the R-tree. Assume an R-tree that indexes *N* spatial objects has the following maximum number of nodes:

$$\text{HMax} = \left\lfloor \frac{N}{M} \right\rfloor + \left\lfloor \frac{N}{M^2} \right\rfloor + \ldots + 1 \qquad (1)$$

The justification is that the worst-case space utilization for a node (except the root) is *N/m*. Hence, every node holds exactly m entries. Since there are N indexed records, it means that in the worst case (which implies the maximum number of nodes) the last level of the tree contains *N/m* leaves. The level immediately above the leaf level has nodes containing each *m* entries. All these nodes at this level have as children the *N/m* leaves. Therefore, this level has

$$\frac{\left(\frac{N}{m}\right)}{m} = \frac{N}{m^2}$$

Nodes, and so forth, until the root level which contains only one node.

The height of the same R-tree (that indexes N spatial objects) is at most Hmax, where

$$\text{HMax} = \lfloor \log_m N \rfloor - 1 \qquad (2)$$

Here the justification follows from the one for the Eq (1). The first level after the root has exactly m nodes. But from Eq. (1), the same level has $\lfloor N/m^{Hmax} \rfloor$. Hence,

$$\left\lfloor \frac{N}{m^{H\max}} \right\rfloor = m \Rightarrow N = m^{Hmax+1} \Rightarrow \text{Hmax} = \lfloor \log_m N \rfloor - 1$$

An R-tree based index is completely dynamic in the sense that it allows concurrent searches and update operations. Also, no periodic reorganization is required.

A search in an R-tree starts at the root and descends the tree in a manner similar to a search in a B-tree. Due to the non-zero size of the query window, and possible overlap between bounding rectangles at each level of the tree, multiple paths from the root downwards may need to be traversed. The R-tree can be updated dynamically, by insertion or deletion of data objects at the leaves. The most typical example of a search is the one where the user asks for all objects overlapping a certain area. Since the MBRs stored in the index entries are allowed to overlap, the R-tree cannot guarantee that only one search path needs to be traversed. The challenge is to build up bounding boxes dynamically and then in a way that will minimize the number of paths needed for solving a search. Next section will give the introduction of searching in R-tree.

3.1.1 R-Tree: Search

Searching an R-tree is done generally by finding all index records whose MBRs overlap a search rectangle. The algorithm will start with the root and will follow recursively all entries in the current node whose MBRs overlap the search rectangle. At the end of each path, a leaf node is processed and the MBRs of all entries in that leaf are tested against the search pattern. The indexes with overlapping rectangles are stored as a list of result candidates.

A search that uses an R-tree index is a two-step process. The search algorithm described above is actually the filter step. The set of qualifying index will be further processed in order to determine the exact result. Let’s use an example in [1] to describe the behavior:

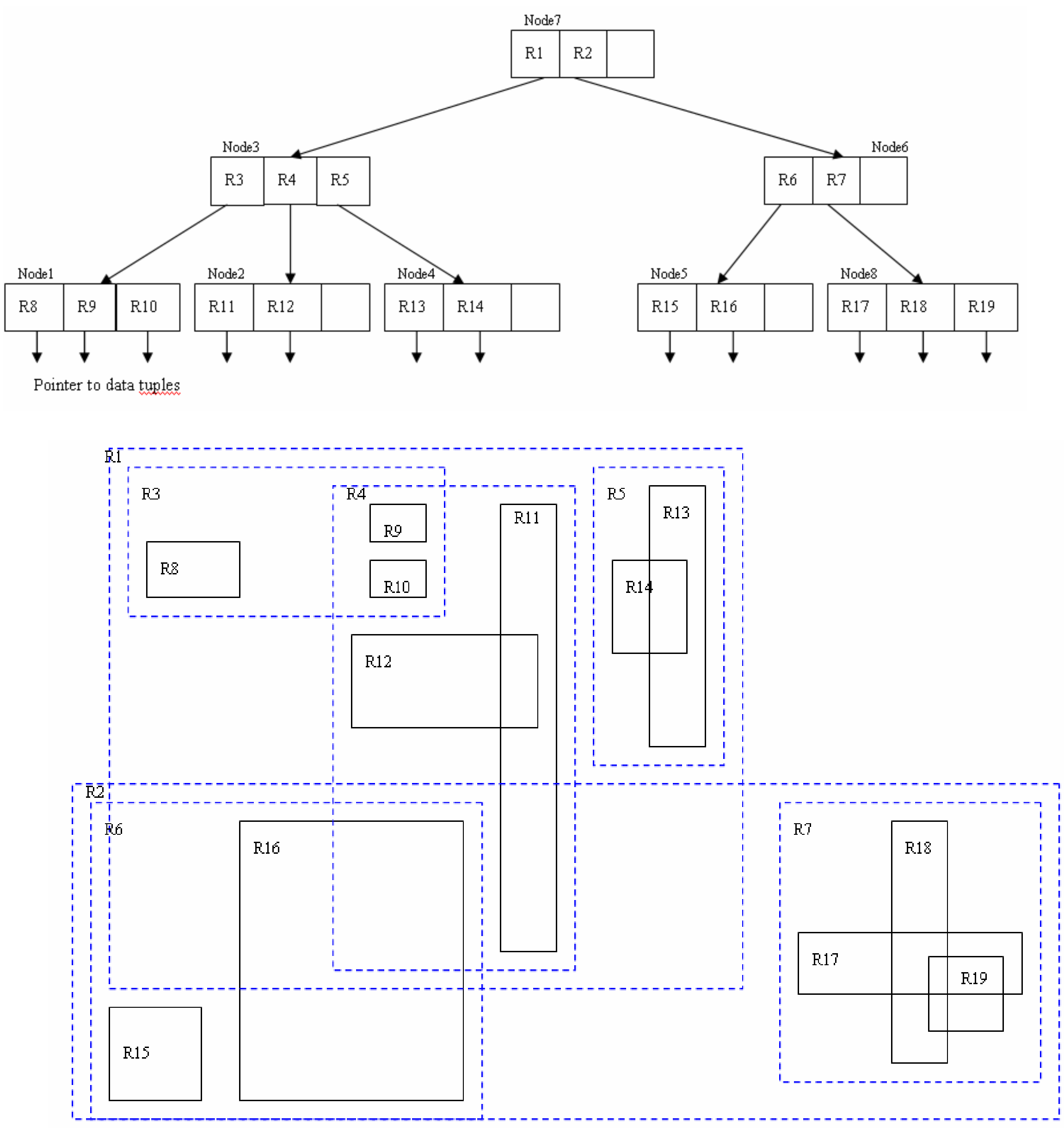


Figure 3. An R-Tree illustrates as MBR

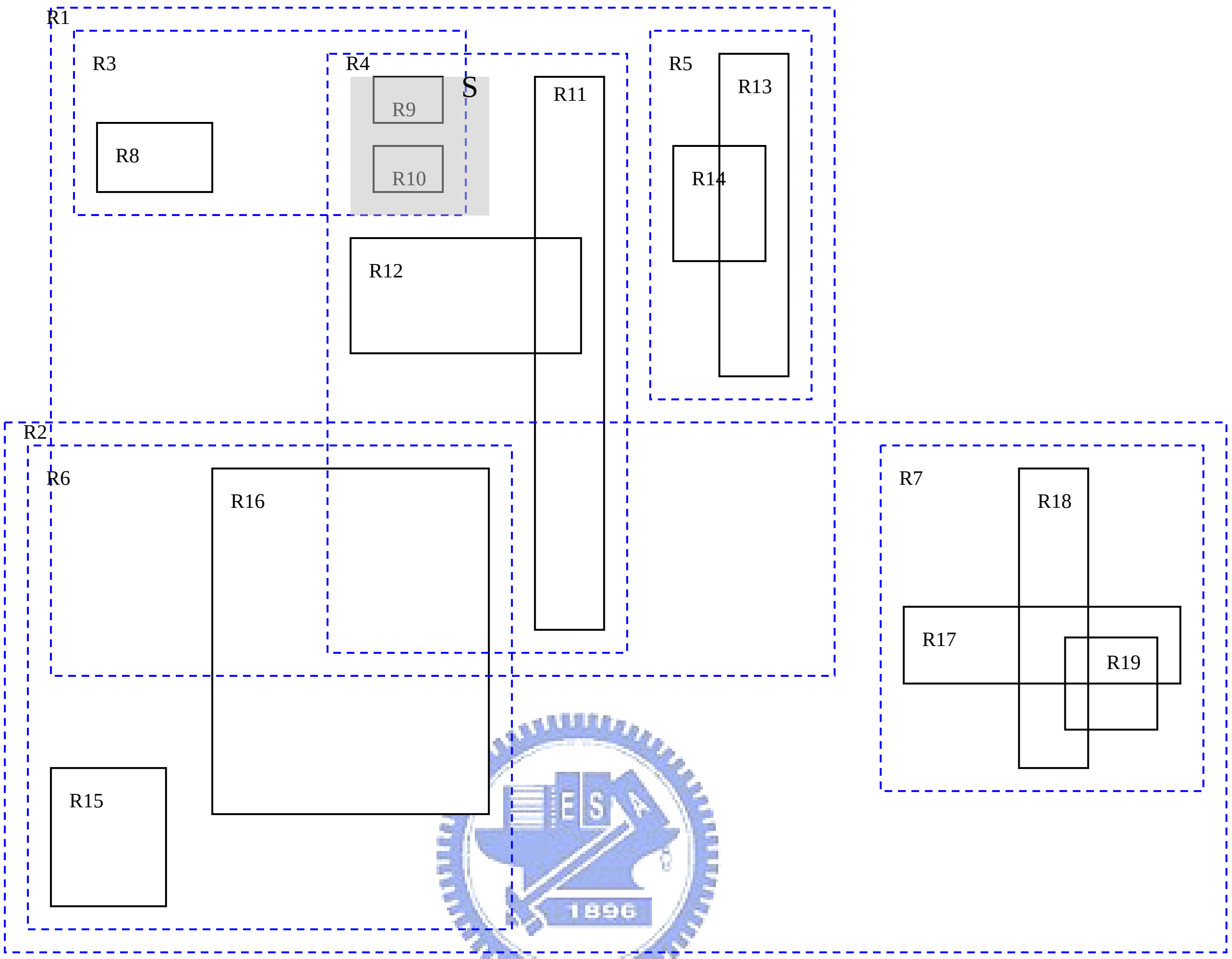


Figure 4. Search in the R-Tree

Suppose we will search for proper nodes. The process would start from the root Node7. During the process, a query could attempt to retrieve all objects that intersect the search rectangle S (as shown overlapped here). We find that S lies inside R1 and is disjoint with regard to R2. As a consequence, only the path rooted by the entry holding R1 will be followed. Next, the node Node3 will be searched. The first entry holds the MBR R3 which overlaps S. The algorithm follows the path rooted by this entry and descends to the node Node1 which is a leaf. A search in this leaf will retrieve the two entries holding the MBRs R9 and R10. Next, the algorithm backs up from the recursion to node Node3 and determines that R4 also contains S. Therefore, it descents to the leaf Node2 and finds that none of the entries contain MBRs that intersect S. Once again, the algorithm returns back to Node3 and since R5 and S are disjoint, it stops and produces the result set {R9, R10}.

3.1.2 R-Tree: Insert and Delete

The insertion algorithm in R-Tree is similar to insertion in the B-Tree. It creates the new index to the nodes or leaves that overflow which are split and the generated changes propagate up the tree. The first operation is to select a leaf where to place the new record. From the definition of the insertion in R-Tree:

1. **Find position for new record**

   Use the method to choose leaves to select a leaf node L in which to place New record

2. **Add record to leaf node**

   If L has room for another entry, install new record. Otherwise Split Node to obtain L and LL containing new record and all the old entries of L

3. **Propagate changes upward**

   Adjust R-Tree on L, also passing LL if a split was performed.

4. **Grow tree taller**

   If node split propagation caused the root to split, create a new root whose children are the two resulting nodes.

This will decide whether or not the MBR of all entries in the node overlap the search area. Therefore, the criterion used to decide where to place an entry consists of finding a distribution for the entries that will minimize their bounding rectangle. The leaf selection will start from the root and will follow the path denoted by the entries whose MBRs need least enlargement in order to contain the MBR of the new record.

For the insertion of the R-Tree algorithm, it adds new index records to the leaves, nodes that overflow are split and the generated changes propagate up the tree. The first operation is to select a leaf to place the new record. This is done by the Choose Leaf algorithm. During a search, the decision whether to visit a node depends on whether the MBR of all entries in the node overlap the search area or not. Therefore, the criterion used to decide where to place an entry (as well as how to split a node) consists of finding a distribution for the entries that will minimize their bounding rectangle. The leaf selection will start from the root and will follow the path denoted by the entries. Their MBRs need to have least enlargement in order to contain the MBR of the new record. The next operation is to add the record to the chosen leaf. If the leaf node already has *M* entries, then it needs to be split into two new nodes that contain *M+1* entries. It is desirable that the distribution of entries will be made such that subsequent searches do not need to examine both nodes. The criterion used here is similar to the one used by the Choose Leaf algorithm: minimize the area of the enclosing rectangle with the nodes inside of it.

An exhaustive method would try all possible groupings of entries and pick the one that generates the smallest bounding rectangles. However, the cost of such a method is of the order of 2*M*-1, so it cannot be used in practice. The founder of R-Tree [1] proposes two versions of the node splitting algorithm that will try to approximate the best result: LinearSplit and QuadraticSplit. These algorithms have a linear (*O(M)*) and quadratic (*O(M2)*) cost with respect to the maximum number of entries in a node. Both methods have been found to yield the same retrieval performance.

The R-tree algorithm may generate certain problems: after the first assignment, one of the two groups has a MBR larger than the other. Hence, it is easy to become that lesser area to accommodate the next entry is required, so it will be enlarged again. Over time, this will create a very uneven distribution, with most entries in one node. Also, when one of the groups becomes full, the rest of *M*-*m*+1 entries are assigned to the second group without any geometric considerations. For an example of a bad split where the MBR of one group is preferred over the other, see Figure 5. The seeds are marked with a gray filling pattern.

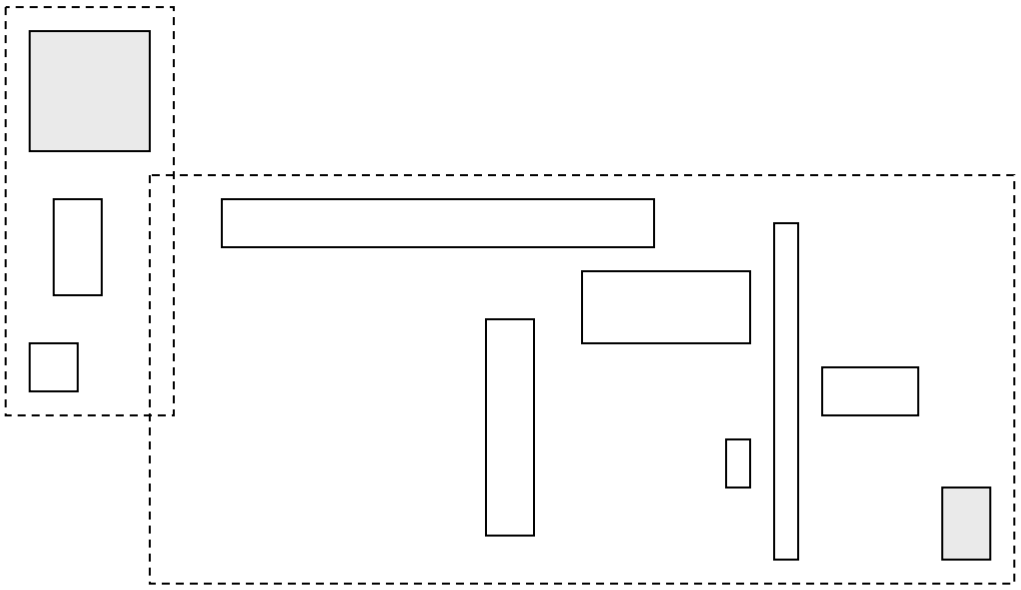


Figure 5. A bad example on split

The insertion algorithm will end with the propagation of the changes up the tree. If a new entry was added to a node, the MBR of all entries stored in the parent node, will need to be adjusted. If a node was split, a new entry will be added to the parent. If the parent is full, it will be split, and so on.

The algorithms for deleting an index entry roughly follow the same logic. First, the leaf containing the entry is identified and then the entry is removed. If the leaf becomes under-populated (number of entries < *m*), all remaining entries are saved in a list and the node is removed first. The changes are propagated up the tree by updating the corresponding covering rectangles and removing the nodes that become unproductively.

The result of the deletion algorithm is a list of orphan entries that will be reintroduced in the tree using the insertion procedure. The entries should be inserted at the same level of the tree where they originally belonged. [1] Guttman argues in his paper that this way of dealing with orphan entries (compared to the B-tree method of merging with sibling nodes) is better suited to the purpose of R-trees. Reinsertion refines the spatial structure of the tree, by correcting some possible initial misplacement or by simply creating over time a better distribution of the entries. Also, if the index is disk resident, the efficiency of the algorithm does not suffer since usually the pages that are needed for the insertion of the orphan entries are already buffered after the preceding search. So R-tree is still widely used in modern spatial database.

3.2 R*-Tree

Among various multi-dimensional access methods, the R*-tree, a variation of the original R-tree, published by N. Beckman and H.P. Kriegel, R. Schneider, B. Seeger [2] has been widely accepted by industry and research community. The R*-tree is a multi-dimensional extension of a B+-tree. Figure 6 illustrates an R*-tree indexing a set of points {a, b, c,…} (corner of the rectangles) of assuming a capacity of three entries per node. Points close in space (e.g., e, f, g) are clustered in the same leaf node (R6) represented as a minimum bounding rectangle (MBR). Nodes are then recursively grouped together following the same principle until the top level, which consists of a single root.

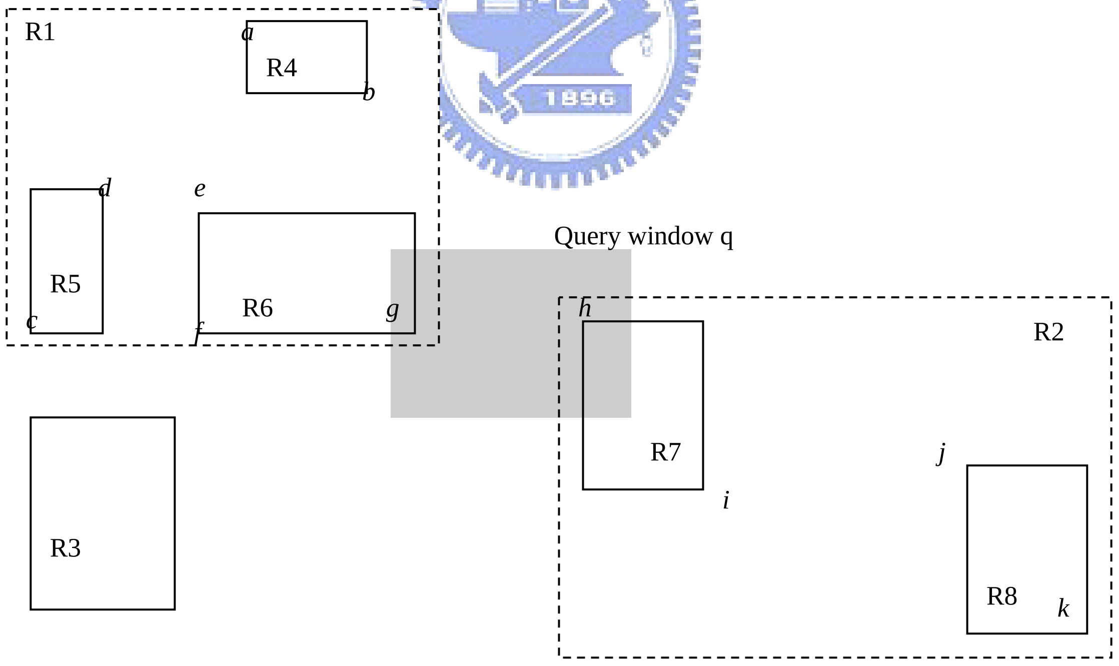


Figure 6. Query window

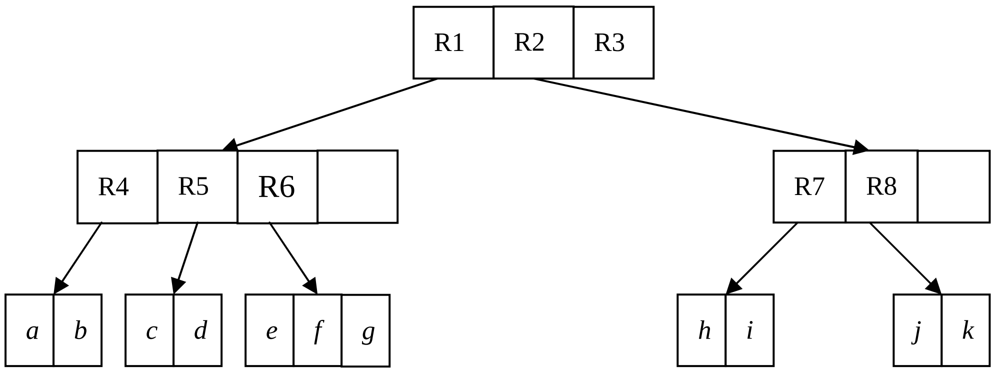


Figure 7. R*-Tree example for range query

R*-trees can efficiently answer various types of multidimensional queries, especially range query. Given a query window q (gray shadow in Figure 6), a range query retrieves all objects inside or intersecting q. Range queries can be processed using the original algorithm from R-Tree. Processing starts from the root level of the R*-tree. For any entry whose MBR intersects the query window, its sub-tree is recursively explored. If a leaf entry is encountered, all objects whose bounding rectangle intersects the query window are examined. Entries not intersecting the query window (e.g., R3) are not examined.

As shown in the figure 6 we can find R*-tree utilized forced reinsertion to improve the original R-Tree by reducing the MBR area and keep the shape of the MBR close to a square and using range query for the data. It can be arranged dynamically and proper fits to the disk-based and balanced tree structure. The root node corresponds to the largest MBR which contains the spatial extent of all objects. At one level down the MBR of every child of the root contains the spatial extent of all objects in the sub-tree and so on. R*-Tree attempts to reduce the coverage and minimize the overlap by using a combination of a revised node split algorithm and the concept of forced reinsertion at node overflow. This is based on the observation that R-tree structures are highly impacted to the order in which their entries are inserted, so a step-by-step insertion structure is to be sub-trees. Deletion and reinsertion of entries gives to find a node in the tree that may be more appropriate than their original location and more efficiently.

Take the overflow of nodes for example, a portion of its entries are removed from the node and reinserted into the tree to avoid the indefinite cascade of reinsertions caused by subset node overflow, the reinsertion routine may be called in each level of the tree once a time for new entry.

3.3 R$^+$-Tree

R$^+$-Tree, the relatives from R-Tree, announced by Timos Sellis, Nick Roussopoulos and Christos Faloutsos at 1987 [3], avoids overlapping of internal nodes by inserting an object into multiple leaves if necessary. An R$^+$-Tree satisfies the following special properties:

** (p, RECT):* An intermediate node is of the form where *p* is a pointer to a lower level node of the tree and *RECT* is a representation of the rectangle that encloses.

- For each entry *(p, RECT)* in an intermediate node, the subtree rooted at the node pointed to by *p* contains a rectangle *R* if and only if *R* is covered by *RECT*. The only exception is when *R* is a rectangle at a leaf node; in that case *R* must just overlap with *RECT*.

- For any two entries *(p_1, $RECT_1$)* and *(p_2, $RECT_2$)* of an intermediate node, the overlap between $RECT_1$, and p_2, $RECT_2$ is zero.
- The root has at least two children unless it is a leaf.
- All leaves are at the same level.

All leaves are located on the same level and the tree is still height-balanced. There are no constraints regarding the number of entries in a node which can be anything between one and the maximum node capacity. The only exception concerns the root that must have at least two children unless it is a leaf. Comparing R$^+$-Tree and R-Tree the first one would nodes are not guaranteed to be at least half filled and the entries of any internal node do not overlap. Besides, the object ID may be stored in more than one leaf node to ensure the construction could point query performance benefits since all spatial regions are covered by at most one node. In R$^+$-Tree, the path of searching a tuple may have shorter path and fewer nodes to visit than with the R-Tree. Let me take the example here: Give the diagram with R-Tree and try to find window W as below:

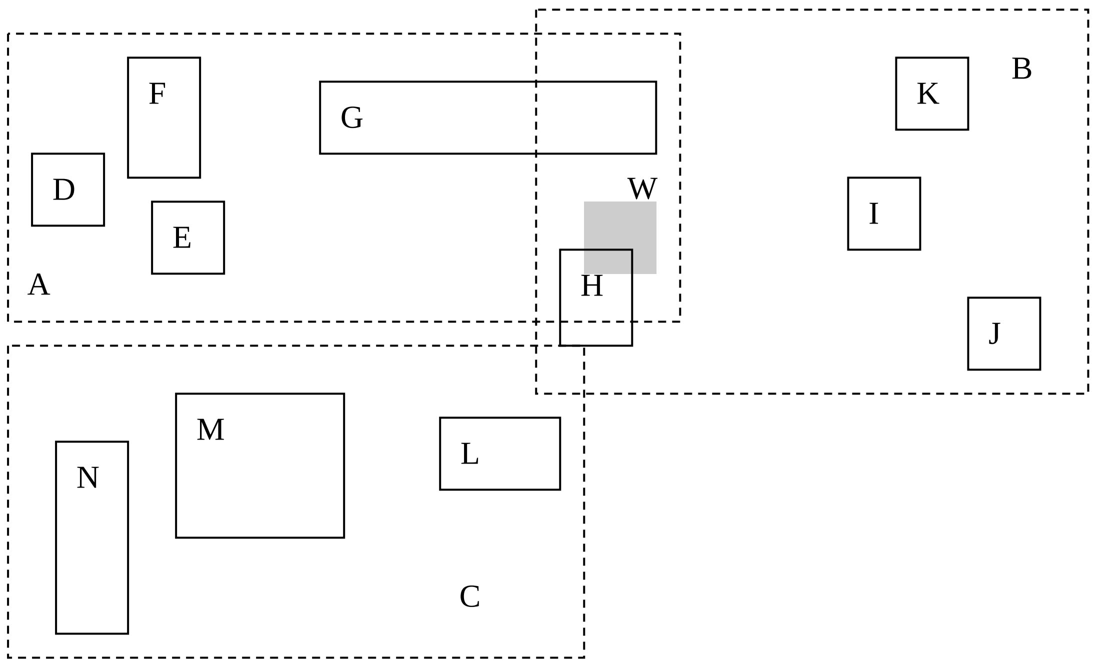


Figure 8. Query Window W

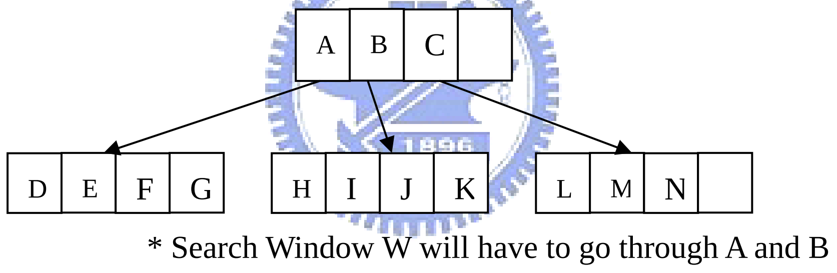


Figure 9. Search Window

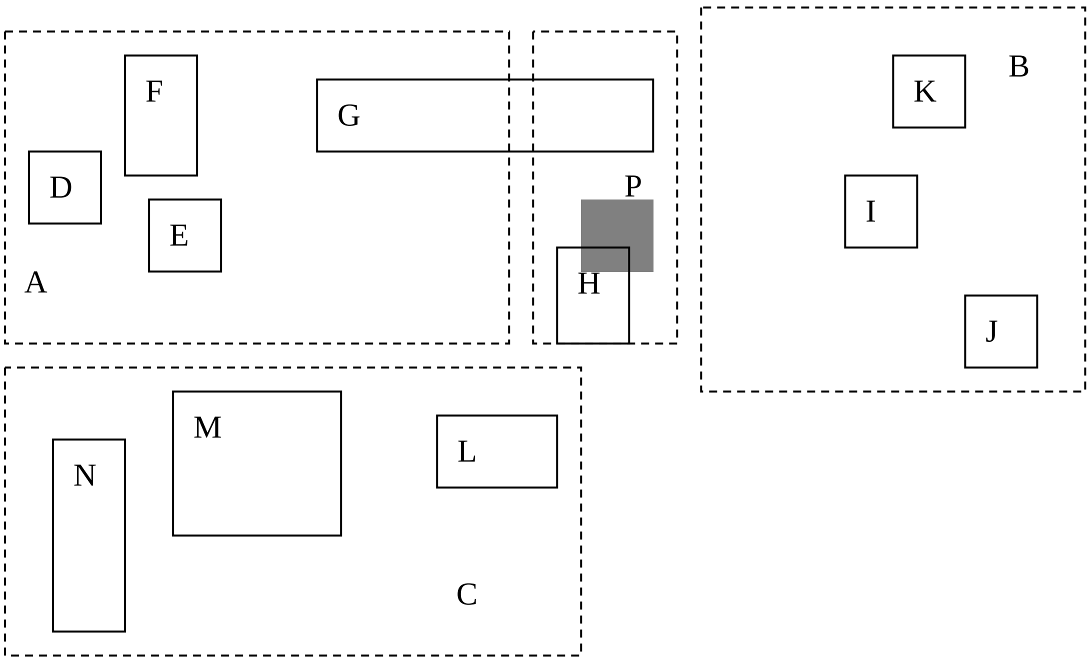


Figure 10. Revised using R+-Tree

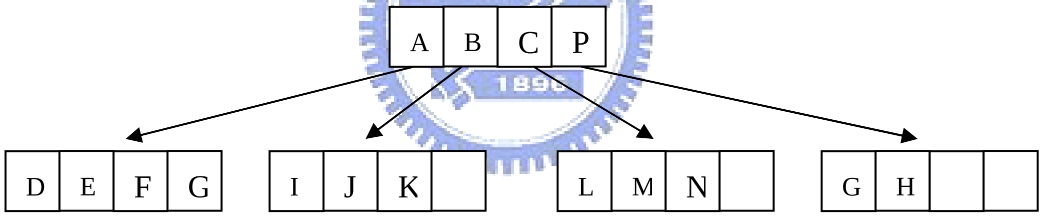


Figure 11.R+-Tree

So in this demonstration it shows the performance improvement comparing to R-Tree that it has to go over both A and B to find out the searching window W. In R+-Tree, we can accomplish this by using split the cell with non-overlapped structure for the enhancement. Since rectangles that we refer inr R+-Tree may be duplicated so it can be larger than original R-Tree built on the same data set. And in the meantime it is more complex to insert and delete for the operations. We will introduce the search and deletion briefly here.

To search R+-Tree is similar to R-Tree. The idea is to first decompose the search space into disjoint sub-regions and each of those descend until the actual data objects are found in the leaves. The major difference with R-trees is that in the latter sub-regions can overlap, thus leading to more expensive searching. Deletion is similar to KDB Trees [4] that subtrees should be periodically re-organized to achieve better performance. In [1], it also suggests a similar procedure where under-utilized nodes are emptied and become as orphan rectangles. They are re-inserted at the top of the tree. Besides, it uses the partition and pack algorithm recursively to traverse the entries of each level of the tree so that it can have better performance on searching. R+-Tree results of the comparison agree completely with the intuition: R-trees suffer in the case of few, large data objects, which force re-insertion during the search. R+-trees handle these cases easily, because they describe these large data objects into smaller ones.

3.4 Variants of R-Tree

After these major R-tree structures we do believe developing efficient index structures is an important research for spatial databases. From the study, multi-dimensional spatial index structures can be used for indexing the positions of objects. Numerous index structures have been proposed for indexing multidimensional data. In [5] Ravi Kanth et Al. have published a paper to discuss Quadtree and R-tree indexes in commercial database and indicate that R-trees are generally better than Quadtrees.

Most commercial spatial databases now recommend using R-tree only. Although traditional spatial index structures can be used, they are not efficient for indexing the positions of spatial objects because of frequent and numerous update operations in moving environments. Because some new index structures have been proposed for indexing objects recently. These index structures can be divided into the two categories: (1) the histories and (2) the current positions of objects as below:

In the first category, object in a d-dimensional space is converted into a trajectory in a (d+1) dimensional space when time is treated as a dimension. Examples of approaches are the Spatio-Temporal R-tree (STR-tree) [6] and Trajectory-Bundle tree (TB-tree) proposed [7] to fulfill the histories that may effect the object allocation. The authors showed that these two structures work better than traditional spatial index structures for queries related to trajectories. In [8], Tao and Papadias have proposed the Multi- version 3D R-tree (MV3R-tree), which combines multi-version B-trees and 3D-Rtrees to help on spatial database with multi-dimension. Also most of recent studies will implement the Hilbert Tree[19] and SR-Tree[20] for better performance improvement [18].

In the second category, most approaches describe an indexing in the spatial database by a linear function, and only when the parameters of the function change during database updated. The time-parameterized R-tree (TPR-tree) has been proposed in by Saltenis [9]. In this scheme, the position of a object is represented by a reference position and a corresponding velocity vector. When splitting nodes, the TPR-tree [9] considers both the positions of the spatial points. All these techniques rely upon a good representation of the future spatial objects. In many applications, the spatial objects is complicated and non-linear. In such situations, the approaches based on a linear function cannot work efficiently – the function changes too often. However, simple indexing cannot efficiently handle the rapid new queries. Also there are some research papers using hybrid and combination of the Quad-tree and R-tree together to leverage the advantage of each one – Quad-tree is good when updating the index, but its query performance is worse than that of the R*tree. R-Tree gives good query performance but poor index update performance. So Yuni Xia et al. [10] submitted the Q+RTree achieves a better performance for both index updating and query evaluation by handling different types of objects separately.

Most of the approaches of indexing in the spatial database will leverage the R–tree structure to filter out the best path when searching with index so that it can sort out rapidly. And also the MBRs (Minimum Boundary Rectangle) should be small and their shapes should be close to squares so that MBRs can have higher quality for indexing. Recent research has been developed among the input and re-constructs the data structure to provide the improvement in the structure.

3.5 Structure of implementation in Oracle10g Spatial database

Oracle has been widely implemented in the commercial environment and spatial is the option for Oracle10g Enterprise Edition. Spatial is the fundamental primitive in the traditional database such as GIS, CAD/CAM, telecommunication, multimedia and location-based applications. So Oracle also provided Spatial features and recently in 10g it introduced a GeoRaster [13] datatype to store and manage image and gridded raster data and metadata, network and topology data models, geo-coding and routing engines, and spatial analysis and mining functions. Oracle also provided the Java API [14] for quick deployment named eLocation so that the developer can come out the design on object-oriented program to support different scale program and data so that can establish the scope of the objects in spatial database. From the Oracle10g Spatial official documents [11] [12][13][14] we can easily develop and display our own spatial objects quickly.

The reason that I am using Oracle in my thesis because the Oracle provided a lot pre-defined objects to handle VLOB(Very Large Object) For instance, by default when you create a spatial index without specifying any quadtree-specific parameters, and R-tree index is created. For the ease of operations and query efficient, Oracle recommends R-tree as default and listed in user guide [11] saying not to use Quad-tree for indexing. For example, the following statement creates a spatial R-tree index named territory_idx using default values for parameters that apply to R-tree index:

CREATE INDEX territory_idx ON territories (territory_geom)
INDEXTYPE IS MDSYS.SPATIAL_INDEX;

R-tree indexes can be built on two, three or four dimensions of data. The default number of dimensions is two, but if the data has more than two dimensions, you can use the SDO_INDX_DIMS parameter keyword to specify the number of dimensions on which to build the index. However, if a spatial index has been built on more than two dimensions of a layer, the only spatial operator that can be used against that layer is SDO_FILTER (the primary filter or index-only query), which considers all dimensions. The SDO_RELATE, SDO_NN (for neighbors), and SDO_WITHIN_DISTANCE operators are disabled if the index has been built on more than two dimensions.

In Oracle10g Spatial database, we can also create the index for query efficiency just like the relational database. Once the index is determined and generated, the data will be loaded and grows rapidly and the performance will be decreased with spatial data. It will be impacted during data query. To solve this, Oracle also provided the enhancement for indexing in Oracle10g database:

1) Indexing Geodetic Data to ensure the entire index in Geodetic R-tree. Geodetic data consists of geometries that have geodetic SDO_SRID

values, reflecting the fact that they are based on a geodetic coordinate system (such as using longitude and latitude) as opposed to a flat or projected plane coordinate system.

2) Constraining Data to a Geometry Type: When you create or rebuild a spatial index, you can ensure that all geometries that are in the table or that are inserted later are of a specified geometry type. To constrain the data to a geometry type in this way, use the layer_gtype keyword in the PARAMETERS clause of the CREATE INDEX or ALTER INDEX REBUILD statement, and specify a value from the Geometry Type column. i.e.

    CREATE INDEX area_spatial_idx
    ON area_markets(shape)
    INDEXTYPE IS MDSYS.SPATIAL_INDEX
    PARAMETERS ('layer_gtype=POLYGON');

3) Using Partitioned Spatial Indexes:

A partitioned spatial index can provide the following benefits:

- Reduce response time for long-running queries, because partitioning reduces disk I/O operations
- Reduce response time for concurrent queries, because I/O operations run concurrently on each partition
- Easier index maintenance, because of partition-level create and rebuild operations
- Index on partitions can be rebuilt without affecting the queries on other partitions, and storage parameters for each local index can be changed independent of other partitions.
- Parallel query on multiple partition searching

- The degree of parallelism is the value from the DEGREE column in the row for the index in the USER_INDEXES view (that is, the value specified or defaulted for the PARALLEL keyword with the *CREATE INDEX*, *ALTER INDEX*, or *ALTER INDEX REBUILD* statement).
- Improve query processing in multiprocessor system environments

In a multiprocessor system environment, if a spatial operator is invoked on a table with partitioned spatial index and if multiple partitions are involved in the query, multiple processors can be used to evaluate the query. The number of processors used is determined by the degree of parallelism and the number of partitions used in evaluating the query. The following restrictions apply to spatial index partitioning:

- The partition key for spatial tables must be a scalar value, and must not be a spatial column.
- Only range partitioning is supported on the underlying table. Hash and composite partitioning are not currently supported for partitioned spatial indexes.
- To create a partitioned spatial index, you must specify the LOCAL keyword. For example:

  CREATE INDEX counties_idx ON counties(geometry)

  INDEXTYPE IS MDSYS.SPATIAL_INDEX LOCAL;

In summary of this section, Oracle10g Database Spatial provides several methodologies to increase the performance of the indexing in spatial database, but it still can be improved from the fundamental structure of R-tree and Quad-Tree through the methodology implemented here.

Chapter 4 A new indexing method in spatial database

4.1 Introduction

Although most of spatial database used several indexing algorithms on the structure and improve the efficiency by variety of works, we still can find some approach to enhance current latest version of Oracle10g Spatial database. In this thesis, I am going to use bulk loading of data to insert the data objects into database and then linear hash will improve the query processing by partitions. Also the indexing performance is better comparing to the original R-tree structure. And we also implement bulk-loading[15] into this approach for additional enhancement together.

The bulk-loading framework can be issued like this:

INSERT INTO <TABLE>   (sub-query)

When use with (sub-query) in SQL statement, it will calculate and execute sub-query first. Any data resulting from the sub-query are grouped into 64K-memory-size batches and inserted into the spatial indexes. In addition to the above array-insert interface through sub-queries in an insert SQL statement, Oracle Spatial provides the following defered-indexing model for performing bulk insertions:

1. User alters the index to be deferred.
2. Subsequent inserts into the index and they are stored in a separate deferred table associated with the index.

3. User alters the index to synchronize. An optional batch size is also specified by the user. This operation retrieves all insertions in the "deferred" table and inserts them in batches of specified size into the index. Optionally, a quad-code could be computed for the centers of the MBRs of these geometry data and can be used to order the data being fetched and inserted in batches into the spatial index.

The batch size for bulk insertion in the deferred indexing model is controlled by the user. In contrast, for the SQL array insert it depends on internal memory chunk-size parameters set for the database.

Once we can use the bulk loading of data into database to enhance the original sequential insertion, now we start to think the improvement in the spatial database. Every time the data inserted into database will be divided into 64K memory size and consolidate recursively until all data has been proceeded. From [15] indicates that uses bulk loading reduced the numbers of traversals of R-Tree and also improves the quality of the constructed R-tree index by reorganizing overlapping sub-trees. Ideally the bulk loading will enhance 50~90% comparing to one by one loading. Now move forward to see what we can improve during bulk loading data.

Originally we will see the data aggregated with 64K-memory-size, so every time we load batch dataset then we start to break the loading sequence and so on. Once we tune with bulk loading data in range query, we can see the improvement on bulk loading efficiency. But this will increase the I/O consumption during operation. But we still can identify this is good approach during loading a great amount of data but maybe invalid when small transaction is in place. So ideally we need to think about from the structure to build up a proper framework so that we can leverage the current spatial database to prevent the extra misleading of data construction. Now we shall move to the combination of bulk-loading and hash together to create the performance gains in new approach in spatial database.

4.2 Methodology and approaches

Here is the new approach in each

1) Hash the original data objects into hash buckets
2) Solve the collision and overflow (if any)
3) Use bulk loading to partition separate regions recursively with data segment during insertion of hashing
4) CREATE INDEX within the R-Tree structure
5) Comparing performance of query to the original operation with R-tree.

4.2.1 Original Operation

I would like to demonstrate the original operation on spatial database. Spatial data can be divided into 3 categories: Point feature, Line feature and Polygon/Area feature and also consisted with attribute data to describe the content of the object data. For example, the most common referenced utility to describe is included coordinates (X, Y) and location, shape, besides we can use attribute data to illustrate name of the road, width of the road, numbers of lanes, traffic and etc.

Let’s discuss about the metadata first. Assume we are using the geo sample data from TIGER to make sure the comprehensive data was inserted in our way. Before applying the algorithm, we need to define the data structure to construct the fundamentals. For example, we have some sample data:

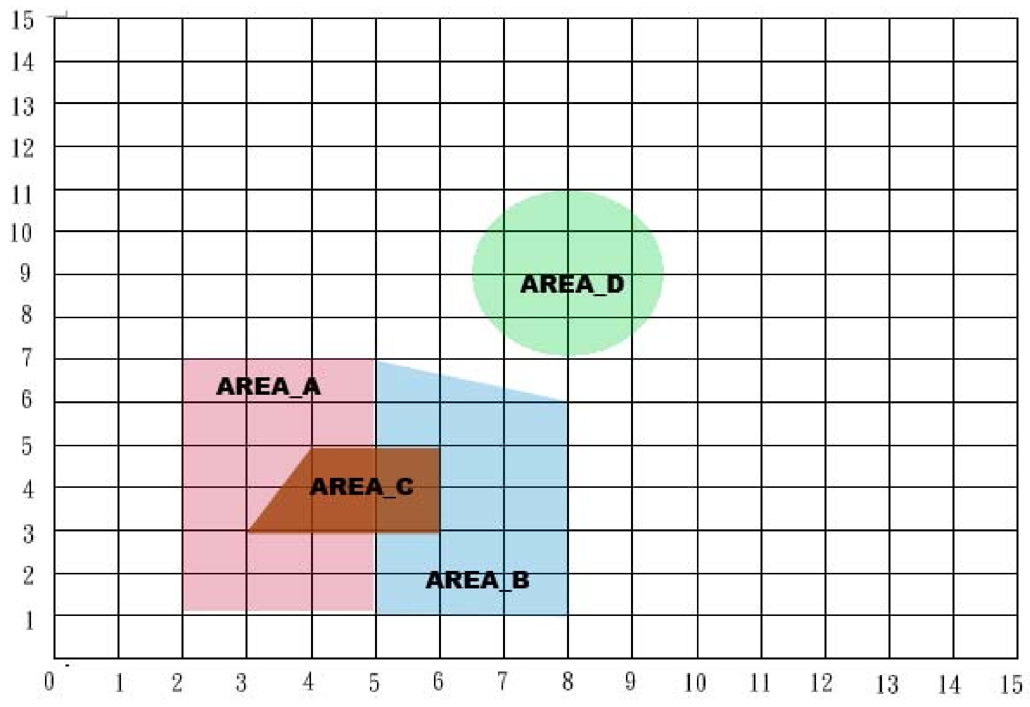


Figure 12. Example on spatial objects

We may need to construct the metadata for the data. (1) Creates a table (AREAS) to hold the spatial data (2) Inserts rows for four areas of interest (AREA_A, AREA_B, AREA_C, AREA_D) (3) Updates the USER_SDO_GEOM_METADATA view to reflect the dimensional information for the areas Creates a spatial index (AREA_SPATIAL_IDX)

(4) then performs some spatial queries.

In step (1), we create a table for area (A, B, C, D) in a given geography (such as city or state). Each row will be an area of product for a specific area (for example, where the area is most preferred by residents, where the manufacturer believes the area has growth potential, and so on).

```
CREATE TABLE area_markets (
    mkt_id NUMBER PRIMARY KEY,
    name VARCHAR2(32),
    shape SDO_GEOMETRY);
```

The next INSERT statement creates an area of product for AREA A. This area happens to be a rectangle. The area could represent any user-defined criterion: for example, where AREA A is the preferred product, where AREA A is under competitive pressure, where AREA A has strong growth potential, and so on.

```
INSERT INTO area_markets VALUES(
    1,
    'AREA_A',
    SDO_GEOMETRY(
        2003,
        NULL,
        NULL,
        SDO_ELEM_INFO_ARRAY(1,1003,3),
        SDO_ORDINATE_ARRAY (1,1, 5,7)
    )
);
```

The next INSERT statements demonstrate how to create areas of interest for AREA B and AREA C. These areas are simple polygons (but not rectangles).

```
INSERT INTO area_markets VALUES(
    2,
    'AREA_B',
    SDO_GEOMETRY(
        2003,
        NULL,
        NULL,
        SDO_ELEM_INFO_ARRAY(1,1003,1),
        SDO_ORDINATE_ARRAY(5,1, 8,1, 8,6, 5,7, 5,1)
    )
);
INSERT INTO area_markets VALUES(
    3,
    'AREA_C',
    SDO_GEOMETRY(
        2003,
        NULL,
        NULL,
        SDO_ELEM_INFO_ARRAY(1,1003,1),
        SDO_ORDINATE_ARRAY(3,3, 6,3, 6,5, 4,5, 3,3)
    )
);
```

Now insert an area of interest for AREA D. This is a circle with a radius of 2. It is completely outside the first three areas of interest.

```
INSERT INTO area_markets VALUES(
    4,
    'AREA_D',
    SDO_GEOMETRY(
        2003,
        NULL,
        NULL,
        SDO_ELEM_INFO_ARRAY(1,1003,4),
        SDO_ORDINATE_ARRAY(8,7, 10,9, 8,11)
    )
);
```

Thus we are going to create the index

```
CREATE INDEX area_spatial_idx
    ON area_markets(shape)
    INDEXTYPE IS MDSYS.SPATIAL_INDEX;
```

Above statement will create an R-tree index with these objects. I will treat this as experimental group for original sequential loading and R-tree construction.

4.2.2 New approach

In order to have comprehensive plan on this approach, first we will start to load bulk data by segment with a high balanced hash table. Next we will construct the dataset with unique identifier and content of the data.

```
CREATE TABLE hash_data (
    obj_id NUMBER PRIMARY KEY,
    name VARCHAR2(32),
    shape SDO_GEOMETRY);
```

Once the object table is been created and we will start to think about the approach on how to increase the performance with hash function. There are several methods of hashing that can describe in the study. We will use the linearly growing and shrinking method to continuous address the space of memory allocation because the database will allocate in the SGA (server global area), Share Pool Area and operation system will take the consideration too.

Here is the pseudo procedure of the approach:

```
DECLARE
        geom SDO_geometry :=
        SDO_geometry (2003, null, null,  SDO_elem_info_array
        (1,1003,3), SDO_ordinate_array (-109,37,-102,40));
        BEGIN
            INSERT_GEOM(geom)
            BEGIN
```

```
          SELECT A.Feature_ID FROM TARGET A
          WHERE sdo_filter(A.shape, DO_geometry
                    (2003,NULL,NULL,
                    SDO_elem_info_array(1,1003,3),
                    SDO_ordinate_array(x1,y1, x2,y2))
                    ) = 'TRUE';
          partition by LINEAR HASH(geom);
          CREATE INDEX hash_idx ON GEOM(geom)
          INDEXTYPE IS MDSYS.SPATIAL_INDEX;
          END
COMMIT;
END;
```

Let me use this as an example. Assume we have 10 data object will be inserted into Hash table before indexing into R-tree. We can set these parameters to ensure the data is been enclosed as the method above:

- *n*: numbers of buckets
- *h*: height of the hash buckets (collision/overflow control)
- *k*: $k = \lfloor \log_2 n \rfloor$, we will start to place data from middle of the hash table
- *r*: $r = n - 2^k$
- *H*: Hash function
- *H(x)*: Generate the pseudokey to determine the location of buckets to be placed.

An address space of n buckets is divided into two areas of sizes 2^k and $r = n - 2^k$ buckets, where $k = \lfloor \log_2 n \rfloor$. A randomizing hash function H is first applied to the access key (x), producing a so-called pseudokey, $H(x)$. The final address is extracted from the tail of $H(x)$ as follows:

Calculate a = $H(x)$ mod 2^k; if a < r, recalculate a = $H(x)$ mod 2^{k+1}.

It is easy to see that the range of a is $[0 \ldots 2^k + r - 1]$, as required. A simple interpretation of the situation is that the first r buckets have been split on the basis of the (k+ 1) 'st bit (from the rear) in the pseudokeys: objects with a 0-bit have remained in their original bucket (p), but those with a l-bit have been moved to bucket $2^k + p$.

Once we retrieve from the table and obtain the coordinates from SDO_Geometry, we can easily calculate point-to-point relatively and choice the best strategy onto the hash buckets. Of course it is possible to come out of the same destination after hash function, we called the symptom "splitting". There are two main criteria for splitting:

1) *when a bucket overflow occurs*
2) *when the overall load exceeds a given threshold.*

The latter is better in the sense that it can be used also in deciding about the opposite operation, joining of buckets, when the load drops below another threshold. Splits are performed in a strict linear order, and the bucket causing an overflow does not usually get split at once. Therefore, linear hashing regularly requires overflow buckets. In splitting a bucket, the related overflow records are naturally taken along, too. To save space, overflow buckets can be shared by several home buckets.

When splitting proceeds to the (2^k - 1)'st bucket, all leading 2^k buckets have been split. Then k is increased by one, and r reset to zero.

Now We will try to load data from the dataset we defined previously. We will start to aggregate the total amount by count (*) of the table for this example and assign to the parameter n. Two approaches to determine n if we are adapting the linear hashing:

1) Prime number that closed to the amount of dataset – which can reduce collision but may take a lot of space. In the other hand, it is more efficient for indexing and searching.
2) Find out the 2^k, and the buckets will divide into 2^k and $r = n - 2^k$ buckets, where $k = \lfloor \log_2 n \rfloor$. The benefit of this will use the space less comparing to 1) and can generate the hash table quickly with existed mathematics functions so I will demonstrate my approach with this one.

Although it takes spaces for the computing but we can prevent the collision that we need to handle during this example. We will have further discussion on collision at the next section in details.

We generate the parameters:

$n = 11$,

$k = \lfloor \log_2 10 \rfloor = 3$,

$r = 11 - 2^3 = 3$,

$a = H(x) \bmod 2^k$;   if $a < r$, recalculate $a = H(x) \bmod 2^{k+1}$

Then we can have the

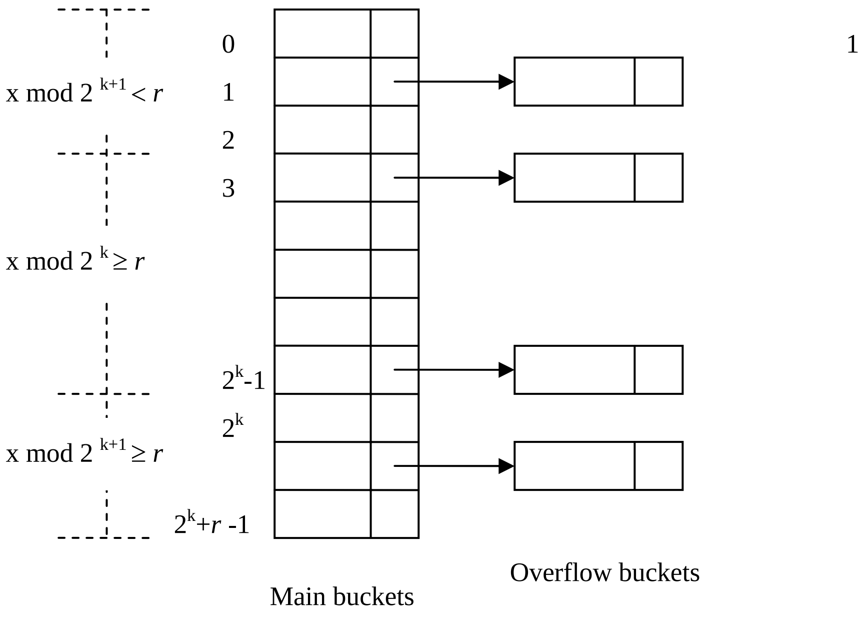
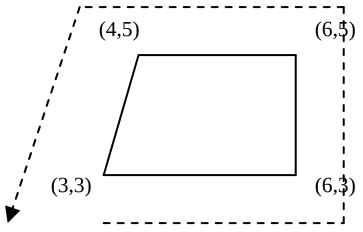


Figure 13. Linear Hash with k, r =3

Once the structure is been built up, we can simple obtain the whole hash tables and insert into corresponding dataspace with the sequential of this at the same time as following:



```
INSERT INTO hash_data VALUES(
    Object_number,               //Object number
    'Object_Name',               //Object Name
    SDO_GEOMETRY(
        2003,
        NULL,
        NULL,
        SDO_ELEM_INFO_ARRAY(1,1003,1),
        SDO_ORDINATE_ARRAY(3,3, 6,3, 6,5, 4,5, 3,3)
    )
);
```

Once the table has been inserted with data, we can move forward to creating the index by Oracle Spatial function:

CREATE INDEX hash_idx ON territories (hash_data)INDEXTYPE IS MDSYS.SPATIAL_INDEX;

I do not specify the SDO_INDEX_TYPE (which contains a VARCHAR2 variable, QTREE means a quadtree index, RTREE indicates for an R-tree index as default) so that this command will create the R-tree index by default. And I also disable the operators SDO_RELATE, SDO_NN, and SDO_WITHIN_DISTANCE because the index was not been built on more than two dimensions.

Once this is done, we can create 2 tablespaces for different operation on 1) original indexing with regular R-tree, 2) New approach in this thesis that the details of the operation will be included in the next section.

4.3 Collision and overflow

In the small number of cases, where multiple keys map to the same integer, then elements with different keys may be stored in the same "slot" of the hash table. It is clear that when the hash function is used to locate a potential match, it will be necessary to compare the key of that element with the search key. But there may be more than one element which should be stored in a single slot of the table. Various techniques are used to manage this problem: chaining, overflow areas, re-hashing, using neighboring slots (linear probing), quadratic probing, ...

1) **Chaining:** One simple scheme is to chain all collisions in lists attached to the appropriate slot. This allows an unlimited number of collisions to be handled and doesn't require a prior knowledge of how many elements are contained in the collection. The tradeoff is the same as with linked lists versus array implementations of collections: linked list overhead in space and, to a lesser extent, in time.
2) **Re-hashing:** Re-hashing schemes use a second hashing operation when there is a collision. If there is a further collision, we re-hash until an empty "slot" in the table is found. The re-hashing function can either be a new function or a re-application of the original one. As long as the functions are applied to a key in the same order, then a sought key can always be located.
3) **Linear probing:** A simple re-hashing scheme in which the next slot in the table is checked on a collision.
4) **Quadratic probing:** A re-hashing scheme in which a higher (usually 2nd) order function of the hash index is used to calculate the address.

In my case I will use the linear probing for the operation so that every time when there is a collision, we will insert the object into next available slot of the hash table. Quick and save a lot of time but still could be harmful because the slot may have collision again which will increase the time and affect the performance. But from the pseudo code of the procedure, I don’t see a serious impact from the data and can solve the problem in time.

Chapter 5 Implementation

5.1 Environment setup

Now we shall move forward to the environmental setup of the approach. In this set of experiments, we install Oracle10g Enterprise Edition (with spatial feature) by defining characterset in AL32UTF8 which is a unicode variable width multibyte characterset on a server of Dual Processor Intel Xeon 2.8GHz; 4 GB DDRII memory with SCSI Hard disk 120GB in my LAB. Besides, we also use datasets provided by Oracle website with minor modification and try to store that in the MDSYS tablespace by creating the same format as above.

Before the preparation, we have to run a SQL command

SQL> SET TIMING ON

to measure the length of operation of each command to compare the difference between original designs to the new approach with hash function. This will help to track all the time consumption in 00:00:00.00 format. We also prepare the store procedure to operate for the behavior we mentioned in previous section.

A set of experiments with varying workloads were performed in order to compare the relative performance of the R-tree in Oracle10g. I am using the sample data from OTN (Oracle Technology Network) with Hillsborough County, New Hampshire road network data that is converted from US Census Tiger Data. Oracle Spatial Network Data Model (abbreviated to NDM) has been used which is one of the many features provided with Oracle Spatial 10g. A network or graph captures relationships between objects using connectivity. NDM supports both directed and undirected networks, which can be either spatial or logical. Spatial networks contain both connectivity information and geometric information. Logical Networks contain connectivity information but no geometric information. NDM consists of two components: a network database schema, and a Java API. The network schema contains network metadata and tables for nodes and links. The Java API enables network representation and network analysis. I am using the Java API to demonstrate in my approach with the PL/SQL and store procedure in the Oracle10g database so that it is very simple to manage the process and approach of this. And at the same time I use the Oracle10g Mapviewer to query the specific data retrieving from Oracle10g Spatial Database and display this with Oracle10g Application Server (OC4J) so that we can obtain the whole picture on details.

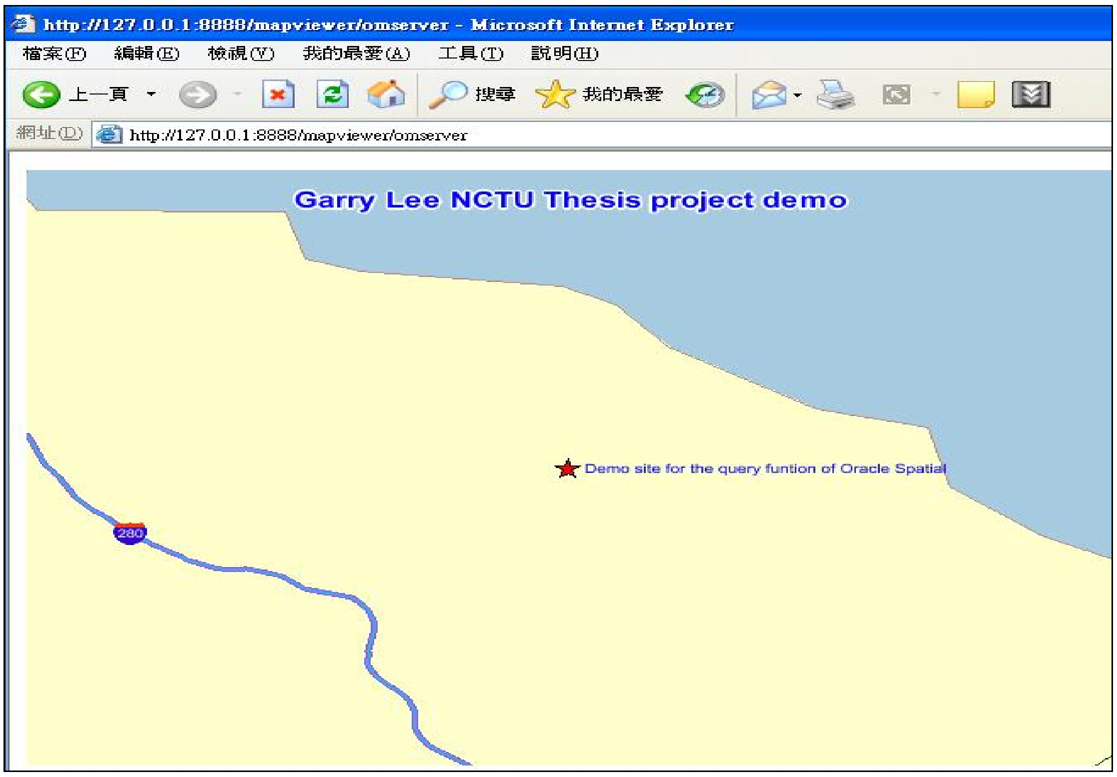
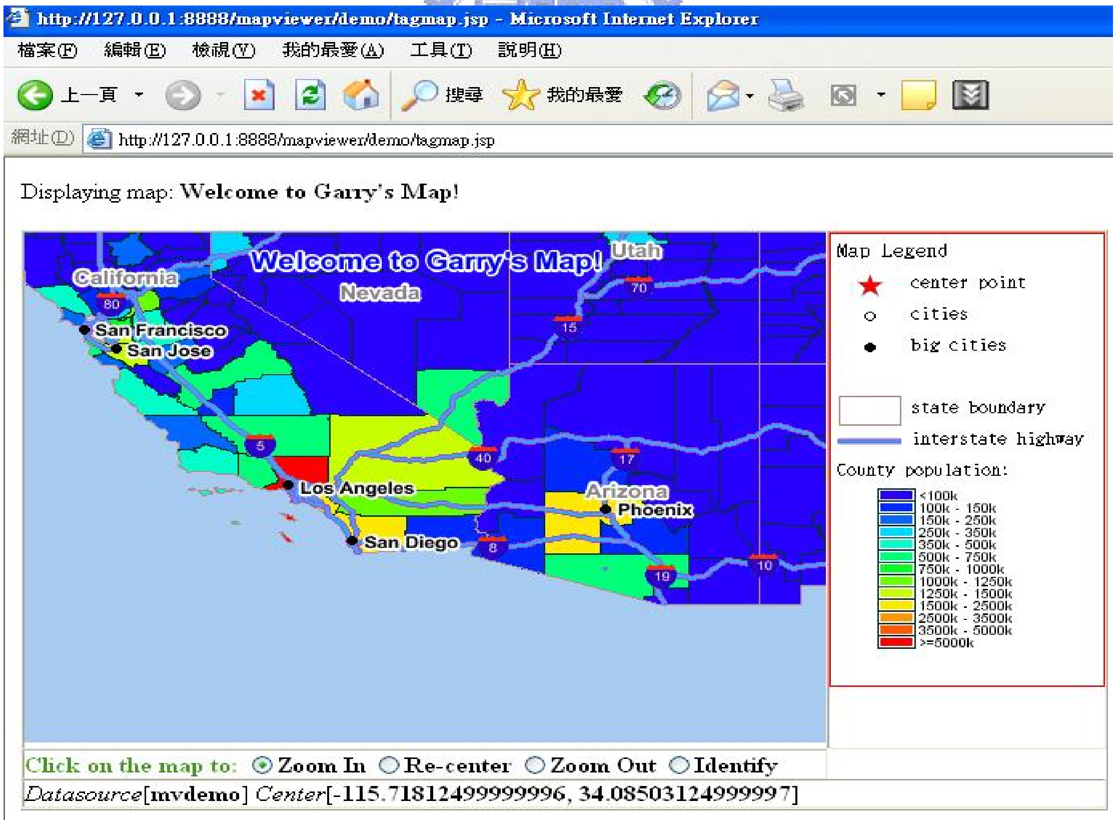

http://127.0.0.1:8888/mapviewer/omserver - Microsoft Internet Explorer
檔案(F) 編輯(E) 檢視(V) 我的最愛(A) 工具(T) 說明(H)
上一頁
搜尋
我的最愛
網址(D) http://127.0.0.1:8888/mapviewer/omserver
Garry Lee NCTU Thesis project demo
Demo site for the query funtion of Oracle Spatial
280

Figure 14. My project demo



Displaying map: **Welcome to Garry's Map!**

Datasource[mvdemo] *Center*[-115.71812499999996, 34.08503124999997]

Figure 15. MapViewer with spatial data

Next we shall move on the approach of the thesis. Every time when we load data we will run simulations on the server. The CPU time involved in various methods is as follows. Depending on the data dimensionality, it took about 1 to 3 minutes of CPU time to construct the leaf level of the R-tree for 5000 data objects for whole table scanning. The current implementation of the proposed iterative method required from 2 minutes to 5 minutes of CPU time, depending on the improvement achieved during iteration. If we construct the data by original data and original method in Oracle10g Spatial database, and create the index without specifying any different data structure, we will prompt out the first indicator as “Experimental Group”, using SQL to query a specific location with county, city, states, interstates and territories. On the other hand, I create the metadata for the tables by using definitions from Oracle10g Spatial database, and divide into several regions as States, Cities…etc. in (USER_SDO_GEOM_METADATA) and create the details with a view by leveraging previous objects.

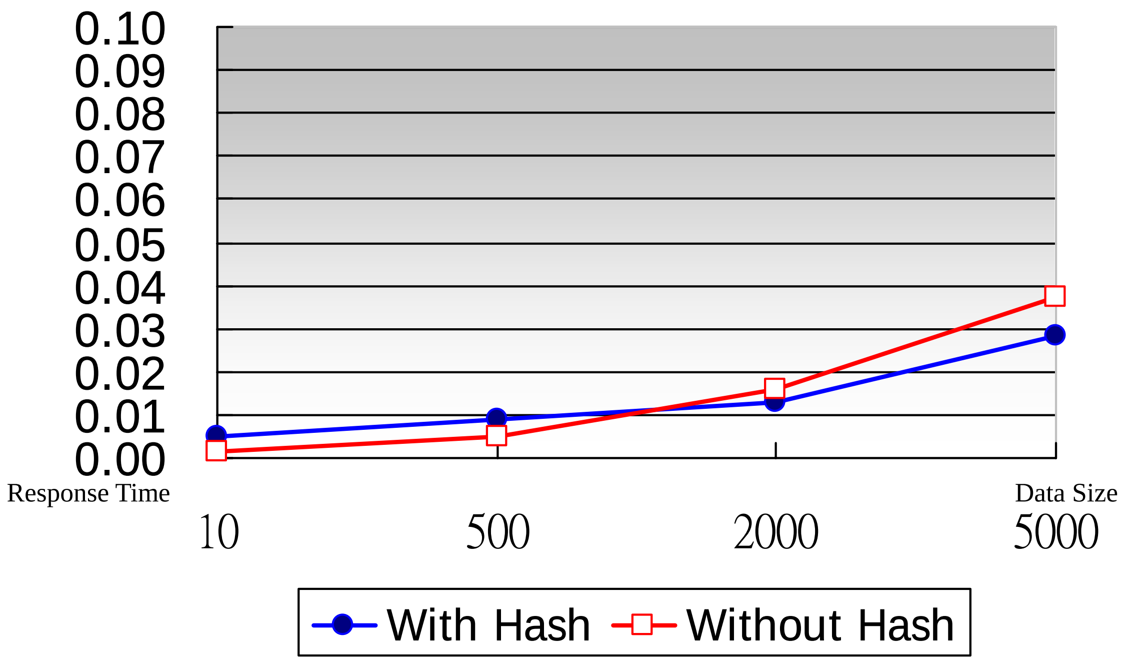
I use DECLARE store procedure and spatial object types to create the hash table and views for operation. Basically Oracle has provided the hash functions but I need to rewrite new PL/SQL store procedure hash_table so that we can use the formula from ANSI SQL-92 e.g. log2( ), floor( ), mod( ) into operations to send to correct address of hash table by bulk-loading (Enter the data in subset of one operation and it will run as recursive into the data structure. The store procedure is attached in appendix.

5.2 Indexing and Querying

After construct the table and we can create the index without specifying any odd parameters. All data entered in the table will take over two minutes to do fully table scan because it has several table joined together. Once we have done everything, we set a target to query in the tables and views. Testing group shows in Red line that go with regular R-Tree structure. The control group is using our approach with hashed data structure. We set the query and get the run time back with the size of table (10, 500, 2000, 5000):

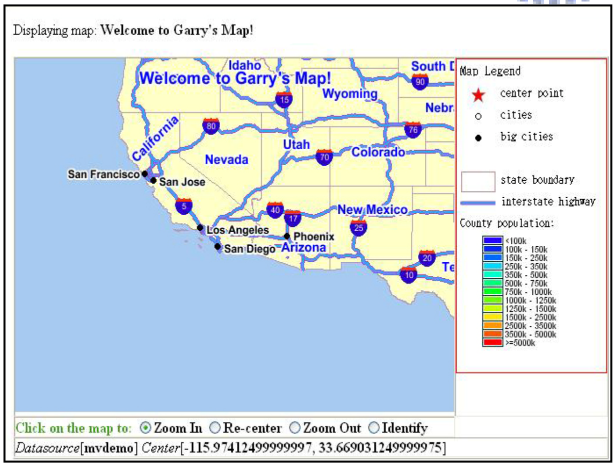
Select SDO_gcdr.geocode('mvdemo', SDO_keywordArray('100 VAN NESS AVE', 'San Francisco, CA'), 'US', 'DEFAULT') from dual;

Apparently when I manually insert the data into hash-functioned table, it takes more time to buildup the SDO object and may waste the spaces so it takes extra 67% after querying out one specific object. Following on the incremental of data entered into tables, we can see it improved and more close and superior than default R-tree type on spatial database. This shows 24% enhancement in 5000 records in the hashed R-tree table.

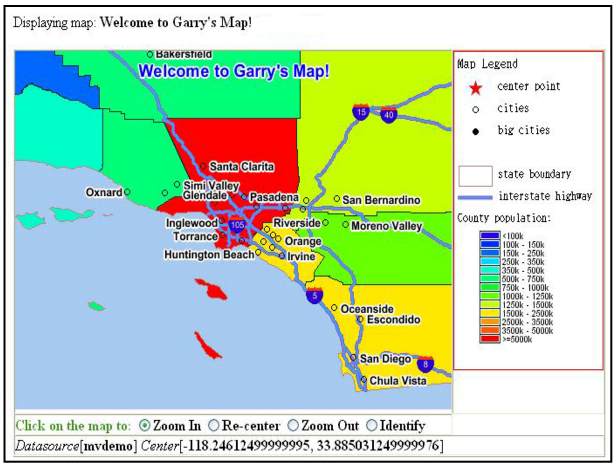


	10	500	2000	5000
With Hash	0.00504	0.00874	0.0129	0.0282
Without Hash	0.00162	0.00481	0.0157	0.0374

Figure 16. Result – Hashed R-Tree v.s. non-hashed R-Tree


Displaying map: Welcome to Garry's Map!

Datasource[mvdemo] Center[-115.97412499999997, 33.669031249999975]


Displaying map: Welcome to Garry's Map!

Datasource[mvdemo] Center[-118.24612499999995, 33.885031249999976]

Figure 17. Project demo – final stage on US sample data.

Chapter 6 Conclusion

In this thesis, we propose an efficient method to implement a new enhanced algorithm in spatial databases. This approach is not only based on an original architecture design implementation in the database but also improves the query time by avoiding the tree recreation and by preventing the skew tree. By our performance analysis, hashing before R-Tree creation on the database could improve the quality of the index thereby adversely affecting subsequent query performance, since in R-Tree splitting balanced tree is guaranteed during database insertion, deletion and most of operations in spatial database. This strategy also improves the quality of the constructed R-tree index by reorganizing overlapping subtrees. The experiments using real datasets show an improvement in insertion performance by 20%-70% using my approach in comparison to original data index creation for querying. Query performance also improves. This paper explored the idea of identifying and storing objects by hashing to tune the performance. The merit of the technique is that it can be applied for database engines at the same time. If the query plan operation is realized after hash operations, it will cost less operation on constructing the R-tree structure.

References

[1] A. Guttman: “R-trees: A dynamic index structure for spatial searching”, Proc. ACM SIG-MOD, pp. 47 – 57, 1984.

[2] N. Beckman and H.P. Kriegel: “The R* tree: An efficient and robust access method for points and rectangles", Proc. ACM SIGMOD, pp. 322 –s 331, 1990.

[3] Timos Sellis, Nick Roussopoulos and Christos Faloutsos: “The R+-Tree: A Dynamic Index for multi-dimensional objects”, 13th VLDB Conference, 1987

[4] Lepouchard, Y.; Orlandic, R.; Pfaltz, J.L.: “Performance of KDB-trees with query-based splitting”, Proceedings of International Conference on Information Technology: Coding and Computing, pp. 218 – 222, April 2002

[5] Ravi Kanth V Kothuri, Siva Ravada, and Daniel Abugov: “Quadtree and r-tree indexes in oracle spatial: a comparison using gis data.” Proceedings of ACM SIGMOD Conference, 2002.

[6] S. Saltenis, C.S. Jensen, “Indexing of Now-Relative Spatio-Bitemporal Data”, The VLDB Journal, pp.1-16, 2002

[7] Saltenis, C. Jensen, S. Leutenegger, and M. Lopez. “Indexing the position of continuously moving object Trajectories.” Proceedings of ACM SIGMOD Conference, 2000.

[8] Y. Tao and D. Papadias. Mv3r-tree: a spatio-temporal access method for timestamp and interval queries. Proc. of 27th Int’l Conf. on Very Large Data Bases, 2001.

[9] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In SIGMOD ’00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data, pp. 331–342, New York, USA, 2000.

[10] Yuni Xia, Sunil Prabhakar: “Q+Rtree- efficient indexing for moving object databases”, Proceedings of the Eighth International Conference on Database Systems for Advanced Applications (DASFAA’03) 2003

[11] Oracle Corporation, Oracle10g Spatial Release 2, B14255-01, “Spatial User's Guide and Reference”, 2005 http://www.oracle.com/technology/documentation/spatial.html

[12] Oracle Corporation, Oracle10g Spatial Release 2, B14256-01 “Oracle Spatial Topology and Network Data Models”, 2005 http://download.oracle.com/docs/html/B14256_01/toc.htm

[13] Oracle Corporation, Oracle10g Spatial Release 2, B14254-01 “Oracle Spatial GeoRaster”, 2005 http://download.oracle.com/docs/html/B14254_01/toc.htm

[14] Oracle Corporation, Oracle10g Spatial Release 2, B14373-01” Oracle Spatial Java API Reference”, 2005 http://download.oracle.com/docs/html/B14373_01/toc.htm

[15] Ning An, Ravi Kanth V Kothuri, Siva Ravada, “Improving Performance with Bulk-inserts in Oracle R-Trees”, Proceedings of the 29th VLDB Conference, 2003

[16] Jeremy Kubica, Joseph Masiero, Andrew Moore, Robert Jedicke, Andrew Connolly, “Variable KD-Tree Algorithms for Efficient Spatial Pattern Search”, Proceedings of Neural Information

Processing Systems - NIPS05, pp. 1-3, 2005, Whistler, CANADA

[17] Egemen Tanin, Aaron Harwood, Hanna Samet:"Indexing distributed complex data for complex queries", Proceedings of the National Conference on Digital Government Research - DGO, pp. 81-90, 2004, Seattle, WA. US

[18] Ricardo Ciferri, Ana Salgado, Mario A. Nascimento, Geovane Magalhaes: "A performance comparison among the traditional R-trees, the Hilbert R-tree and the SR-tree", Proceedings of the XXIII International Conference of the Chilean Computer Science Society, 2003

[19] Kamel, I., and Faloutsos, C. "Hilbert R-tree: An Improved R-tree using Fractals". In Proc. VLDB Conference, pp. 500-509, 1994.

[20] Katayama, N., and Satoh, S. "The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries. In Proc. SIGMOD Conference, pp. 369-380, 1997.

[21] Jukka Teuhola, "Effective Clustering of Objects Stored by Linear Hashing", Conference on Information and Knowledge Management – CIKM '95, Baltimore MD USA, ACM 1995

[22] David Sonnen, Henry D. Morris, Dan Vesset, "Worldwide Spatial Information management 2005-2009 Forecast and 2004 Vendor Shares, International Data Centre (IDC) Market Analysis, Doc# 34321, Nov 2005.

[23] Andrei Radu Popescu, "A Study of R-tree Based Spatial Access Methods", UNIVERSITY OF HELSINKI, pp. 14-17, 2004