A COMPARISON OF NON-POINT SPATIAL DATA INDEXING METHODOLOGIES

by

Liping Xie

B.Sc.E. Jianghan Petroleum Institute, China, 1990

A Thesis Submitted in Partial Fulfillment of
the Requirements for the Degree of

Master of Computer Science
in the Faculty of Computer Science

THE UNIVERSITY OF NEW BRUNSWICK

November 1999

ABSTRACT

The general objective of this research is to compare the performance of several spatial indexing methods. In order to do so, we built spatial indices for two-dimensional non-zero size objects and performed range searches on those objects. There are two ways to create an index: 1) use the spatial data support in commercially available databases such as Oracle 8i and O2, and 2) create the spatial data-indexing scheme directly using spatial data structures such as the R-tree or Grid file. A new data structure based on the binary-radix bucket-region (BR^2) grid file was created for non-point objects.

Two sets of data from the National Topological Database were used to perform the experimental comparisons. We performed two different types of range search: search based on the bounding box of the objects only (primary) and search based on the real data (secondary). Testing was performed with query window areas = 1%, 4%, 9%, 16%, 25%, 36%, 49%, 64%, 81%, and 100% of the data extent.

For secondary range search, the R-tree and the modified grid file took the least amount of time (0.497 seconds and 0.628 seconds respectively) on average to report (out of a test set of 20,000 objects) the objects in range for a query window area = 49% of the original data extent. The Oracle 8i Spatial object-relational, and the Oracle 8i Spatial relational methods required, on average, 220, and 400 times more time for the same search, respectively. Significant difficulties were encountered in getting the commercially available databases to perform range searches on the test data.

ACKNOWLEDGEMENTS

I would like to express sincere thanks to my supervisor, Dr. Bradford G. Nickerson, for his suggestion of this subject, consistent guidance through each phase of this research, and encouragement concerning my academic and nonacademic endeavors. His valuable advice and insightful thoughts have greatly contributed in shaping this research work. I am very grateful for all of these.

I would also like to thank Dr. Collin Ware, and Dr. Alejandro Lopez-Drtiz, who read the thesis drafts and provided valuable comments and suggestions.

Thanks are extended to the many individuals who contributed to this thesis, including Mr. Kirby Ward, Mr. Matthew Travors, Mr. Jeff Arsenault, Mr. Scott Steven, and Mr. Mark Cooper for their assistance in setting up and solving my problems with Oracle and O_2 database management system. I would especially like to thank Mr. Nhu Le for his valuable advice in debugging the programs and consistent encouragement.

Finally, a great appreciation should also be given to the Faculty of Computer Science whose continual funding made this work possible. The financial support of the Natural Sciences and Engineering Research Council is also gratefully acknowledged.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER ONE

INTRODUCTION

1.1 Spatial Data and Range Queries

1.1.1 Spatial Data

A common example of spatial data can be seen in a road map. A road map is a two-dimensional object that contains points, lines, and polygons that can represent cities, roads, and political boundaries such as states or provinces. A road map is a visualization of geographic information. The location of cities, roads, and political boundaries that exist on the surface of the Earth are projected onto a two-dimensional display or piece of paper, preserving the relative positions and relative distances of the rendered objects. The data that indicates the Earth location (latitude, longitude, and height or depth) of these rendered objects is the spatial data.

Spatial data is rich in meaning including not only locational but also topological information stored either explicitly or implicitly. From the data we can acquire the locations and shapes of the objects, as well as their relation with other objects.

There are two basic types of spatial data: raster and vector. For vector data, there are three primitives: a point, a polyline, and a polygon. Points can be considered as a basic type, since any complicated object can be ultimately decomposed into points. Spatial operations for points can be used for the objects of the other two types. In this thesis, we will deal with non-point data such as polylines and polygons.

1.1.2 Range Search

The range Search based on a range query is a frequent operation performed on spatial objects. It can be described as retrieving or counting a collection of spatial data. Given a set of spatial data and a query window, there are two types of range queries. First, tell if a specified object intersects the query window. Second, report all objects that intersect the query window. For example, a typical one-dimensional range search requires retrieval of all the objects within the range $[x_1, x_2]$ among N objects ordered along the x-axis. In this thesis we will be primarily concerned with a two-dimensional range query with a 2-d orthogonal query window.

Since the large volume spatial data must be physically stored on a medium such as a disc, overhead is always a problem in range search. Reading data from the medium or writing data to it involves relative movement of the disc heads. Two locationally close data objects may not be close on the disk, and a range search operation may require substantial disc head movement. This results in a slow range search time.
In order to solve this problem, a spatial indexing mechanism is used. The spatial indexing structure of the real objects is much smaller than the set of real objects themselves, so it can be kept in memory to speed up the range search processing.

1.2 Data Structures for Spatial Indexing

In order to speed up range searches in geographic information systems and digital mapping, advanced indexing mechanisms for spatial data are used. There are many spatial indexing methods developed starting in the 1970s.

Three indexing schemes were investigated by Greene [1989] from the University of California at Berkeley. Those schemes included the R- tree, KDB-tree, and 2D- ISAM.

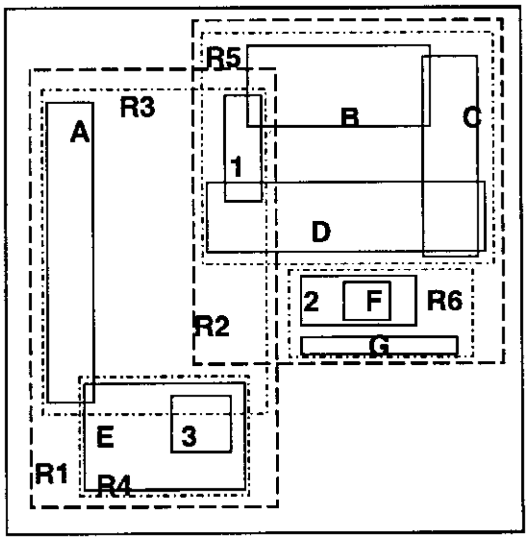
R-tree indexing [Guttman, 1984] is based on a multi-level tree structure designed to handle n-dimensional objects. All non-leaf nodes are assigned a minimum and maximum number of entries and each entry contains an identifier which locates its child node and also the spatial region which the child covers (see Figure 1.1). The time for range search in an R-tree is O(N).

Similarly, the KDB-tree [Robinson, 1981] is another multi-level balanced tree structure designed to handle n-dimensional point data. The nodes at a given level in the tree are partitioned into disjoint sub-regions such that when summed, span the entire search space.

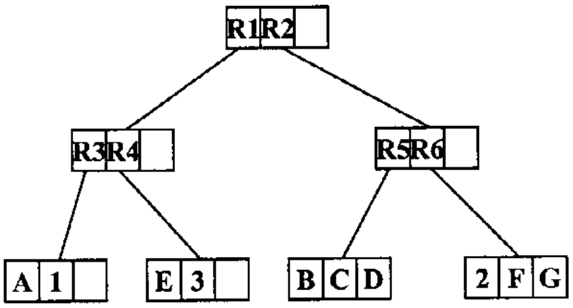
On the other hand, 2D-ISAM [Greene, 1989] is a two-level tree structure similar to grid files; the first level partitions the data along the x-axis and the second level partitions along the y-axis. Compared to the R-tree, 2D-ISAM is better in terms of storage required (approximately 1/3 less space for 2D-ISAM). This is the only advantage of 2D-ISAM over the R-tree. R-trees can be made to simulate 2D-ISAM performance-wise, with a significant advantage: the data in the R-tree has a dynamic index and can accept updates without risking overflow and requiring re-organization.

Empirical testing on insertion and retrieval operations indicated that R-tree indexing provided the best overall performance [Greene, 1989].

Gao [1994] conducted an experiment to compare internal indexing and external indexing [Nickerson and Gao, 1998]. In his research, a Morton code indexing scheme using the INGRES database was compared to a Morton code external indexing scheme written directly in C code. The results were clear; external Morton code indexing provided excellent performance compared to Morton code indexing using the INGRES database; direct Morton code indexing took an average of 1,535 times less CPU time for searching, and required approximately 4.5 times less space. Again, this is also an empirical result based on real data obtained from the Hydrographic Data Cleaning System (HDCS) [Ware et al., 1990].



(a) A planar representation



(b) The directory of a R-tree

Figure 1.1 The structure of an R-tree where M=3 (adapted from Samet [1990]).

The R-tree indexing structure has many advantages. Firstly, minimal bounding rectangles (MBRs) give us a rough idea about the Spatial extent of objects. MBRs are easy to describe and to be compared with search ranges since their edges are parallel to the x- and y-axes, respectively, and an MBR can be simply defined by four numbers. Secondly, when massive data are involved, a hierarchy can be formed by using large MBRs covering small MBRs. Thirdly, leaf nodes store the descriptions of complete spatial objects. Thus the spatial search can be performed in an object-oriented way. Range search can be performed at fast speed because of the R-tree's hierarchical characteristic as well as its using MBRs in an object-oriented manner. Fourthly, R-trees can be used in dynamic indexing. For further details of the R-tree structure, please see chapter 3.

Abel and Smith [1983] solved the rectangle cover problem using a B^+ tree. This consists of a sorted list of rectangles held in ascending sequence by key value, with each member of that list consisting of the rectangle identifier and the assigned locational key. The B-tree index provides direct access to the member of the sorted list with the smallest locational key value greater than or equal to a given key. The number of accesses to locate a number through the B tree is O(logN).

Nievergelt's grid file [1984] is a metric data structure designed to store points and simple geometric objects in multidimensional space. Its goal is to retrieve records with at most two disk accesses and to handle range queries efficiently (see Figure 1.2). Smith and Gao [1990] evaluated the performance of grid-file spatial access method. The grid file, like the B^+ tree, performs very well in the insert and delete operation and achieves high storage utilization. They report that the grid file has poor search performance, but no details are given. This seems to contradict Nievergelt et al's [1984] paper that claims a worst case of two disk accesses to retrieve a record.

Hinrichs [1990] introduced the new BR^2 –directory structure for the grid file. BR^2 means binary radix bucket region. It is a tree structure, designed to handle multidimensional point data. Two disk accesses for search is not guaranteed; however, this new kind of directory guarantees a linear growth with the number of buckets. Figure 1.2 shows the construction of BR^2-directory for the two-dimensional partitioning of the data space by a tree-like structure. Data buckets are indicated by the label B_1, B_2, B_3, …, B_7.

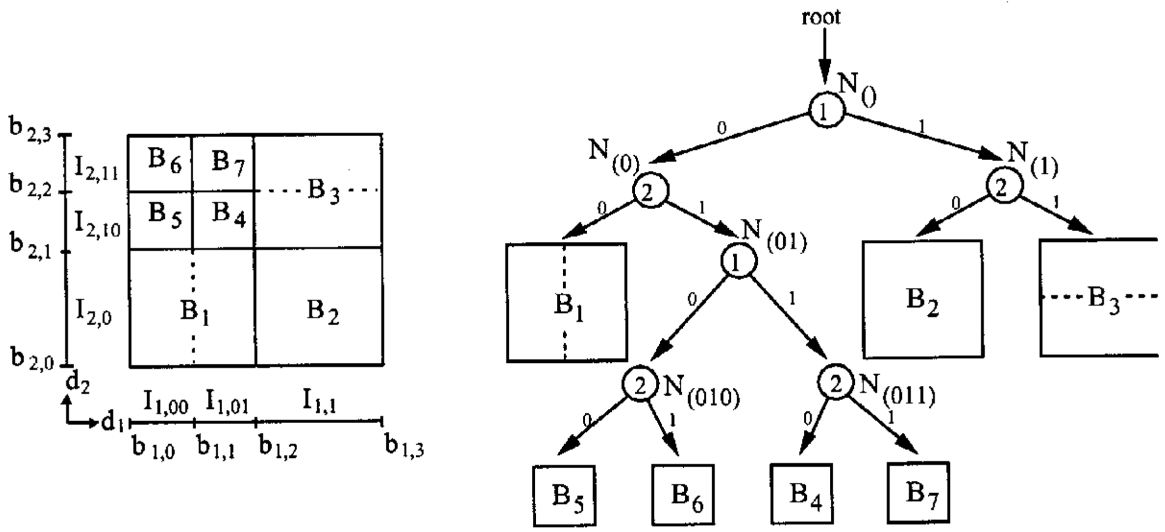


Figure 1.2 The grid file structure: **BR^2**-directory and **BR^2**-tree (adapted from Hinrichs [1990]).

The BR^2-directory has the advantage that every bucket in the query range is returned exactly once. In the grid array representation [Hinrichs, 1985] a bucket may cover more than one grid cell and hence it may be returned more than once during range searches. Therefore we have to decide for each bucket whether it has already been visited. For this test we must construct its bucket region, and determine whether this region extends into a part of the data space that has already been processed. Constructing the bucket region from the grid array representation is a time-consuming task.

The BR^2-directory is superior to the region directory [Hinterberger, 1987] for the following reasons. Simpler and fewer comparisons are necessary to find the relevant buckets. The space to represent bucket regions in the BR^2-directory does not depend on the number of dimensions of the grid file. The overall space requirements are higher for the region directory. For further details of grid file structure, please see chapter 4.

1.3 Problem Definition

It is expected that the application characterizes the degree of suitability of an indexing mechanism. An effective data structure on one kind of spatial data may behave differently on other kinds of spatial data.

The general objective of this thesis is to build efficient spatial indices for two-dimensional objects and perform range searches upon those objects. There are two ways to create spatial indexing: 1) use the spatial data support available with some databases, and 2) create the spatial data-indexing scheme by ourselves. The following are the specific objectives:

- Can the grid-file method [Hinrichs, 1991] be extended to handle non-point data?
- How well do commercial database spatial indexing tools handle non-point data?
- How do commercial database spatial indexing tools compare in terms of time for the range search to direct spatial indexing methods?

For commercial database spatial indexing methods, we choose to use the Oracle 8i^{TM} spatial data cartridge [Oracle 8i, 1999] and O_2 spatial [Ardent, 1998]. For direct spatial indexing methods, we implemented the R-tree and the grid-file.

CHAPTER TWO

NTDB TEST DATA

2.1 Data Range

The test data we used was obtained from the National Topographic Database (NTDB). The contents of the NTDB largely correspond to that of the topographic maps in the National Topographic System (NTS). Under the NTS, Canada is divided into numbered primary quadrangles, each 4° latitude by 8° longitude south of 80°, and 4° latitude by 16° longitude north of 80°. In this research we will use two sets of data. The first covers the area bounded by longitude 66° to 68° W and latitude 45° to 46° N, and represents the 1:250,000 scale map sheet NTS21G. This data set has approximately 9 MB of geographical data. It contains 20,108 objects (15,370 polylines and 4,738 polygons), and the data extent is 114,994m in the NS direction, and 159,018m in the EW direction. The average bounding box of a polyline object is 1051m in the NS direction, and 986m in the EW direction. The average length of a polyline is 1906m. The average bounding box of a polygon object is 671m in the NS direction, and 599m in the EW direction. The average area of a polygon is 69,825m^2. The second set of data covers the area bounded by longitude 66°30′ to 67° W and latitude 45°45′ to 46°N, and represents the 1:50,000 scale map sheet NTS21G15; it contains about 14MB of geographical data. It contains 17,034 objects (13,198 polylines and 3,836 polygons), the data extent is 28,870m in the NS direction, and 40,106m in the EW direction. The average bounding box of a polyline object is 410m in the NS direction and 408m in the EW direction. The average length of a polyline is 713m. The average bounding box of a polygon object is 172m in the NS direction, and 159m in the EW direction. The average area of a polygon is 17,037m^2. Plots of real objects as well as histograms of sizes of objects for these two datasets are shown in Appendix 0. Digital data sets for all of Canada are available, and can be ordered using http://maps.nrcan.gc.ca [NRCanada, 1999].

2.2 Characteristics of the Data

To represent a topographic feature in the NTDB, three distinct data components are needed, namely: entity, geometric representation, and ancillary data. For NTDB purposes, an entity is the digital representation of the descriptive component of a topographic feature. It is associated with a name in order to distinguish it from other entities (e.g. road, dam, transmission line, and contour).

The geometric representation is the digital representation of the spatial component of a topographic feature. The NTDB supports three different types of geometric representation: point, line and area. Each entity may be associated with more than one type of geometric representation. Each type of representation (point, line, and area) may be defined in two or three dimensions. Three-dimensional definition is given by association of an elevation (Z) with each point or vertex, both defined by a pair of planimetric coordinates (X, Y). Coordinates are represented in the Universal Transverse Mercator (UTM) projection on the North American Datum 1983 (NAD83) to a resolution of one meter. All points, lines, and areas are assigned an identifying feature code. Information inherent to data acquisition and validation is assembled in the ancillary data of entities. A maximum of two distinct occurrences of ancillary data of entities may be simultaneously associated with an occurrence of geometric representation: one specific to data acquisition and one specific to data validation.

2.3 Data Format

The digital data is in Intergraph Standard Interchange Format (ISIF). ISIF is a product of Intergraph Systems Ltd. For a full description of ISIF, see EMR Canada [1983, 1987,1989]. Only a sub-set of ISIF commands are used here. They are as follows:

LST- Line String, used to define a line object consisting of n coordinate pairs.

e.g. LST/x1,y1,z1,...,xn,yn,zn

    LST/OP,x1,y1,z1,...,xn,yn,zn

CUR- Curve, followed by n coordinate pairs.

e.g. CUR/x1,y1,z1,...,xn,yn,zn

CON- Continuation data, followed by n coordinate pairs.

e.g. CON/x1,y1,z1,...,xn,yn,zn

ASC- Association, used to define the feature code assigned to an object.

e.g. ASC/Feature code (e.g. 5790)

DID- Drawing identification, used to define the name of the file that was created.

e.g. DID/NA = file name, DA = date, MO = dimension, RA= Range

Note that all the coordinates are given in meters using the UTM map projection [e.g. Krakiwsky, 1973]. Figures 2.1 and 2.2 show sample data from the two test datasets.

```
DID/NA=21GEMR2DZ,DA=02/28/90,MO=2,RA=577435,4983220,736453,5098214
ASC/5790
LST/OP,629829,4998263,629816,4998287,629766,4998331,629722,4998330,
    629722,4998305,629792,4998256,629829,4998263
LST/OP,629769,4998486,629662,4998529,629581,4998540,629537,4998565,
    629487,4998571,629468,4998539,629487,4998489,629525,4998452,629613,
    4998403,629695,4998385,629745,4998411,629769,4998486
LST/OP,627536,5000496,627529,5000546,627460,5000614,627416,5000645,
    627353,5000657,627334,5000650,627322,5000625,627354,5000550,627436,
    5000470,627487,5000458,627524,5000471,627536,5000496
LST/OP,605425,4999195,605418,4999233,605393,4999251,605356,4999245,
    605312,4999188,605319,4999144,605345,4999113,605358,4999063,605383,
    4999045,605414,4999052,605419,4999189,605425,4999195
```

Figure 2.1 Sample ISIF data for 1:250,000 map sheet NTS21G.

```
DID/NA=21G15T3D,DA=07/02/87,MO=3,RA=654868,5067998,-2147483648,694974,
5096868,2147483647
ASC/9963
LST/OP,655564,5067998,-32769,675009,5068515,-32769,694454,5069093,-
32769,693584, 5096868,-32769,674226,5096290,-32769,654868,5095774,-
32769,655564,5067998, -32769
LST/OP,654868,5095774,-32769,655056,5088263,-32769,655245,5080753,-
32769
ASC/10100
CUR/665328,5082570,100,665337,5082559,100,665344,5082545,100,665345,508
2536,100,65346,5082502,100,665346,5082469,100,665350,5082446,100,665362
5082433,100,665373,5082413,100,665381,5082383,100,665391,5082361,100,66
5404,5082348,100,665417,5082346,100,665437,5082361,100,665467,5082401,1
00
```

Figure 2.2 Sample ISIF data for 1:50,000 map sheet NTS21G15.

2.4 Data Preprocessing

Since we use only these two sets of data to compare the performance of several spatial indexing methods, we do not have to distinguish different layers. This means we do not keep track of the object type (feature code). Instead we only take coordinates for each object (point, polyline, and polygon) and put all the objects in one layer for each set of data. For the 1:50,000 dataset, we discard the z value. All the objects in the two sets of data are considered as two-dimensional. For the 1:250,000 dataset, we only read in the coordinates after LST/OP. For the 1:50,000 set of data, we read in the coordinates after LST/OP, CUR and CON.

Since the original data is in ASCII format, we first pre-process the data to transform the original data into a format that represents the same information using a smaller amount of space. This will also allow faster retrieval of information. We split the original data into two files: the control file and the detail file. The detail file acts as a database that contains detailed information and the control file acts as the index for the database. For each record in the detail file, there is a corresponding record in the control file that contains meta-data of the record such as the position of the record, the length of the record, and the bounding box of the record. Notice that although the bounding box can be derived from the data, we decided to store the bounding box in the control file for faster searching. This means that we only need the control file to do a primary search (i.e. a search on bounding rectangles only). Once we have narrowed down the number of records that could potentially match our search criteria, we can then access the detail file to do a more accurate, slower search. In both the R-tree and grid file methods, we used this preprocessed form of data. The control file is used to build the tree itself and the detail file can be stored anywhere on disk and accessed as needed. Figure 2.3 illustrates how original data files are preprocessed. Figure 2.4 shows the format of the control file.

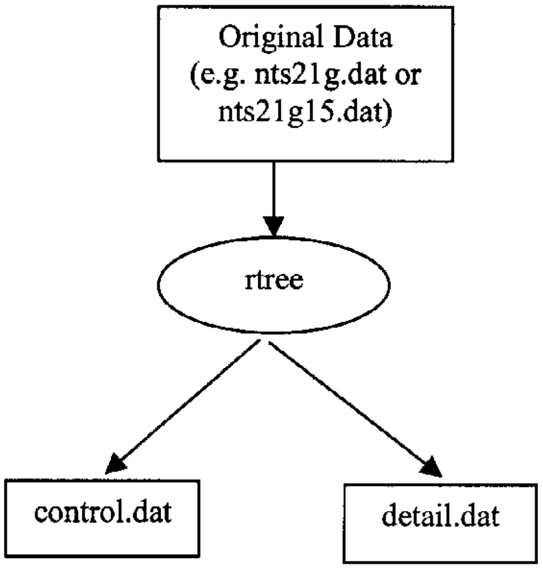


Figure 2.3 Test data preprocessing.

CoverAll	Position1	Length1	BoundingBox1	…	PositionN	LengthN	BoundingBoxN
*(4 * int)*	(long int)	(int)	(4 * int)		(long int)	(int)	(4 * int)

Figure 2.4 Format of the control file.

Coverall means the data extent, and includes 4 integers giving the minimum value of X and Y and the maximum value of X and Y for all objects in the file. The rest of the file contains position, length and bounding box for each object in turn.

Position is the byte offset of the beginning of the coordinates for this object in the detail file. Length is the number of vertices stored for this object in the detail file.

Fig. 2.5 shows the structure of the detail file.

X1 Array	Y1 Array	...	...	Xn Array	Yn Array

Figure 2.5 Format of detail file.

X1 Array and Y1 Array contain length1 X and Y coordinates for the first object, and Xn Array, Yn Array contain lengthn X and Y coordinates for the nth object.

CHAPTER THREE

R-TREE SPATIAL INDEXING

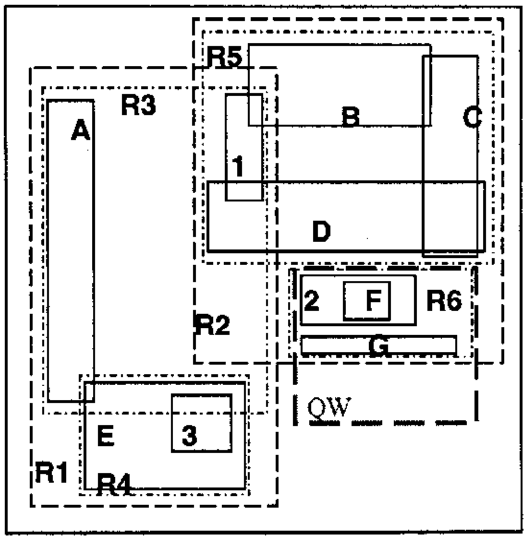
3.1 The Structure of an R-tree

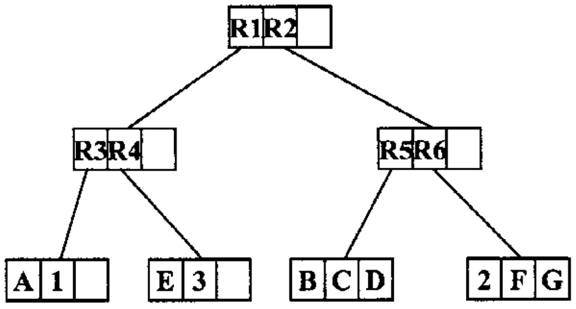
In order to handle spatial data efficiently, we choose to use the R-tree, a dynamic index structure, designed to handle multi-dimensional spatial data.

An R-tree is a height-balanced tree similar to a B-tree [Comer, 79]. Each node in the tree corresponds to the smallest d-dimensional rectangle that encloses its child nodes. The leaf nodes contain pointers to the actual geometric objects in the data file, instead of children. The objects are represented by the smallest aligned rectangle in which they are contained. Nodes correspond to disk pages if the index is disk-resident, and the structure is designed so that pruning of nodes during spatial search takes place as close to the root as possible. The index is completely dynamic; inserts can be intermixed with searches and no periodic reorganization is required.

The basic rules for the formation of an R-tree are similar to those for a B-tree. All leaf nodes appear at the same level. Each entry in a leaf node is a 2-tuple of the form (R,O) such that R is the smallest rectangle that spatially contains data object O. Each entry in a non-leaf node is a 2-tuple of the form (R, P) such that R is the smallest rectangle that spatially contains the rectangles in the child node pointed at by P. As for a B-tree, an R-tree of order m means that each node in the tree, with the exception of the root, contains between ⌈m/2⌉ and m children. The root node has at least two children unless it is a leaf node.

For example, consider the collection of rectangles given in Figure 3.1, and treat the query rectangles (i.e., 1, 2, and 3) as elements of the collection so that there are 10 rectangles. Let m=3. One possible R-tree for this collection is given in Figure 3.1(b). Figure3.1(a) shows the spatial extent of the rectangles of the nodes in Figure3.1(a), with broken lines denoting the rectangles corresponding to the sub-trees rooted at the non-leaf nodes. Note that the R-tree is not unique. Its structure depends heavily on the order in which the individual rectangles were inserted into (and possibly deleted from) the tree.





(a) A planar representation

(b) The directory of an R-tree

Figure 3.1 The structure of an R-tree where m=3 (adapted from Samet [1990]).

3.2 R-Tree Operations

3.2.1 Searching

The search algorithm descends the tree from the root in a manner similar to a B-tree. More than one sub-tree under a node visited may need to be searched. It is not possible to guarantee good worst-case performance. With most kinds of data the update algorithms will maintain the tree in a form that allows the search algorithm to eliminate irrelevant regions of the indexed space, and examine only data near the search area.

In the following we denote the rectangle part entry E by E.i and the child-pointer part by E.p. Given an R-tree whose root node is T, find all index records whose rectangles overlap a search rectangle S. If T is not a leaf, check each entry E to determine whether E.i overlaps S. For all overlapping entries, search on the tree whose root node is pointed to by E.p. If T is a leaf, check all entries E to determine whether E.i overlaps S. If so, E is a qualifying record. Checking bounding rectangles only against S is called a primary search.

3.2.2 Insertion

The algorithm for inserting an object (i.e., a record corresponding to its enclosing rectangle) in an R-tree is analogous to that used for B-trees. New rectangles are added to leaf nodes. The appropriate leaf node is determined by traversing the R-tree starting at its root and at each step choosing the sub-tree whose corresponding rectangle would have to be enlarged the least. Once the leaf node has been determined, a check is made to see if insertion of the rectangle will cause the node to overflow. If yes, then the node must be split, and the m+1 records must be distributed in the two nodes. Splits are propagated up the tree. Details of algorithms for splitting nodes are discussed in Guttman [1984].

3.3 Implementation

We stored object detail data on disk, stored bounding box and disk location of each object in leaf nodes of the R-tree, and the entire R-tree indexing structure is kept in memory.

In building a tree, we use index information available in the control file (control.dat) to populate the nodes of the tree. This allows the control information to be searched much faster. Once all the information is in the tree, we can do an in-memory primary search (see Figure 3.2). Once we have narrowed the search to a smaller set of possibilities, we can then use the data in the detail file (detail.dat) to perform a secondary search to check for objects intersecting the search rectangle S. This is illustrated in Figure 3.2 (b). The formats of control.dat and detail.dat files are described in Chapter Two.

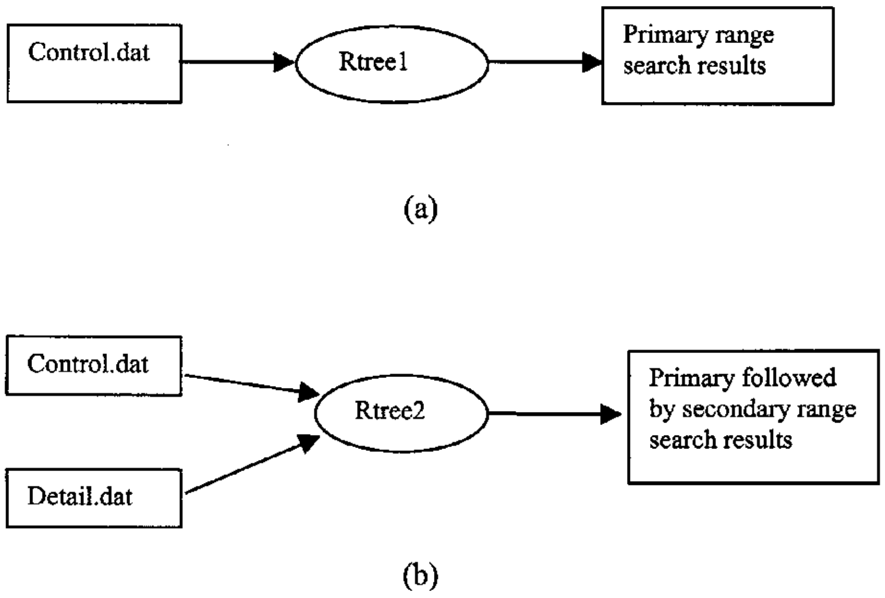

Control.dat
Rtree1
Primary range search results
(a)
Control.dat
Detail.dat
Rtree2
Primary followed by secondary range search results
(b)

Figure 3.2 Implementation architecture for R-tree testing.

3.4 Testing

We performed two different types of range search to test the performance of the R-tree indexing methods. The first was a search based on the bounding box of the objects (primary search) and the second was a search based on the real objects (secondary search). Figure3.3 illustrates an example range search on six objects. The query window is shown in dashed lines.

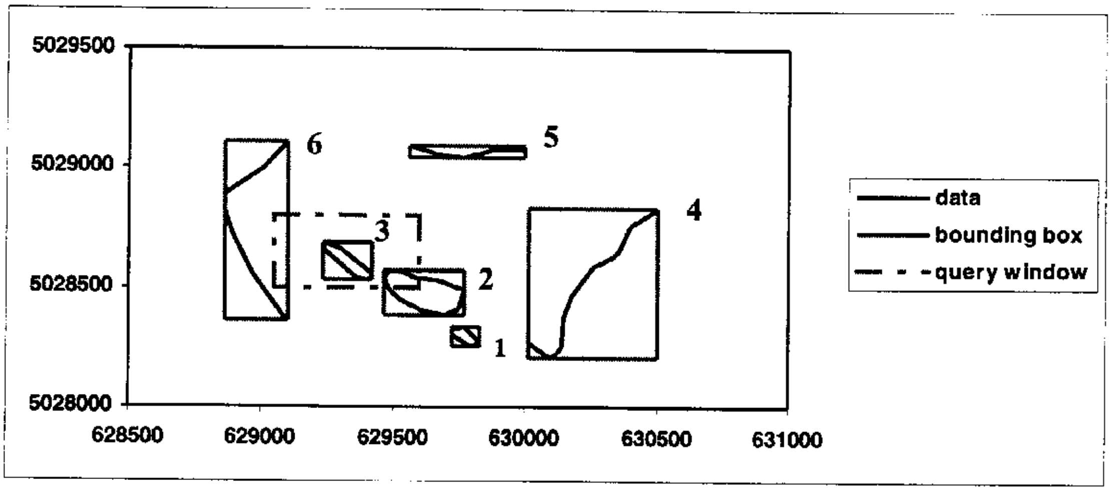


Figure 3.3 Example spatial query on six objects.

For a search based on bounding boxes of the objects, we will get objects 2, 3, and 6 reported in range. For a query based on real objects, we will only get objects 2 and 3 reported, since the real data of object 6 does not intersect the query window.

For testing, we chose to use query window (QW) areas of 1%, 4%, 9%, 16%, 25%, 36%, 49%, 64%, 81%, and 100% of the data extent area. These query window sizes are not from real life query logs; they are generated by 10% x 10%, 20% x 20%, ..., etc. of each dimension of data extent. (X_{MIN}, Y_{MIN}) and (X_{MAX}, Y_{MAX}) are the bottom-left and upper-right coordinates defining the data extent area, (XW_{min}, YW_{min}) and (XW_{max}, YW_{max}) are the bottom-left and upper-right coordinates defining the query windows (see Figure 3.4). With each different size of query window we ran 50 random searches.

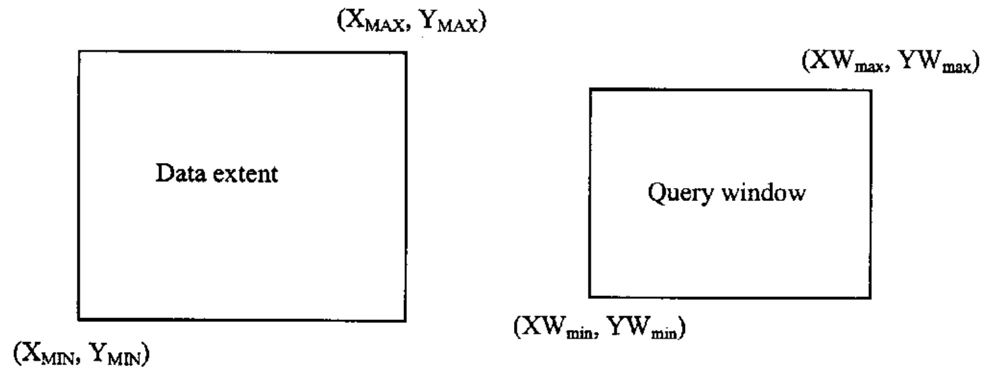


Figure 3.4 Coordinates of data extent and query window.

Figure 3.6 shows the algorithms for performing testing and generating random search windows 50 times. Figure 3.7 shows the R-tree testing algorithm for primary range search. The structures used in the algorithm are shown in Figure 3.5.

```
struct Rect
{ int b[4];//xmin, ymin, xmax, ymax
     //The position and len are needed to retrieve the detail data
     //from disk.
  long position; //byte offset
  int len;  //length, counts each four bytes as one
    // xArr and yArr is just place holder, it does not contain
    // real data until we do a secondary search.  Allocation
    // is done in Search function
  int* xArr;
  int* yArr;
};

struct Branch
{ Rect rect;
  Node *child;
};

struct Node
{ int count;
  int level; // 0 is leaf, others positive
  Branch b[m]; // m is the order of the R-tree.
};
```

Figure 3.5 Structures used for R-tree range search.

```
main()
{
1. Loop = 50;
2. for (j=0; j<Loop; j++)  {
3.   for (i=0; i<10; i++)  {
4.     SearchWindow(&w, &CoverAll, (i+1)*10);
5.     HitCount[i] += Search(Root, &w, &ResultList);}}
}
```

```
SearchWindow(Rect* window, Rect* data, int percentage)
{
1. float p;
2. p = (float)percentage/100.0;
3. for (i=0; i< DIMS; i++)  {
4.   range[i] = data->b[DIMS+i] - data->b[i];
5.   window->b[i] =(rand()%(int)((1.0-p)*range[i])+ data->b[i]);
6.   window->b[i+DIMS] = window->b[i] + p*range[i];}
}
```

Figure 3.6 Algorithms for testing and generating random search windows 50 times.

The code in main() function in Figure 3.6 are explained as follows:

Line 3: For 10 times (once per query window size). Line 4: Create a *random* search window based on the data extent. Line 5: Do a search using this generated search window.

The code in the SearchWindow() procedure in Figure 3.7 is explained as follows:

Line 4: For each dimension (2 in this case), range = Maximum value – Minimum value. Lines 5 and 6: Generate random range search windows defined by bottom-left and upper-right coordinates. Table 3.1 shows the boundary values as used in the code. Note that the testing for the range search is considered as best case testing, since the entire indexing structure is kept in memory.

Table 3.1 Boundary values for the range search testing algorithm.

Point	Corresponding code
XW_{min}, YW_{min}	`window->b[0], window->b[1]`
XW_{max}, YW_{max}	`window->b[2], window->b[3]`
X_{MIN}, Y_{MIN}	`data->b[0], data->b[1]`
X_{MAX}, Y_{MAX}	`data->b[2], data->b[3]`

```
int Search(Node *n, Rect *w, ListNode * ResultList)
{
int hitCount = 0;

if (n->level ==0) /* this is a leaf node */
   for (i=0; i< m; i++)
      if (n->b[i].child && Overlap(w, &n->b[i].rect))
         hitCount++;
         AddtoResultList(&n->b[i].rect, &ResultList);

else /* this is an internal node in the tree */
   for (i=0; i< m; i++)
      if (n->b[i].child && Overlap(w, &n->b[i].rect))
            hitCount += Search(n->b[i].child, w);

   return hitCount;
}
```

Figure 3.7 R-tree testing algorithm for primary range search.

Procedure Search() for primary range search is shown in Figure 3.7, and is explained with help of Figure 3.1(a) and (b) as follows:

First we check if the query window (QW) intersects with any rectangles in the root node. As we can see in Figure 3.1 (a) QW only intersects with R2. We prune R1 and follow the R2 link to the next level. This time QW intersects with R6 only. We prune R5 and follow the R6 link to the leaf level. Now we are at leaf node, and we check if QW intersects with any object rectangles in this node. Rectangles 2, F, and G, which are the bounding boxes of three real objects, are in range, so we report them.

The R-tree testing algorithm for secondary range search is split into two parts as shown in Figures 3.9 and 3.10. The Ptcoord structure used in the algorithm is defined as follows:

```
struct Ptcoord // point coordinate
{
  float x,y;
};
```

The sub-functions Overlap(), Contained() and twoline_intersect() are omitted; the details about these functions are shown in Appendix 2. Figure 3.8 shows the "overlap" and "contained" relationships between two rectangles d and w.

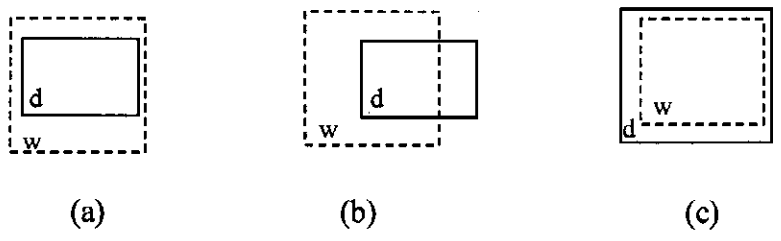


Figure 3.8 Relationship between two rectangles for overlapping and containing.

Rectangle w (dotted line) stands for a query window, and rectangle d (solid line) stands for a data object. Two rectangles in Figure 3.8 (a), (b), and (c) are all considered as overlapping. Two rectangles in Figure 3.8 (a) and (c) are considered as containing (w contains d in (a), d contains w in (c)).

```
int Search(Node * n, Rect * w)
{
  Ptcoord tempb, tempe, lineb, linee;
  if (n->level > 0) // this is an internal node in the tree
    for (i=0; i<m; i++) // m is the order of the r-tree
      if (n->b[i].child && Overlap(w, &n->b[i].rect))
        hitCount += Search(n->b[i].child, w);

  else // this is a leaf node
  {
    for (i=0; i<m; i++)

      //case 1: bounding rectangle of data is inside search window
      if (n->b[i].child && Contained ( &n->b[i].rect, w ))
        hitCount++;

      //case 2: bounding rectangle of data interact search window
      else if (n->b[i].child && Overlap(w, &n->b[i].rect))
      {
        fseek(theStream, n->b[i].rect.position, SEEK_SET);
        if (n->b[i].rect.xArr == NULL)
        n->b[i].rect.xArr = (int*)malloc(sizeof(int)*n->b[i].rect
        .itsLen);
        if (n->b[i].rect.yArr == NULL)
        n->b[i].rect.yArr = (int*)malloc(sizeof(int)*n->b[i].rect
        .itsLen);

        //Read in the real data
        fread(n->b[i].rect.xArr, sizeof(int), n->b[i].rect.itsLen,
        theStream);
        fread(n->b[i].rect.yArr, sizeof(int), n->b[i].rect.itsLen,
        theStream);

        intersect=0;
        //check if real data intersect with search window
        //for each segment of the real data

        for (k=0; k<(n->b[i].rect.itsLen)-1; k++)
          for (each side of the search window)
            if (twoline_intersect(data_segment.begin,
            data_segment.end, window_side.begin, window_side.end)
            hitCount++;

            break;
```

Figure 3.9 First part of the R-tree secondary range search algorithm.

```
    //case 3: search window is inside the bounding rectangle
        else
          {
          //check if the search window is inside the polygon data
          for (k=0; k<(n->b[i].rect.itsLen)-1; k++)
            {
            tempb.x=n->b[i].rect.xArr[k];
            tempb.y=n->b[i].rect.yArr[k];
            tempe.x=n->b[i].rect.xArr[k+1];
            tempe.y=n->b[i].rect.yArr[k+1];

            lineb.x = w->b[0];
            lineb.y = w->b[1];
            linee.x = INT_MIN;
            linee.y = w->b[1];

            if (w->b[1]>=tempb.y && w->b[1]<=tempe.y)
              if (twoline_intersect(tempb, tempe, lineb, linee))
                intersect_point++;
            }//end of for

            //check the number of intersect point is odd
            if (intersect_point%2 !=0)
              hitCount++;
          }//end of if

      free(n->branch[i].rect.xArr);
      free(n->branch[i].rect.yArr);
      }//end of elseif (end of case 2)
    }//end of for
  }//end of else
  return hitCount;
}
```

Figure 3.10 Second part of the R-tree secondary range search algorithm.

Case 1 in Figure 3.9 will handle situation (a) in Figure 3.8. If the data rectangle is contained in the query window, then the real data must be contained in the query window.

Case 2 in Figure 3.9 handles situation (b) in Figure 3.8. If the data rectangle overlaps the query window, the real data may or may not overlap the query window (see Figure 3.11). The polygon inside the rectangle d is the real data.

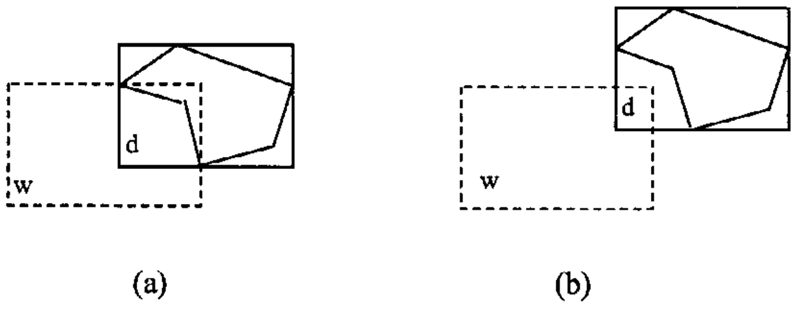


Figure 3.11 Real data may (a) or may not (b) intersect with QW for overlapping.

Case 3 in Figure 3.9 handles situation (c) in Figure 3.8. If the data rectangle contains the query window, the real data may or may not intersect with the query window (see Figure 3.12). At this point, all cases where intersection can occur have been tested.

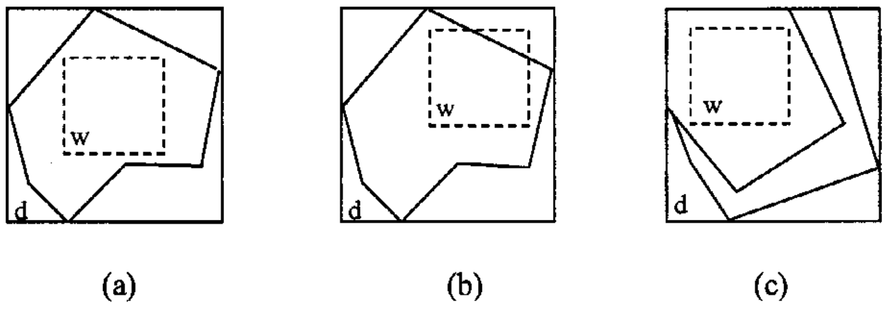


Figure 3.12 Real data may ((a) and (b)) or may not (c) intersect with QW for containing.

CHAPTER FOUR

MODIFIED GRID FILE INDEXING

This chapter investigates a new approach based on Hinrichs's grid file indexing method [Hinrichs , 1992]. Hinrichs's binary-radix bucket-region (BR^2) grid file directory was originally designed for point data. In order to handle non-point data, we modified Hinrichs's grid file. One object can be inserted into several buckets. For each bucket, we used an R-tree to store all the objects in that bucket. As far as we know, this is the first time this approach was used to index non-point data.

4.1 A Grid File Index

Grid file index for non-point data is almost the same as that for point data, except one object can be inserted into several buckets. In order to simplify the explanation, we use point data as an example to explain the concept of the construction and the properties of the BR^2 – directory grid file.

4.1.1 Construction of the BR^2 –tree for the grid file

The BR^2-directory structure is a tree structure. It grows linearly with the number of data buckets and reduces both the CPU-costs for directory operations and the I/O costs for directory pages. Figure 4.1 shows the construction of a BR^2 -directory for two-dimensional partitioning of the data space.

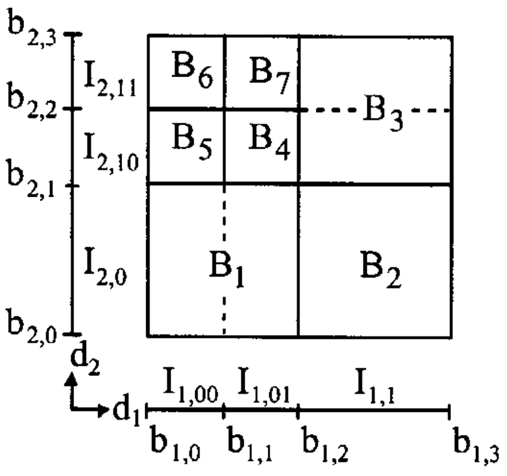


Figure 4.1 An example partitioning of 2D space.

In Figure 4.1, two scales each containing four boundaries $b_{j,0} \dots b_{j,3}$ determine the grid consisting of nine cells. I_j stands for intervals in dimension j. The binary sequence S determines how an interval $I_{j,S}$ is obtained by subsequent bisection from d_j : A zero (0) means selection of the lower interval, a one (1) selection of the upper interval. $B_1 \dots B_7$ represent the buckets. Each covers one grid cell with the exception of B_1 and B_3 .

Each boundary partitions the data space into two. For a grid file, which has been generated by insertions, only the buddy system [e.g. Samet, 1990, pp139] guarantees the existence of at least one boundary that does not intersect any bucket region, e.g. $b_{1,2}$ and $b_{2,1}$ in Figure 4.1. We select boundary $b_{1,2}$ that separates the buckets B_1 , $B_4 \dots B_7$ from the buckets B_2 and B_3 (Figure 4.2). Therefore the root node $N_{()}$ of the BR^2 -tree obtains an entry for the dimension of $b_{1,2}$, i.e. node dimension $d_{N_{()}} = 1$.

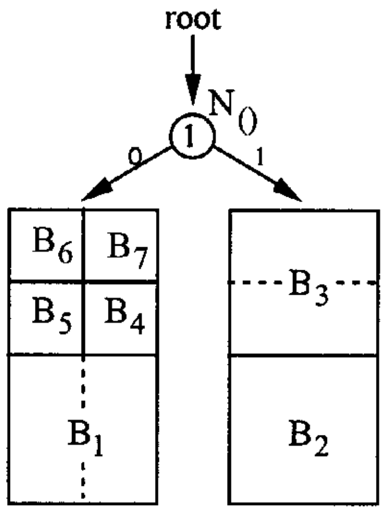


Figure 4.2 Bisection along boundary $b_{1,2}$.

We continue by selecting boundaries and represent them by new nodes, which contain their dimension as an entry. Figure 4.3 shows one of the possible BR^2 -trees for our example. The pointers to the leaves are the identifiers of the buckets on disk.

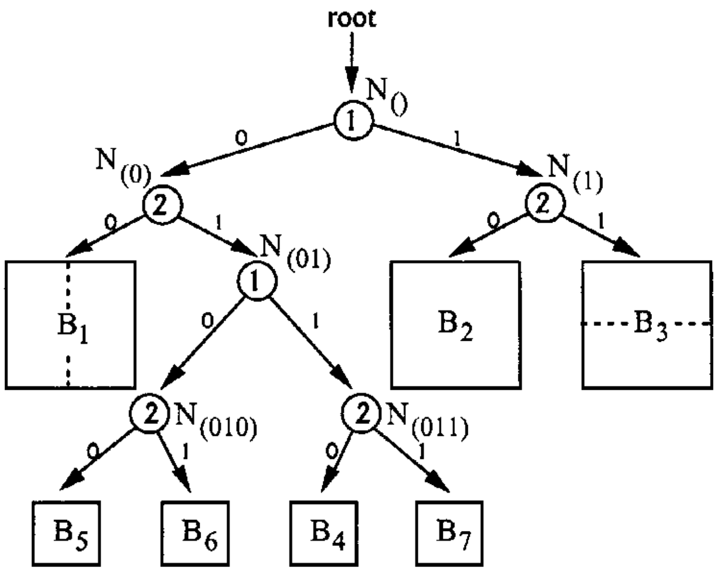


Figure 4.3 One possible BR^2-tree for the BR^2-directory in Figure 4.1.

4.1.2 Properties of BR^2 -trees

Let n be the number of buckets. Some properties of BR^2 -trees is shown as follows [Hinrichs, 1992]:

- In binary trees the number of internal nodes is equal to the number of leaves minus one. Therefore the size of the BR^2 -tree grows linearly in n, so the space requirement for a BR^2-tree is $\Theta(n)$.

- Each internal node corresponds to a unique boundary in one of the scales. A boundary may be assigned to more than one internal node.

- Each node N (internal node or leaf) represents a rectangular region D_N of the data space; this region is the Cartesian product of binary-radix intervals represented by the scales. The depth of a node or leaf determines the size of this region.

- Each internal node or leaf owns a unique path, which can be described by a bit-sequence S. In Figure 4.3, these bit sequences S form the node identifiers $N_{(S)}$. If S_1 is the bit-sequence of a parent node N_1 and S_2 the bit-sequence of a child node N_2, then S_1 is prefix of S_2 .

- The BR^2 -representation of the partitioning is not unique. In most cases there exist several different equivalent BR^2 –trees.

Each tree can be obtained from any equivalent tree by performing a number of transformations. Figure 4.4 shows a subdivision of one part of a data space.

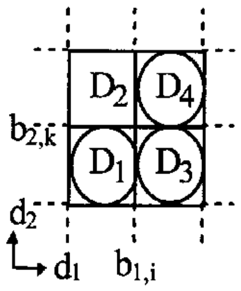


Figure 4.4 One subdivision of a data space.

D_1..D_4 in Figure 4.4 could be bucket regions or parent nodes of further subdivisions. The two equivalent BR^2 -subtrees representing this subdivision are shown in Figure 4.5.

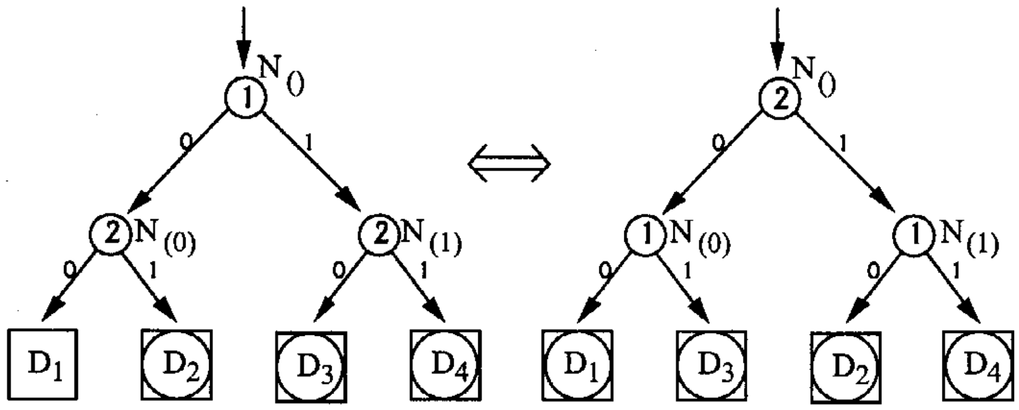


Figure 4.5 Two equivalent BR^2 –subtrees for the data subdivision shown in Figure 4.4.

In an implementation of the BR^2-tree by Hinrichs [Hinrichs, 1992], each node can be represented by 8 bytes; three bytes for each pointer to the children and two bytes for additional information, e.g. the node dimension. Since the identifiers for buckets are stored in their parent nodes we need 8(n – 1) bytes for n buckets. This is less than what is needed in the grid array representation [Hinrichs, 1985] (even under uniform data distribution) and the region representation [Hinterberger, 1987].

4.2 Operations on BR^2 -trees

Fundamental operations on directory structures are searching, splitting and merging buckets. In the following we discuss only the first two operations.

4.2.1 Range Searching

The range search is formally defined as follows: Search all buckets B whose bucket regions intersect a given query range $R = (I_1, \dots, I_k)$ where I_j is an interval in the range of dimension d_j. The query is performed in two steps as follows:

1. Determine for each interval $I_j = [l_j, u_j]$ two bit sequences L_j and U_j for the lower and upper bound of I_j in the binary-radix tree for dimension d_j.
2. Traverse the BR^2-directory.

Remember that each node N of the BR^2-directory is represented by a unique bit sequence S_N. For example, the representation (or path) to bucket B_4 in Figure 4.3 is $S_{B_4} = 1_0\, 2_1\, 1_1$

2_0 . The corresponding part of the data space is the Cartesian product of k binary radix intervals that can again be described by k unique bit sequences. Let $S_{N,j}$ be the bit sequence for dimension j (e.g. $S_{B_4,2} = 2_1 2_0$).

A node N is relevant for the range query if all sequences $S_{N,j}$ $(1 \le j \le k)$ satisfy $L_j \le_s S_{N,j} \le_s U_j$ where $\le_s$ is the lexicographic order on bit strings. Starting at the root of the tree, we distinguish the following cases at a node N that has node dimension j (see Figure 4.6):

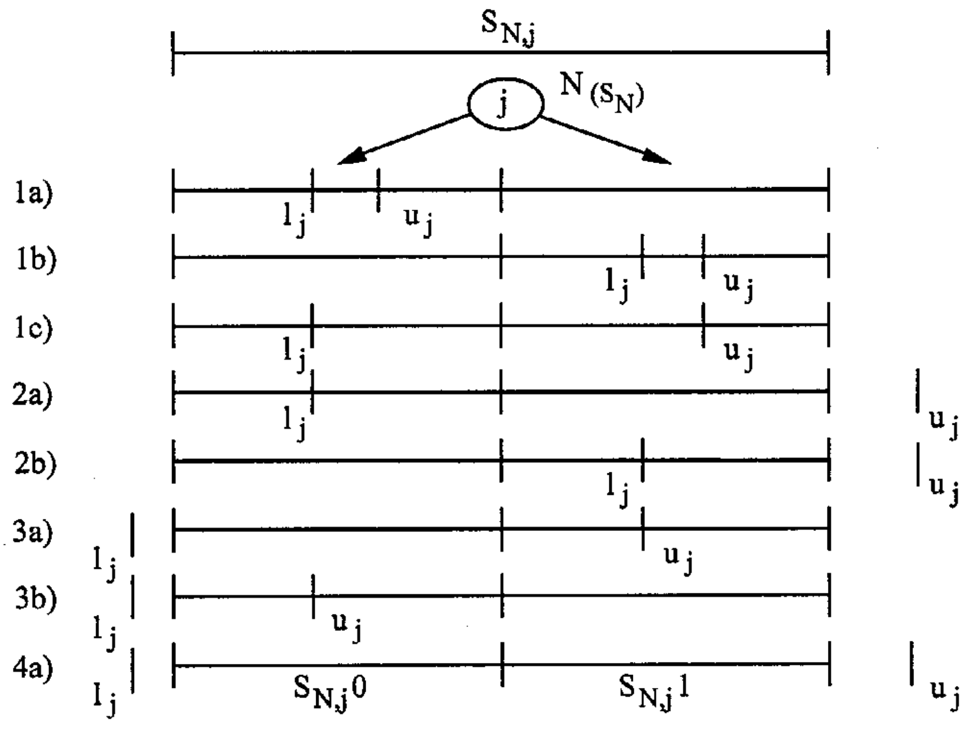


Figure 4.6 Eight cases for range query in the BR^2-tree.

1) $S_{N,j}$ is a prefix of L_j and U_j, i.e. the binary radix interval corresponding to the region of node N in dimension j contains both the lower bound l_j and the upper bound u_j. If

the current bits of the sequences L_j and U_j are both 0, we have to continue with the left child (1a), if the current bits are both 1, we have to continue with the right child (1b). If the current bits are different we have to visit both children (1c).

2) $S_{N,j}$ is a prefix of L_j, but not of U_j, i.e. the binary radix interval corresponding to the region of node N in dimension j contains the lower bound l_j. If the current bit of sequence L_j is 0 we have to visit both children of N (2a). Otherwise only the right child is considered (2b).

3) $S_{N,j}$ is a prefix of U_j, but not of L_j, i.e. the binary radix interval corresponding to the region of node N in dimension j contains the upper bound u_j. If the current bit of sequence U_j is 1 we have to visit both children of N (3a). Otherwise only the left child is considered (3b).

4) $S_{N,j}$ is neither a prefix of L_j nor of U_j, i.e. the binary radix interval corresponding to the region of node N in dimension j contains neither the lower bound l_j nor the upper bound u_j. Therefore we have to visit both children (4a).

It is not necessary to construct the bit sequences of a visited node because the prefix properties can be inherited from the parent. It is sufficient to introduce 2k status variables to describe the prefix properties of each bit sequence. The other tests in the algorithm are simple bit comparisons. The complete range search algorithm for non-point data is given in Figure 4.7.

The range search algorithm for a Grid file is done via a recursive procedure. This procedure will take the following formal parameters: the range to search, the root of the grid file tree and an AVL tree to put the results in. The AVL tree is used in the search algorithm in the grid file tree to avoid reporting duplicates.

```
void GDir::Search(int minx, int miny, int maxx, int maxy,
GNode* theRoot, AVLTree<Object>* T)
  //First we check if the tree passed in is in fact a valid tree.
  if (theRoot == NULL) return;
     //this is not a leaf node, we have to check if we should perform
     //the search down the left subtree, the right subtree, or both.
     //We know that a node is not the leaf node because its bucket is
     NULL (and therefore no data)

  if (theRoot->_bucket == NULL)   {
          //If the search window overlaps with the left subtree's
          //range, we should perform the search on the left subtree
          if (theRoot->Overlap(minx, miny, maxx, maxy, theRoot-
                >_node1->_minx, theRoot->_node1->_miny, theRoot-
                >_node1->_maxx, theRoot->_node1->_maxy)){
                //Search down the left tree recursively should the
                //above condition holds true
      Search(minx, miny, maxx, maxy, theRoot->_node1, T);}
    //If the search window overlaps with the right subtree's range, we
    //should perform the search on the right subtree
          if (theRoot->Overlap(minx, miny, maxx, maxy, theRoot-
                >_node2->_minx, theRoot->_node2->_miny, theRoot-
                >_node2->_maxx, theRoot->_node2->_maxy)){
                //Search down the right tree recursively should the
                //above condition holds true
      Search(minx, miny, maxx, maxy, theRoot->_node2, T);}}
     //When the bucket is not NULL, we know that this should be the
     //leaf node since there is data stored in the bucket

  else //now we are at the leaf node
     {     //Now that we are at the leaf node, we can just do the
          //search on the Rtree contained in this node in much the
          //same way as in chapter 3.
          //Call the Rtree search function
    theRoot->_bucket->_rtree.Search(minx, miny, maxx, maxy);
```

Figure 4.7 (part 1 of 2) Grid file range search algorithm.

```
    int i;
    for (i = 0; i < SearchResultCounter; ++i)
    { //Create a temporary object to consolidate all
      //information of the search result
      Object aTmp(SearchResult[i].boundary[0],
      SearchResult[i].boundary[1],SearchResult[i].boundary[2],
      SearchResult[i].boundary[3], SearchResult[i].position,
      SearchResult[i].len);

      //Insert the result into the AVL tree passed in as formal
      //parameter.
      T->Insert(aTmp);
  }//else if
}
```

Figure 4.7 (part 2 of 2) Grid file range search algorithm.

4.2.2 Bucket Splitting

A bucket B can overflow when a new data point $P = (k_1 ,k_2 ,\ldots,k_k)$ has to be inserted into B. If B overflows it has to be split and new disk blocks have to be allocated. Let S_j be the bit sequences determining the BR^2 -path to B, then the split can be performed as follows:

First the split dimension, i.e. the dimension according to which B is split, must be determined. If the bucket region covers more than one grid cell, B can be split along an existing boundary.

If no such existing boundary divides B's region, we select a split dimension j and a boundary b_j and insert the boundary into the particular radix tree of scale j. Now we can split B by allocating a new internal node N and a new bucket B'. The old parent of B now points to N. The node dimension of N is j, the left child is B, and the right child is B'. All data points stored in B are distributed between B and B' depending on their key value in dimension j. For non point data, if one object intersect with both of the subtrees (or children), we will insert the object in both subtrees. Even if we insert in both subtrees, we don't increase the memory requirement significantly because we only keep a pointer to the same location in the file.

The insertion of a new boundary in a scale costs O(log n) time for uniform data distribution. The remaining operations of a split require $\Theta(1)$ time. The algorithm of insertion and bucket splitting for non-point data is shown in Figure 4.8.

```
//Inserting into grid file tree.  We pass in the object, the root of
//the tree or subtree, the bounding box the underlining of the grid
//file tree or subtree (minx,miny,maxx,maxy), and the level.
void GDir::Insert(const Object& theObj, GNode *&theRoot, int gb_bbox,
int theLevel)
{
   if (theRoot == NULL)//first node of the tree only.  Called ONCE
   {
      theRoot = new GNode(minx, miny, maxx, maxy);
      //Insert the object in to the tree
      theRoot->_bucket->_rtree.Insert(theObj.bbox, theObj._pos,
theObj._size);
      //Change the size of the bucket (number of objects in the bucket)
accordingly
      theRoot->_bucket->_size++;
      return;
   }
   else //theRoot is not NULL
   {
      //Check if we can just insert it here or we have to go further
      //down, the bucket is not NULL indicating this is a leaf node
      if (theRoot->_bucket != NULL && theRoot->_bucket->Size() <
_limit)//this can contain data
      {
         theRoot->_bucket->_rtree.Insert(theObj.bbox, theObj._pos,
theObj._size);
         //Change the size of the bucket (number of objects in the
bucket) accordingly
         theRoot->_bucket->_size++;
         return;
      }
```

Figure 4.8 (part 1 of 3) Grid file insertion and bucket splitting algorithm.

```
    //If the bucket is NULL, we don't have any data in this tree just
    //yet.  This also means that this node is not the leaf node as
    //all leaf node should contain some data
    if (theRoot->_bucket == NULL)    {
       if (theLevel%2 == 1)//horizontal insert
       {
          //Check to see in which child node(S) should we insert data
          //Notice that in this implementation, if one object
          //intersect with both of the subtrees, we will insert the
          //object in both subtrees.
          if (theRoot->OverlapTop(theObj.bbox))
          {
             Insert(theObj, theRoot->_node1, _minx, (_maxy+_miny)/2,
             _maxx, _maxy, theLevel+1);
          }
          if (theRoot->OverlapBottom(theObj.bbox))
          {
             Insert(theObj, theRoot->_node2, _minx, _miny, _maxx,
             (maxy+_miny)/2, theLevel+1);
          }//if
          return;
       }//if
       else//VERTICAL insert
       {
          //Check to see which child node(S) should we insert
          if (theRoot->OverlapLeft(theObj.bbox))
          {
             Insert(theObj, theRoot->_node1, _minx, _miny,
             (maxx+_minx)/2, _maxy, theLevel+1);
          }//if
          if (theRoot->OverlapRight(theObj.bbox))
          {
             Insert(theObj, theRoot->_node2, (_maxx+_minx)/2, _miny,
             _maxx, _maxy, theLevel+1);
          }//if
          return;
       }//else if
    }//if
    //overflow, using the current level to keep track of splitting
    //direction
    if (theRoot->_bucket->Size() >= _limit)
    {
       if (theLevel%2 == 1)//horizontal split
       {
          theRoot->_node1 = new GNode(theRoot->_minx, (theRoot-
>_maxy+theRoot->_miny)/2, theRoot->_maxx, theRoot->_maxy);
          theRoot->_node2 = new GNode(theRoot->_minx, theRoot->_miny,
theRoot->_maxx, (theRoot->_maxy+theRoot->_miny)/2);
       }
```

Figure 4.8 (part 2 of 3) Grid file insertion and bucket splitting algorithm.

```
         else // VERTICAL split
         {
            theRoot->_node1 = new GNode(theRoot->_minx, theRoot->_miny,
(theRoot->_maxx+theRoot->_minx)/2, theRoot->_maxy);
            theRoot->_node2 = new GNode((theRoot->_minx+theRoot-
>_maxx)/2, theRoot->_miny, theRoot->_maxx, theRoot->_maxy);
         }
         //Cleanup the current bucket and put objects inside bucket
         //into appropriate subtree. We do this by searching inserting
         //each object in the current Rtree into the appropriate
         //subtree.
         theRoot->_bucket->_rtree.Search(theRoot->bbox);
         int i;
         for (i = 0; i < SearchResultCounter; ++i)
         {
            Object aTmp(SearchResult[i].boundary,
            SearchResult[i].position, SearchResult[i].len);
            if (theRoot->Overlap(SearchResult[i].boundary,
               theRoot->_node1->bbox))
            {
               Insert(aTmp, theRoot->_node1, theRoot->_node1->bbox,
theLevel+1);
            }
            if (theRoot->Overlap(SearchResult[i].boundary,
               theRoot->_node2->bbox))
            {
               Insert(aTmp, theRoot->_node2, theRoot->_node2->bbox,
theLevel+1);
            }
         }
         //Once inserted, we have to clear the _bucket in the current
         //node since it is not a leaf node anymore
         theRoot->Clear();

         //Now we can try to reinsert the same object at the same level
         //because the current tree now is one level deeper than
         //before.  This is a recursive call.
         Insert(theObj, theRoot, gb_bbox, theLevel);
      }
   }//else if
}
```

Figure 4.8 (part 3 of 3) Grid file insertion and bucket splitting algorithm.

4.3 Implementation

We implemented Hinrichs' BR^2-directory for non-point data in two dimensions. The following is the example in which we build a BR^2-tree for 6 objects. Let M = 2, i.e. for each bucket we allow a maximum of 2 objects to be stored. Similar to the R-tree implementation, we kept the bounding box and the location of objects in the BR^2-tree structure, and stored the real objects' data in a file on disk. The bounding boxes (indicated by the coordinates of bottom-left and upper-right vertices) of 6 objects are as follow:

```
Object1: (629722,5028256), (629829,5028331)
Object2: (629468,5028385), (629769,5028571)
Object3: (629231,5028527), (629420,5028682)
Object4: (630011,5028202), (630500,5028831)
Object5: (629563,5029040), (630000,5029095)
Object6: (628861,5028356), (629102,5029106)
```

The bounding box for all 6 objects is (628861,5028202), (630500,5029106). Figure 4.9 and 4.10 show the BR^2-directory and BR^2-tree structure for these 6 objects. There are 11 buckets. Note that one object may appear in several different buckets.

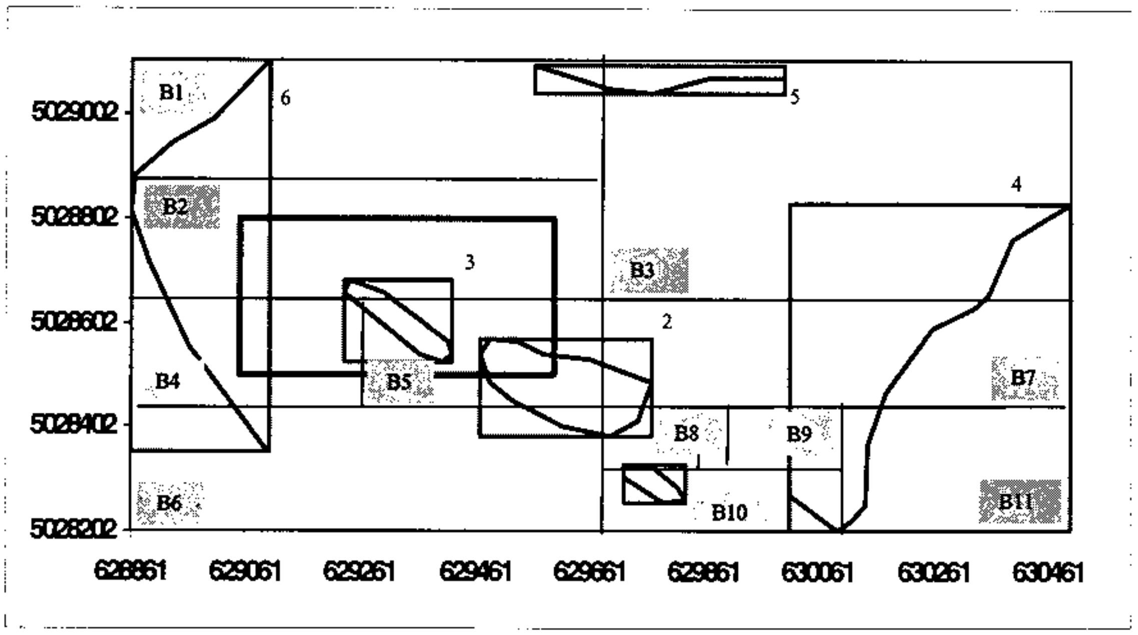


Figure 4.9 BR^2-directory for six objects.

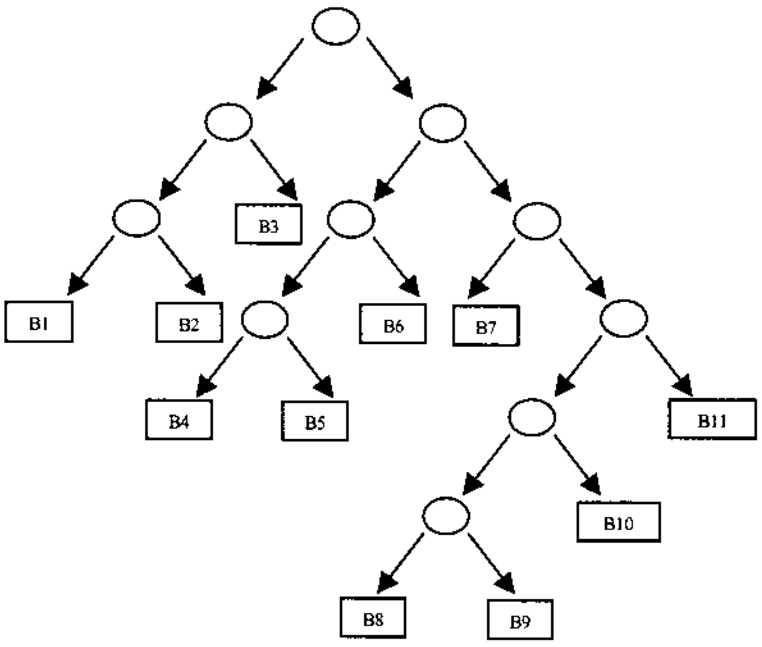


Figure 4.10 BR^2-tree structure for six objects.

For the data stored in each bucket, we used two different methods to index them; a linked list and an R-tree.

4.3.1 Linked list

This is a straightforward solution to keeping track of the data in each bucket. The linked list implementation is slow when the list gets longer since the list can only be traversed sequentially. In this particular case, the performance would degrade dramatically if we set M to be a large number. In addition, the linked list implementation only reports primary search (i.e. bounding rectangle intersection) results.

4.3.2 Combining Grid files and R-trees

It is clear that the linked list degrades the performance of Grid file tree. A possible solution is to replace the linked list with an R tree since the R tree offers a much better way to index data. Using an R tree, out-of-range branches are pruned so we do not have to traverse through the complete data set in each bucket. It is worthy to note that this solution is superior to using a grid file with a small linked list because the grid file tree would be very deep causing performance degradation.

In order to integrate the R tree into the Grid file tree easily, we have created a wrapper for the R tree original implementation (which is written in C) to provide a C++ interface with the grid file (which is written entirely in C++). Figure 4.11 shows the combination of R-tree and Grid file tree structures.

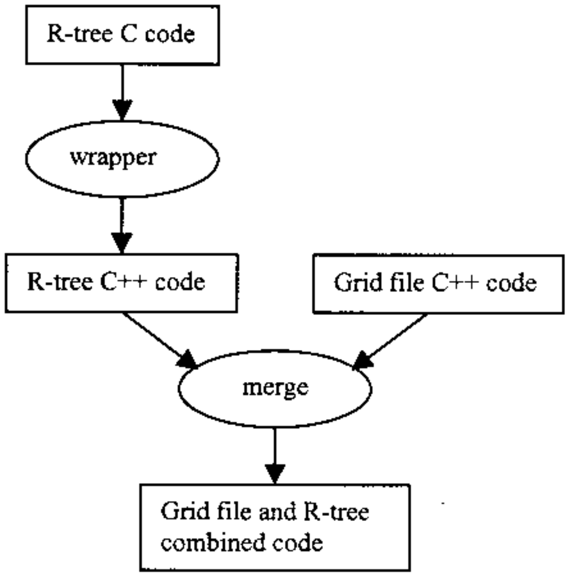


Figure 4.11 Combining grid file and R-tree software.

4.4 Testing

The range search includes search in the grid file tree and search in the R-tree. The naive range search reports redundant objects since one object may appear in several buckets. We used an AVL tree structure to report only unique objects during the range search. The testing algorithm for range search is shown in Figure 4.7.

We chose M = 1000 for the grid file and R-tree order m = 50 for the R-tree combined with the grid file structure. The same two sets of data (nts21g.dat and nts21g15.dat) used for testing with the R-tree and for Oracle Spatial were used for testing the grid file.

CHAPTER FIVE

RELATIONAL DATABASE SPATIAL INDEXING

5.1 Oracle 8i Spatial Data Cartridge

Oracle8i Spatial (often referred to as Spatial) provides a standard SQL schema and functions that facilitate the storage, retrieval, update, and query of collections of spatial objects in an Oracle8i database.

The release of Spatial used (Oracle 8i release 8.1.5) supports two mechanisms for representing geometry. The first, an object-relational scheme, uses a table with single column of type MDSYS.SDO_GEOMETRY and a single row per geometry instance. The second, a relational scheme, uses a table with a predefined set of columns of type NUMBER and one or more rows for each geometry instance. Details are described in the following sections entitled "Object-Relational Model" and "Relational Model".

Spatial uses a two-tier query model to resolve spatial queries. Two distinct operations are performed in order to resolve queries. The output of both operations yields the exact result set.

The two operations are referred to as primary and secondary filter operations. The primary filter permits fast selection of candidate records to pass along to the secondary filter. The primary filter compares geometry approximations to reduce computation complexity and is considered a lower cost filter. The secondary filter applies exact computations to geometries that result from the primary filter. It yields an accurate answer to a spatial query. The secondary filter operation is computationally expensive, but it is only applied to the primary filter results, not the entire data set.

Figure 5.1 illustrates the relationship between the primary and secondary filters.

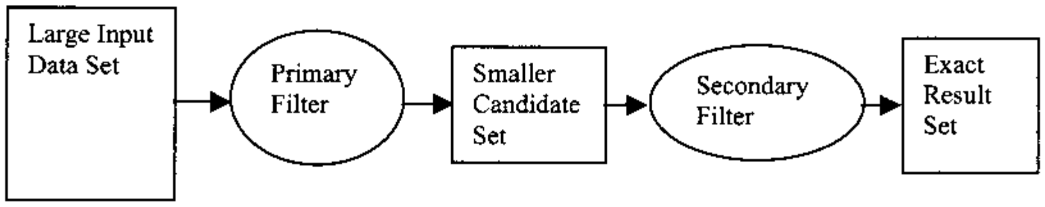


Figure 5.1 Oracle Spatial query model.

Spatial uses a linear quadtree-based spatial index to implement the primary filter [Oracle 8i, 1998]. The function SDO_GEOM.RELATE() is used as a secondary filter. It evaluates the topological relationship among two objects, such as whether two given geometries are touching, covering each other, or have any intersection.

The purpose of the primary filter is to quickly create a subset of the data and reduce the processing burden on the secondary filter. The primary filter therefore should be as efficient, that is selective yet fast, as possible. The efficiency of a primary filter is determined by the characteristics of the spatial index generated on the data.

5.2 Object – Relational Model

The object-relational implementation of Oracle8i Spatial consists of a set of object data types, an index method type, and operators on these types. A geometry is stored as an object, in a single row, in a column of type SDO_GEOMETRY. Spatial index creation and maintenance is done using basic Data Description Language (DDL) (CREATE, ALTER, DROP) and Data Manipulation Language (DML) (INSERT, UPDATE, DELETE) statements.

5.2.1 Object-Relational Data Structures

In the Spatial object-relational model, a single SDO_GEOMETRY object replaces the rows and columns in a <layername>_SDOGEOM table of the relational model.

The geometric description of a spatial object is stored in a single row, in a single column of object type SDO_GEOMETRY in a user-defined table. The table does not require the "_SDOGEOM" suffix anymore. Because the SDO_GEOMETRY type does not have an SDO_GID attribute, any table that has a column of type SDO_GEOMETRY must have another column, or set of columns, that defines a unique primary key for that table. Tables of this sort are sometimes referred to as geometry tables.

Oracle8i Spatial defines the object type SDO_GEOMETRY as shown in Figure 5.2.

```
CREATE TYPE sdo_geometry AS OBJECT (
SDO_GTYPE NUMBER,
SDO_SRID NUMBER,
SDO_POINT SDO_POINT_TYPE,
SDO_ELEM_INFO MDSYS.SDO_ELEM_INFO_ARRAY,
SDO_ORDINATES MDSYS.SDO_ORDINATE_ARRAY);
```

Figure 5.2 SDO_GEOMETRY object definitions.

The attributes of the SDO_GEOMETRY object type have the following semantics:

SDO_GTYPE - Indicates the type of the geometry.
For our test data, we only have type 2 (polyline) and type 3 (polygon).

SDO_SRID - Is reserved for future use. This attribute is intended to be a foreign key in a spatial reference system definition table.

SDO_POINT - Is an object type with attributes X, Y, and Z, all of type NUMBER. If the SDO_ELEM_INFO and SDO_ORDINATES arrays are both null, and the SDO_POINT attribute is non-null, then the X and Y values are considered to be the coordinates for a point geometry. Otherwise the SDO_POINT attribute is ignored by Spatial. You should store points in the SDO_POINT attribute for optimal storage.

SDO_ELEM_INFO - Is a varying length array of numbers. This attribute lets you know how to interpret the ordinates stored in the SDO_ORDINATES attribute.

Each triplet set of numbers conveys information about one geometry element, and a geometry may contain many elements. If a geometry has one element, then the SDO_ELEM_INFO array has three numbers; if the geometry has two elements, then the array has six numbers, and so on. Each triplet set is interpreted as shown in Figure 5.3.

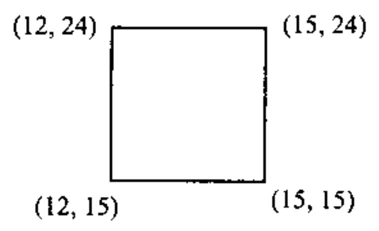
```
1. SDO_STARTING_OFFSET -- Indicates the offset within the
   SDO_ORDINATES array where the first ordinate for this
   element is stored. Offset values start at 1 and not at 0.

2. SDO_ETYPE - Indicates the type of the element. Valid values
   are 0 through 5.

   SDO_ETYPEs 1, 2, and 3, are considered simple elements.
   They are defined by a single triplet entry in the
   SDO_ELEMINFO array. SDO_ETYPEs 4 and 5 are considered
   compound elements. They contain at least one header triplet
   with a series of triplet values that belong to the compound
   element.

3. SDO_INTERPRETATION - Means one of two things, depending on
   whether or not SDO_ETYPE is a compound element.

   If SDO_ETYPE is a compound element (4 or 5), this field
   specifies how many subsequent triplet values are parts of
   the element.

   If the SDO_ETYPE is not a compound element (1, 2, or 3),
   the interpretation attribute determines how the sequence of
   ordinates for this element is interpreted. For example, a
   line string or polygon boundary may be made up of a
   sequence of connected straight line segments or circular
   arcs.
```

Figure 5.3 Definition of SDO_ELEM_INFO triplets.

A detailed description of valid SDO_ETYPE and SDO_INTERPRETATION value pairs is given in Oracle 8i Spatial User's guide and reference [Oracle 8i, 1998].

SDO_ORDINATES - Is a varying length array of NUMBER type that stores the coordinate values that make up the boundary of a spatial object. This array must always be used in conjunction with the SDO_ELEM_INFO varying length array.

Figure 5.4 illustrates a geometry with only one element.



```
SDO_GEOMETRY (3, NULL, NULL, SDO_ELEM_INFO_ARRAY (1,3,1),
   SDO_ORDINATE_ARRAY(12,15, 15,15,15,24,12,24,12,15))
```

Figure 5.4 Geometry polygon formed by straight lines.

The geometry metadata describing the dimensions, lower and upper bounds, and tolerance in each dimension must be stored as a single entry in a table named SDO_GEOM_METADATA created in your schema and defined as shown in Figure 5.5.

```
Create Table SDO_GEOM_METADATA (
  TABLE_NAME    VARCHAR2(30),
  COLUMN_NAME   VARCHAR2(30),
  DIMINFO       MDSYS.SDO_DIM_ARRAY);
```

Figure 5.5 SDO_GEOM_METADATA table description.

The SDO_GEOM_METADATA.TABLE_NAME column contains the name of a feature table, such as Roads or Parks, which has a column of type SDO_GEOMETRY. The name of this column, of type SDO_GEOMETRY, is stored in the feature table in the SDO_GEOM_METADATA.COLUMN_NAME column. For the tables Roads and Parks, this column is called theGeometry, and therefore the SDO_GEOM_METADATA table for Roads should contain rows (Roads, theGeometry, SomeDimInfo1). The

SDO_GEOM_METADATA.DIMINFO row is a varying length array of an object type, ordered by dimension, and has one entry per dimension.

5.2.2 Preprocessing

This section describes how to load spatial data into a database, including storing the data in a table with a column of type SDO_GEOMETRY and creating a spatial index for it. Figure 5.6 shows the overall approach to preprocess the data.

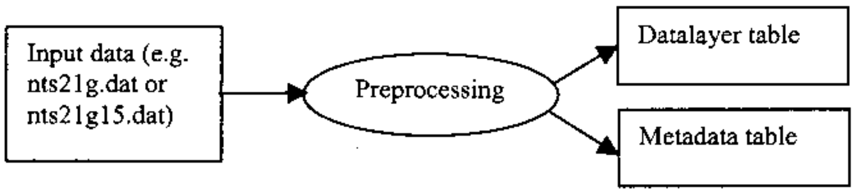


Figure 5.6 Overall approach for preprocessing data for Oracle 8i Spatial.

5.2.2.1 Loading Spatial Data

First we create and insert data in Metadata table. For the first set of data, which contains 20,000 objects, the two dimensions are named X and Y, their bounds are 577435 to 736453 in X dimension, 4982220 to 5098214 in Y dimension, and the tolerance for both dimensions is 1. The SQL statement for creating and loading the SDO_GEOM_METADATA table is shown in Figure 5.7.

```
CREATE TABLE SDO_GEOM_METADATA_20k (TABLE_NAME VARCHAR2(30),
COLUMN_NAME VARCHAR2(30), DIMINFO MDSYS.SDO_DIM_ARRAY);


INSERT INTO SDO_GEOM_METADATA_20k
  VALUES ('LAYER_20k', 'SHAPE', MDSYS.SDO_DIM_ARRAY(
MDSYS.SDO_DIM_ELEMENT('X', 577435, 736453, 1),
MDSYS.SDO_DIM_ELEMENT('Y', 4982220, 5098214,1)));
```

Figure 5.7 An example of creating and inserting data

into an Oracle 8i Spatial metadata table.

As an example of inserting data into the datalayer table, assume the geometry in Figure 5.4 represents a park. Table Parks can be created as shown in Figure 5.8 (a). The SQL statement for inserting the data for geometry OBJ_1 is shown in Figure 5.8 (b).

```
CREATE TABLE PARKS (NAME VARCHAR2(32),
                            SHAPE MDSYS.SDO_GEOMETRY);
```

(a)

```
INSERT INTO PARKS
  VALUES ('OBJ_1', MDSYS.SDO_GEOMETRY(3, NULL,NULL,
                        MDSYS.SDO_ELEM_INFO_ARRAY(1,3,1),
                        MDSYS.SDO_ORDINATE_ARRAY( 12,15, 15,15,
15,24,12,24,12,15)));
```

(b)

Figure 5.8 Creating and inserting data into table Parks.

The SQL statements for creating and inserting one object into table 'LAYER_20K' are listed in Figure 5.9 (a). Figure 5.9 (b) shows the code for generating SQL script to insert data into table 'LAYER_20K'.

```
CREATE TABLE LAYER_20k (NAME VARCHAR2(32),
                        SHAPE MDSYS.SDO_GEOMETRY);

INSERT INTO LAYER_20k
  VALUES('1', MDSYS.SDO_GEOMETRY(3, NULL, NULL,
                MDSYS.SDO_ELEM_INFO_ARRAY(1,3,1),
                MDSYS.SDO_ORDINATE_ARRAY(629829,5028263, 629816,5028287,
629766,5028331, 629722,5028330, 629722,5028305, 629792,5028256,
629829,5028263)));
```

Figure 5.9 (a) Creating and inserting data into datalayer table LAYER_20K.

For one object:

```
if (xArray[0] == xArray[arrayLen-1] && yArray[0] == yArray[arrayLen-1])
{
  j = sprintf(insertCommand, "EXEC SQL INSERT INTO LAYER_20K VALUES
  ('%d',
  MDSYS.SDO_GEOMETRY(3,NULL,NULL,MDSYS.SDO_ELEM_INFO_ARRAY(1,3,1),
  MDSYS.SDO_ORDINATE_ARRAY(", ++objCount);
  for (i = 0; i < arrayLen; ++i)
    {
      j += sprintf(insertCommand+j, "%d,%d, ", xArray[i], yArray[i]);
    }//end for
  sprintf(insertCommand+j-2, ")));\n\n");
}
else
{
  j = sprintf(insertCommand, "EXEC SQL INSERT INTO LAYER_20k VALUES
  ('%d',
  MDSYS.SDO_GEOMETRY(2,NULL,NULL,MDSYS.SDO_ELEM_INFO_ARRAY(1,2,1),
  MDSYS.SDO_ORDINATE_ARRAY(", ++objCount);
  for (i = 0; i < arrayLen; ++i)
    {
      j += sprintf(insertCommand+j, "%d,%d, ", xArray[i], yArray[i]);
    }//end for
  sprintf(insertCommand+j-2, ")));\n\n");
}
fwrite(insertCommand, sizeof(char), strlen(insertCommand),
theSQLOutput);
```

Figure 5.9 (b) Partial code for inserting data into datalayer table LAYER_20K.

This was used inside the rtree program (index.c).

Note that theSQLOutput is the data file that will contain the SQL statements for inserting all data into table LAYER_20K.

5.2.2.2 Index Creation

Once data has been loaded into the spatial tables, a spatial index must be created on the tables for efficient access to the data. This is done by approximating geometries with tiles. For each geometry, we have a set of tiles that fully cover the geometry.

Spatial provides two methods for spatial indexing; fixed and hybrid. Fixed indexing uses fixed-size tessellation whereas hybrid indexing uses a combination of fixed-size and variable-size tessellation. Hybrid indexing is recommended for the Spatial object-relational model [Oracle8i, 1998]. If specified correctly, it will provide better selectivity and spatial join performance for most data sets and application scenarios.

The tessellation algorithm used by the CREATE INDEX and index maintenance routines on INSERT, or UPDATE, is determined by the SDO_LEVEL and SDO_NUMTILES values supplied by the user in the PARAMETERS clause of the CREATE INDEX statement. SDO_LEVEL specifies the desired fixed-size tiling level, data type is NUMBER. SDO_NUMTILES specifies the number of variable-sized tiles to be used in tessellating an object, data type is NUMBER. The SDO_TUNE.ESTIMATE_ TILING_LEVEL() function can be used to determine an appropriate level for indexing. For the first set of test data, which contains 20,000 objects, the SDO_TUNE.ESTIMATE _TILING_LEVEL() function returned SDO_LEVEL = 10. To use fixed-size tile indexing, the SDO_NUMTILES parameter is omitted and the SDO_LEVEL value is set to the desired tiling level. To use hybrid tiling, both SDO_LEVEL and

SDO_NUMTILES must be greater than 1. The SDO_LEVEL value determines the number of fixed-size tiles, and the SDO_NUMTILES value determines the number of variable tiles that will be used to fully cover a geometry being indexed. Typically this value is small. For points, SDO_NUMTILES is always one. For other element types, we set SDO_NUMTILES to a value around 4 as suggested in Oracle 8i, [1998].

For example, assume that the LAYER_20K table has been loaded. The statement shown in Figure 5.10 is used to create the spatial index on LAYER_20K.SHAPE.

```
 CREATE INDEX LAYER_20K_INDEX ON LAYER_20K (SHAPE)
  INDEXTYPE IS MDSYS.SPATIAL_INDEX PARAMETERS ('SDO_LEVEL = 10,
SDO_NUMTILES = 4');
```

Figure 5.10. Creating the spatial index on LAYER_20K.SHAPE.

5.2.3 Search (Primary and secondary filter)Testing

An important concept in the spatial data model is that each geometry is represented by a set of exclusive and exhaustive tiles. This means that no tiles overlap each other (exclusive), and the tiles fully cover the object (exhaustive).
A typical spatial range query is to request all objects that lie within a defined fence or window. A query window is shown in Figure 5.11 by the dotted-line box. A dynamic query window refers to a fence that is not defined in the database, but that must be defined prior to using it.

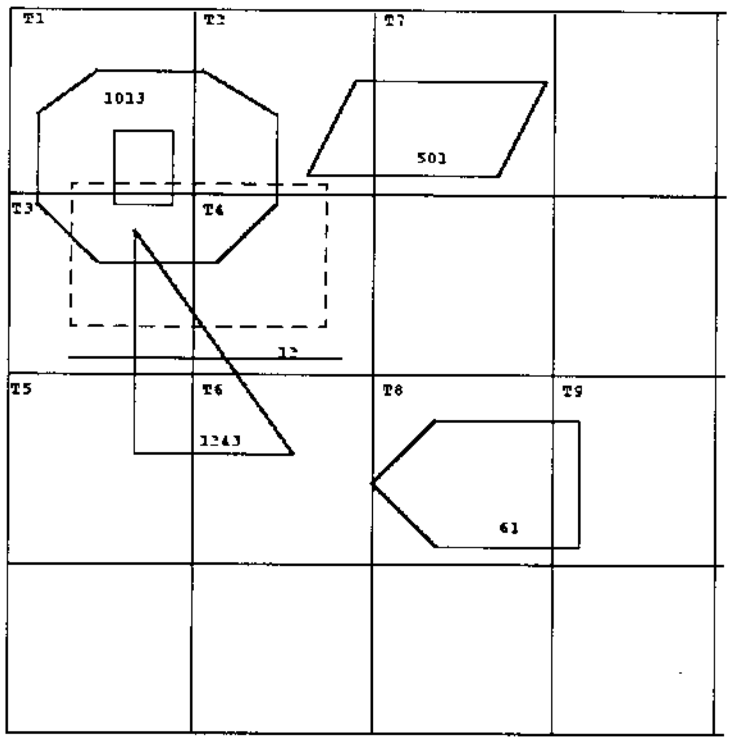


Figure 5.11 Tessellated Layer with a query window.

The statement in Figure 5.12 performs a primary filter operation without inserting the query window into a table. The window will be indexed in memory which should improve performance.

```
SELECT A.Feature_ID FROM TARGET A
WHERE mdsys.sdo_filter(A.shape, mdsys.sdo_geometry(3,NULL,NULL,
mdsys.sdo_elem_info(1,3,3), mdsys.sdo_ordinates(x1,y1, x2,y2)),
'querytype=window') = 'TRUE';
```

Figure 5.12 Primary filter with a temporary query window.

Note that (x1,y1) and (x2,y2) are the lower left and upper right corners of the query window. This statement will return all the geometries shown in Figure 5.11 that have an index tile in common with one of the index tiles that approximates the query window; i.e.

tiles T1, T2, T2, and T4. The result of the primary filter are geometries with IDs 1013, 1243, 12, and 501.

The SDO_RELATE( ) operator performs both the primary and secondary filter stages when processing a query. The statement in Figure 5.13 performs both primary and secondary filter operations without inserting the query window into a table. They return all the geometries in Figure 5.11 that lie within or overlap the query window. The result of these filters is objects 1243 and 1013.

```
SELECT A.Feature_ID FROM TARGET A
   WHERE mdsys.sdo_relate(A.shape, mdsys.sdo_geometry(3,NULL,NULL,
mdsys.sdo_elem_info(1,3,3),mdsys.sdo_ordinates(x1,y1, x2,y2)),
'mask=anyinteract querytype=window') = 'TRUE';
```

Figure 5.13 Secondary filter using a temporary query window.

5.3 Relational Model

Prior to release 8.1.5, the Spatial product always used four database tables to store and index spatial data. This schema is different from the Spatial object model introduced in the previous section. The four tables, used to store and index geometry, are collectively referred to as a layer. Tables 5.1 describes the schema of a Spatial layer.

<layername>_SDOLAYER Table or view

SDO_ORDCNT	SDO_LEVEL	SDO_NUMTILES
<number>	<number>	<number>

<layername>_SDODIM Table or View

SDO_DIMNUM	SDO_LB	SDO_UB	SDO_TOLERANCE	SDO_DIMNAME
<number>	<number>	<number>	<number>	<varchar>

<layername>_SDOGEOM Table or View

SDO_GID	SDO_ESEQ	SDO_ETYPE	SDO_SEQ	SDO_X1	SDO_Y1	...	SDO_Xn	SDO_Yn
<number>	<number>	<number>	<number>	<number>	<number>	...	<number>	<number>

<layername>_SDOINDEX Table or view

SDO_GID	SDO_CODE	SDO_MAXCODE
<number>	<raw>	<raw>

Table 5.1 Layers in an Oracle Spatial relational model database.

The detailed definitions for the column of each table are given in the Oracle 8i Spatial User's guide and reference [Oracle 8i, 1998].

5.3.1 Preprocessing

Similar to the object-relation model, we used ProC/C++ with embedded SQL to load the raw data into spatial tables and created the spatial index on the tables.

For the relational model, we created four tables and loaded data into the first three tables, Figures 5.14 and 5.15 show how to create and insert data into these tables for the first set of data nts21g.dat. Then we used the given procedure SDO_ADMIN.POPULATE_INDEX() to generate a spatial index for the data and insert into a fourth table which is the index table. Spatial provides two methods for spatial indexing; fixed and hybrid. Fixed indexing is recommended for the Spatial relational model. The tessellation algorithm used by the SDO_ADMIN.POPULATE_INDEX() and SDO_ADMIN.UPDATE_INDEX() procedures is determined by the values of the SDO_LEVEL and SDO_NUMTILES.

```
CREATE TABLE LAYER_20K_SDOLAYER(SDO_ORDCNT NUMBER, SDO_LEVEL NUMBER,
      SDO_NUMTILES NUMBER);
CREATE TABLE LAYER_20K_SDODIM(SDO_DIMNUM NUMBER, SDO_LB NUMBER, SDO_UB
      NUMBER, SDO_TOLERANCE NUMBER, SDO_DIMNAME VARCHAR(20));
CREATE TABLE LAYER_20K_SDOGEOM(SDO_GID NUMBER, SDO_ESEQ NUMBER,
      SDO_ETYPE NUMBER, SDO_SEQ NUMBER, SDO_X1 NUMBER, SDO_Y1 NUMBER,
      SDO_X2 NUMBER, SDO_Y2 NUMBER);
CREATE TABLE LAYER_20K_SDOINDEX(SDO_GID NUMBER, SDO_CODE RAW(255));

INSERT INTO LAYER_20K_SDOLAYER VALUES (4,10, NULL);
INSERT INTO LAYER_20K_SDODIM VALUES (1,577435,736453,1,'X');
INSERT INTO LAYER_20K_SDODIM VALUES (2,4983220,5098214,1,'Y');
```

Figure 5.14 Creating four tables and inserting data into the first two tables for layer_20k in SQL for the Oracle Spatial relational model.

```
//for each object;
h_sequence = 0;
h_serial++;
for (int i = 0; i < Len-1; ++i)
  h_x1 = xValues[i]; h_y1 = yValues[i];
  h_x2 = xValues[i+1];  h_y2 = yValues[i+1];
  if  (xValues[0] == xValues[Len-1] && yValues[0] == yValues[Len-1])
       EXEC SQL INSERT INTO LAYER_20K_SDOGEOM VALUES
       (:h_serial,0,3,:h_sequence, :h_x1, :h_y1, :h_x2, :h_y2);
  else
       EXEC SQL INSERT INTO LAYER_20K_SDOGEOM VALUES
       (:h_serial,0,2,:h_sequence, :h_x1, :h_y1, :h_x2, :h_y2);
  h_sequence++;
```

Figure 5.15 ProC/C++ coding for inserting one object into the third table for the Oracle Spatial relational model.

In Figure 5.15 Len is the number of vertices for each object, and xValues and yValues are arrays containing the x and y coordinates of the vertices for each object.

5.3.2 Search (primary and secondary filter) Testing

In a different approach compared to the object model, we must insert dynamic query windows in the tables and create an index for them. In order to do this, we installed the SDO_WINDOW package first, and then built a layer for the query window. Figure 5.16 shows how to define a query window.

```
SQL> EXECUTE MDSYS.SDO_WINDOW.CREATE_WINDOW_LAYER (fencelayer, DIMNUM1,
LB1, UB1, TOLERANCE1, DIMNAME1, DIMNUM2, LB2, UB2,TOLERANCE2,DIMNAME2);

SQL> EXECUTE DBMS_OUTPUT.PUTLINE(MDSYS.SDO_WINDOW.BUILD_WINDOW_FIXED
(comp_user, fencelayer, SDO_ETYPE, TILE_SIZE, X1,Y1, X2,Y2, X3,Y3,
X4,Y4, X1,Y1));
```

Figure 5.16 Defining a query window in SQL for the Oracle Spatial relational model.

After these two statements, the index table for the query window is automatically generated. For the objects in Figure 5.11, the primary filter query for the relational model is performed as shown in Figure 5.17.

```
SELECT DISTINCT A.SDO_GID
FROM <layer1>_SDOINDEX A, <fencelayer>_SDOINDEX B
WHERE                  A.SDO_CODE = B.SDO_CODE
  AND B.SDO_GID = (GID returned from SDO_WINDOW.BUILD_WINDOW_FIXED);
```

Figure 5.17. Primary filter for the relational model.

The result set of this query is the set of objects: {1013, 501, 1243, 12}.

The secondary filter performs exact geometry calculations of the tiles selected by the primary filter. The example in Figure 5.18 shows the primary and secondary filters combined:

```
SELECT SDO_GID
FROM <layer1>_SDOGEOM,
(
SELECT SDO_GID GID1
   FROM (
          SELECT DISTINCT A.SDO_GID
             FROM <layer1>_SDOINDEX A,
                  <fencelayer>_SDOINDEX B
             WHERE A.SDO_CODE = B.SDO_CODE
              AND B.SDO_GID = {GID returned from
SDO_WINDOW.BUILD_WINDOW_FIXED}
        )
   WHERE SDO_GEOM.RELATE('<layer1>', SDO_GID, 'ANYINTERACT', '<fence>',
1) = 'TRUE'    )
WHERE SDO_GID = GID1;
```

Figure 5.18. Primary and secondary filters for relational model.

This query would return all the geometry IDs that lie within or overlap the window. In this example, the results of the secondary filter are: {1243,1013}

5.4 Oracle Call Interface (OCI)

An alternative way to access Oracle 8i Spatial is via OCI. This is relatively new compared to the traditional ProC/C++. In this research, ProC/C++ was used exclusively. Time did not permit exploring the use of OCI with Oracle Spatial.

CHAPTER SIX

OBJECT ORIENTED DATABASE SPATIAL INDEXING

6.1 O_2 System Architecture

We attempted to use O_2 Spatial from Ardent Software for object oriented database spatial indexing [Ardent, 1998]. The system architecture of O_2 is illustrated in Figure 6.1.

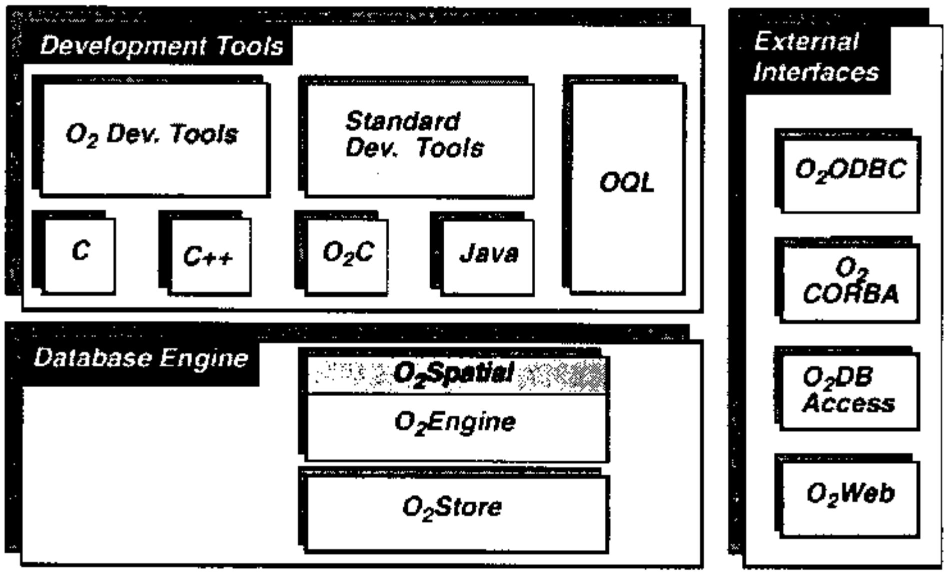


Figure 6.1: O_2 system architecture.

The O_2 system can be viewed as consisting of three components. The *Database Engine* provides all the features of a database system and an object-oriented system. This engine can be accessed with *Development Tools*, such as various programming languages, O_2 development tools and any standard development tool. Numerous *External Interfaces* are provided. The details of each component are described in the O_2 Spatial User Manual [Ardent, 1998].

The object database engine provides direct control of schemas, classes, objects and transactions through the O_2Engine API. It provides full text indexing and search capabilities with O_2Search and spatial indexing and retrieval capabilities with O_2Spatial.

6.2 O_2 Spatial

The O_2 Spatial module enables us to create, delete, update and query a spatial index associated with a collection of persistent geographical objects. It uses the O_2 object database management group (ODMG) C++ binding and the OQL query language.

The data structure for the spatial indexing mentioned in [Ardent, 1998] is the R-Quad tree, which stands for rectangle-quadrant tree. We have not been able to determine the precise meaning of this term.

When using the O_2 Spatial engine the spatial index stores a geo-referenced object as a minimum rectangle (called a bounding box) that encompasses the object for identifying the spatial location of the object. To index an object, we simply provide the object and its bounding box. This means O_2 Spatial only checks if the bounding box of an object intersects with the query window for a range search.

6.3 Best Efforts

I have been in contact with the Ardent Software technical support personnel since we started to install the system on a machine in GE 136 in February of 1999. Appendix 1 contains details of contacts made with Ardent support stuff by Emails. Ardent did not have any C++ sample code that would run on Windows NT. They did send me some code so I could do system testing after we installed O_2 Spatial. The code they sent had many errors. After debugging, it ran once out of ten times. The error message said "theBag, which is the collection for the reported objects, could not be created." It was very difficult to get the code to run again after making any changes. The system seemed very unstable. When the O_2 Spatial program did run, it seemed that the speed for range search was faster than that of the Oracle 8.0.4 relational model. This was just my impression. We didn't have the exact time recorded as the program stopped running before completing the test.

I obtained a sample O_2 Spatial test code from a local company, Universal System Limited (USL). Their code works on the Unix platform. After making certain changes, I got it to run on Windows NT and I obtained some results from it. The program reported the number of objects in a query range. Once the O_2 Spatial code mentioned above failed to run, the USL code would not run either. This gave me the impression that O_2 retains some variables from the previous sessions and uses it for the new session, which is not correct. I sent four emails from April to June 1999 to Ardent Spatial support people about this problem, but never got any useful help. At this point we decided to stop the research work with O_2 Spatial starting from July 1999. Universal Systems Limited also stopped their O_2 development due to the lack of support and functional deficiencies.

CHAPTER SEVEN

COMPARISON OF THE SPATIAL INDEXING METHODS

7.1 Experimental Description

Experiments for the R-tree, the BR^2-grid file, the Oracle 8i object model and Oracle 8i relational model were conducted using two sets of data: nts21g15.dat with 17,000 objects (called the test1 dataset) and nts21g.dat with 20,000 objects (called the test2 dataset).

The programs for building and searching the indices were developed on a machine with a Pentium II 400 MHz processor and 128 MB memory. The compiler used was MS Visual C++ 6.0 under Windows NT 4.0. The Oracle 8i database was run on an IBM Netfinity 5000 server with a Pentium II 450 MHz processor, 2 GB of RAM, running MS Windows NT Server 4.0.

The time for primary search and secondary search spatial indices using the four approaches was measured. The amount of source code required for the R-tree, modified grid file, Oracle 8i object-relational model and Oracle 8i relational model are 2452, 3637, 104 and 523 lines respectively. The memory space cost for the R-tree and modified grid file are approximately 1.5 Mbytes (33%) and 3.3 Mbytes (73%) for dataset test1, 1.0 Mbytes (31%) and 2.5 Mbytes (78%) for dataset test2. The percentages are the ratios of the index memory space cost to the original dataset size. The dataset size is the space required for the control and detail files together; i.e. 4.5 Mbytes for test1, 3.2 Mbytes for test2. The space for the Oracle relational and object-relational tables and files are approximately 275.8 Mbytes and 48.8 Mbytes for test1, and 270Mbytes and 62.7Mbytes for test2. The details about computing the space cost for Oracle are shown in Appendix 3.

7.2 Times for Range Searches Using R-tree Spatial Indices

We chose two orders for R-tree indexing; m=5 and m=50. Tables 7.1 and 7.2 (or Figure 7.1) show the times for primary and secondary search on dataset test1. Tables 7.3 and 7.4 (or Figure 7.2) show the times for primary and secondary search on dataset test2. SD stands for the standard deviation for 50 searches. It took 15.800, 16.494, 11.316 and 11.667 seconds to build the R-tree spatial indexes for the test1 dataset (m=5 and m=50) and the test2 dataset (m=5 and m=50), respectively.

Table 7.1 Average primary R-tree range search time (in seconds) for dataset test1.

Size(%)	m=5	SD	m=50	SD
1	0.00100	0.00001	0.00100	0.00001
4	0.00542	0.00003	0.00664	0.00003
9	0.01140	0.00002	0.00942	0.00002
16	0.01882	0.00005	0.01642	0.00004
25	0.02944	0.00001	0.02724	0.00002
36	0.04912	0.00005	0.04344	0.00004
49	0.05308	0.00007	0.04650	0.00006
64	0.07786	0.00004	0.07026	0.00004
81	0.09898	0.00002	0.09076	0.00003
100	0.11334	0.00002	0.09972	0.00001

The size column gives the ratios of query window area to the data extent area, in %.

Table 7.2 Average secondary R-tree range search time (in seconds) for dataset test1.

Size(%)	m=5	SD	m=50	SD
1	0.07694	0.00086	0.07108	0.00076
4	0.13232	0.00110	0.13714	0.00090
9	0.20364	0.00215	0.16788	0.00044
16	0.30974	0.00202	0.31550	0.00090
25	0.39008	0.00496	0.39232	0.00320
36	0.45992	0.00401	0.46026	0.00362
49	0.52876	0.00386	0.54160	0.00434
64	0.58084	0.00484	0.60892	0.00518
81	0.54596	0.00587	0.54554	0.00641
100	0.12042	0.00003	0.10758	0.00004

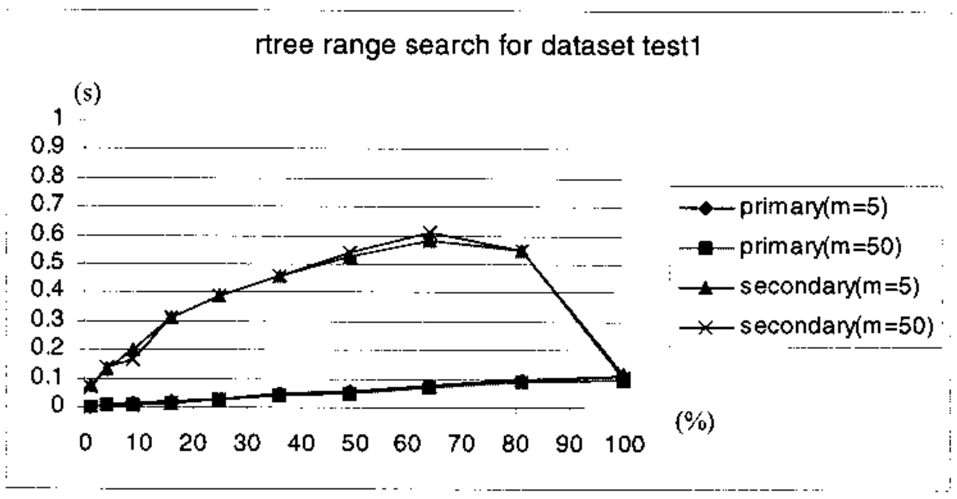


Figure 7.1 Average R-tree range search time (in seconds) for dataset test1.

Table 7.3 Average primary R-tree range search time (in seconds) for dataset test2.

Size(%)	m=5	SD	m=50	SD
1	0.00160	0.00001	0.00380	0.00002
4	0.00442	0.00003	0.00562	0.00003
9	0.01022	0.00001	0.01320	0.00002
16	0.02102	0.00001	0.02524	0.00003
25	0.03264	0.00002	0.03242	0.00002
36	0.05188	0.00003	0.05254	0.00002
49	0.07590	0.00002	0.07726	0.00004
64	0.10558	0.00004	0.10418	0.00003
81	0.11734	0.00002	0.11438	0.00003
100	0.12718	0.00003	0.12636	0.00003

Table 7.4 Average secondary R-tree range search time (in seconds) for dataset test2.

Size(%)	m=5	SD	m=50	SD
1	0.01102	0.00004	0.01602	0.00007
4	0.05566	0.00017	0.06192	0.00020
9	0.11400	0.00031	0.12010	0.00028
16	0.18162	0.00219	0.19114	0.00215
25	0.28082	0.00453	0.29178	0.00458
36	0.38130	0.00442	0.37298	0.00483
49	0.48618	0.00864	0.49666	0.00889
64	0.45764	0.00299	0.45146	0.00332
81	0.40996	0.00245	0.39440	0.00198
100	0.14002	0.00005	0.12256	0.00003

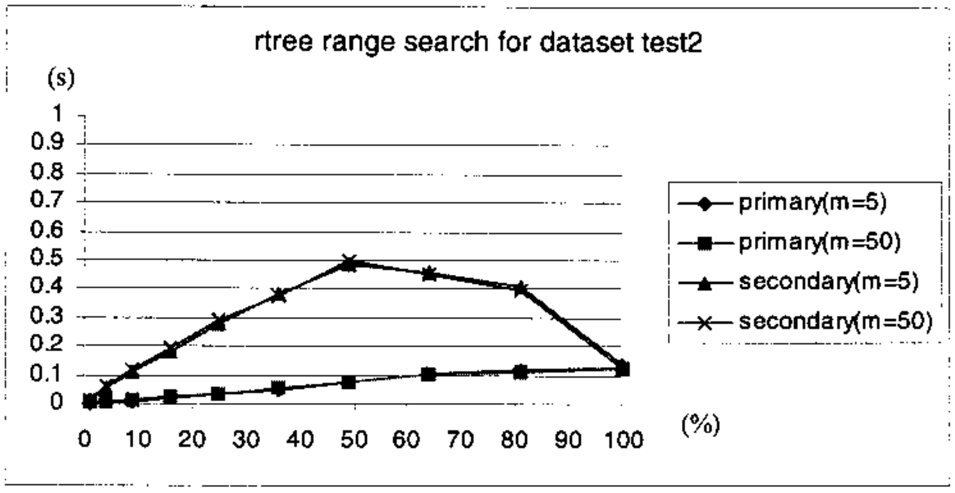


Figure 7.2 Average R-tree range search time (in seconds) for dataset test2.

7.3 Times for Range Searches Using Grid File Spatial Indices

We chose to use M=1000 for the grid file, i.e. each bucket is allowed to store a maximum of 1000 objects. The data inside each bucket were indexed using an R-tree with order m=50. Table 7.5 (or Figure 7.3) and Table 7.6 (or Figure 7.4) list the times for range search on both test datasets. The testing was performed 50 times for each different size of query window. It took 9.453 and 6.329 seconds to build the modified grid file spatial index for the test1 and test2 datasets.

Table 7.5 Average range search time (in seconds) for dataset test1

using modified grid files.

Size(%)	Primary	SD	Secondary	SD
1	0.00640	0.00598	0.01942	0.00767
4	0.02902	0.01933	0.05890	0.02627
9	0.05772	0.02221	0.10196	0.03188
16	0.12070	0.03286	0.15620	0.02967
25	0.19196	0.01470	0.27018	0.02730
36	0.27944	0.04978	0.37558	0.06269
49	0.39294	0.05674	0.45046	0.05307
64	0.54560	0.03684	0.62046	0.04590
81	0.71482	0.01750	0.80076	0.01788
100	0.83498	0.00530	0.82638	0.00550

Table 7.6 Average range search time (in seconds) for dataset test2

using modified grid files.

Size(%)	Primary	SD	Secondary	SD
1	0.00000	0.00000	0.00340	0.00479
4	0.00862	0.00406	0.02862	0.00834
9	0.04304	0.00579	0.07806	0.01130
16	0.10350	0.00752	0.14662	0.01460
25	0.17364	0.00713	0.24472	0.01795
36	0.31256	0.03855	0.41024	0.03482
49	0.50886	0.03564	0.62812	0.06095
64	0.75142	0.04498	0.85938	0.05953
81	0.84926	0.02848	0.94916	0.02777
100	0.99948	0.06132	0.97210	0.00778

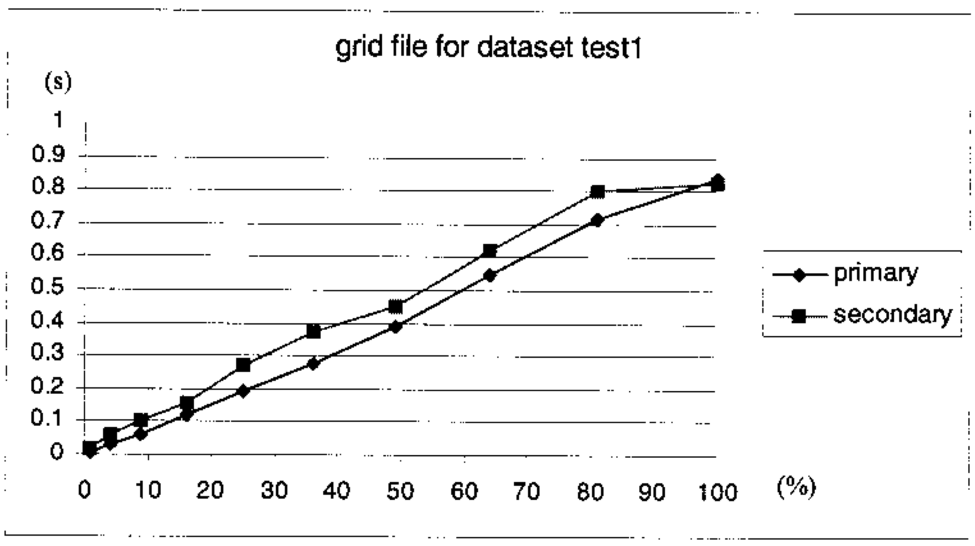


Figure 7.3 Average range search time (in seconds) for dataset test1

using modified grid files.

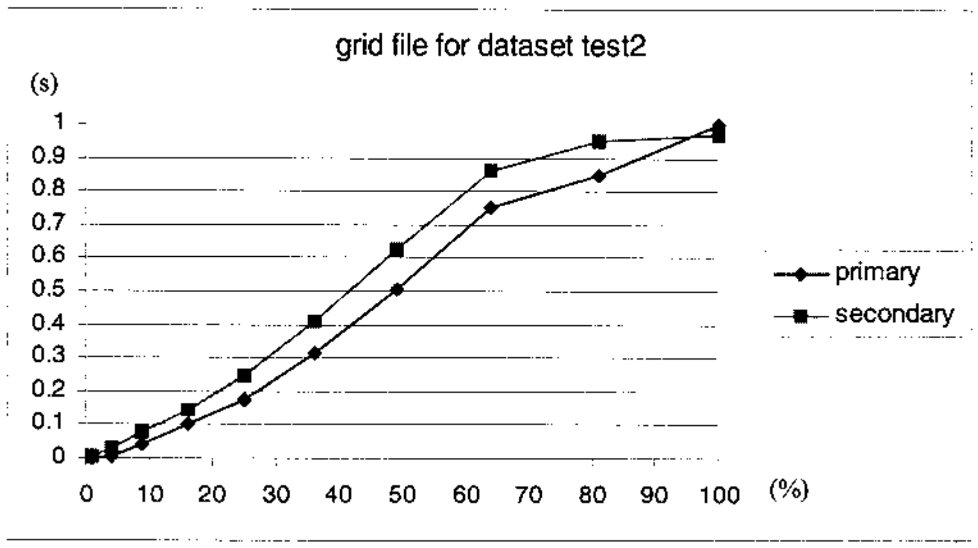


Figure 7.4 Average range search time (in seconds) for dataset test2

using modified grid files.

7.4 Times for Range Searches Using Oracle 8i Object-relational and Oracle 8i Relational Models

Table 7.7 (or Figure 7.5) and Table 7.8 (or Figure 7.6) list the time for range search on both datasets using Oracle 8i object-relational model. The testing was run 10 times for each different size of query window. It required approximately 660 and 900 seconds to build the spatial index for dataset test1 and test2, respectively.

Table 7.7 Average time for range search
using Oracle 8i object-relational model on dataset test1.

Size(%)	Primary(s)	SD	Secondary(s)	SD
1	0.270	0.014	1.863	0.156
4	0.401	0.000	10.060	4.228
9	0.796	0.545	13.074	1.820
16	2.689	0.120	27.074	3.817
25	3.986	0.313	44.995	5.849
36	5.833	0.134	66.796	8.370
49	8.773	0.694	90.976	9.737
64	13.049	0.566	N/A	N/A
81	20.205	0.163	N/A	N/A
100	25.232	0.375	N/A	N/A

Table 7.8 Average time for range search
using Oracle 8i object-relational model on dataset test2.

Size(%)	Primary(s)	SD	Secondary(s)	SD
1	0.171	0.042	0.906	0.021
4	0.351	0.001	5.743	2.414
9	1.327	0.446	14.330	4.617
16	3.265	0.396	22.015	1.680
25	5.821	1.618	40.018	2.847
36	8.095	0.841	69.307	10.717
49	13.484	1.297	113.868	13.999
64	25.213	1.694	157.862	11.734
81	30.705	1.631	179.193	2.189
100	35.867	1.750	207.103	6.620

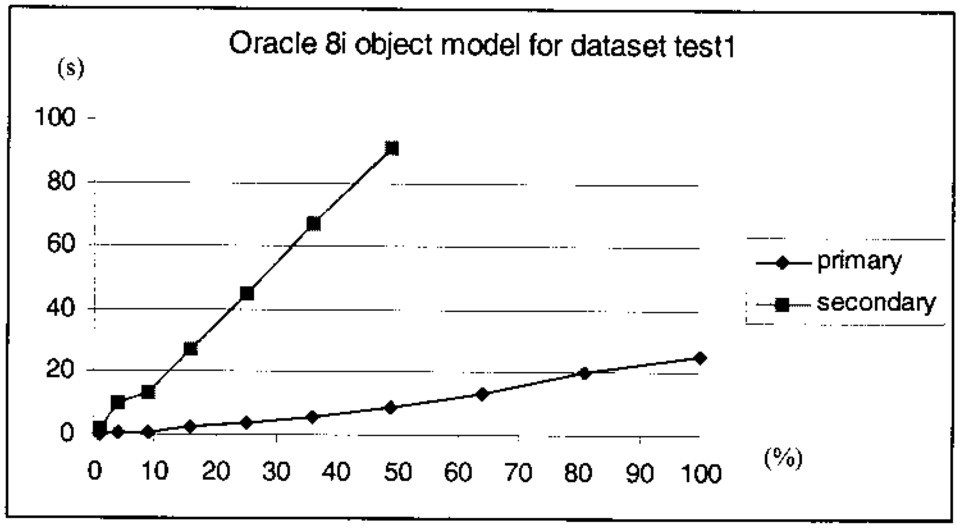


Figure 7.5 Average time for range search
using Oracle 8i object-relational model on dataset test1.

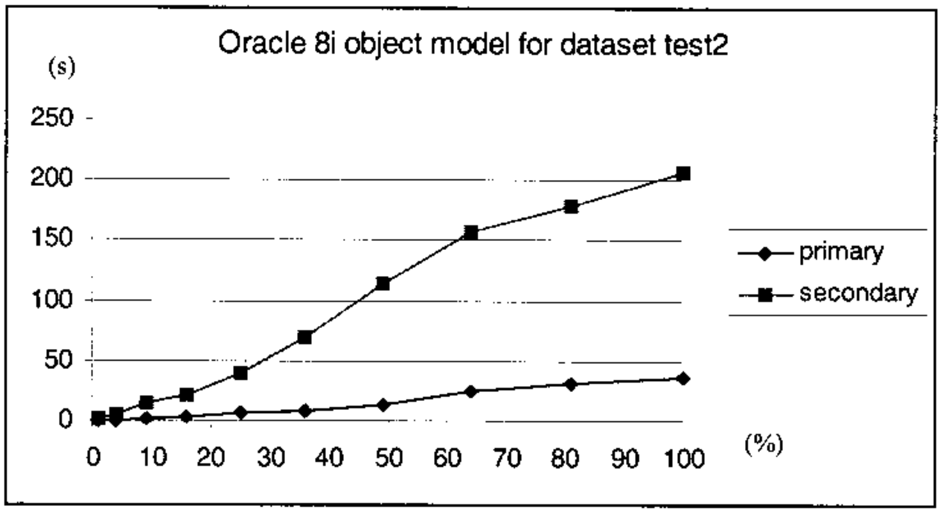


Figure 7.6 Average time for range search
using Oracle 8i object-relational model on dataset test2.

Although the Oracle 8i object-relational model is recommended in the Oracle 8i Spatial users guide, it suffers from several problems. For example, the SDO_LEVEL value returned from a recommended procedure SDO_TUNE.ESTIMATE_TILING_LEVEL() is not efficient to perform range searches. For our datasets, the procedure returned SDO_LEVEL values of 8 and 10 for test1 and test2. We used these values to generate a spatial index, and then perform primary and secondary range searches using this spatial index. When the query window size was increased up to 49%, the primary search for test1 dataset generated the error massage "too many tiles to create spatial index for window object". The same thing happened for the test2 dataset for a window size of 4%. In order to complete a range search with our 10 differently sized windows, we reduced the level values and set SDO_LEVEL to 5 for the test1 dataset and SDO_LEVEL to 7 for the test2 dataset. Another problem encountered is that the secondary range search with certain sizes of query windows seems to be running in an infinite loop. We can see this from Table 7.8, the fourth and fifth columns, where N/A means not available due to this problem.

Table 7.9 (or Figure 7.7) lists the time for range search on the test2 dataset using the Oracle 8i relational model. The Oracle 8i relational model seems unstable compared to the Oracle 8i object-relational model. I could only generate a spatial index for the test2 dataset and perform primary and secondary range searches based on this generated spatial index. The time required to generate the Oracle 8ia relational index for dataset test2 was approximately 480 seconds. The commands to use two procedures to generate the spatial index for the test1 dataset and the according error messages are shown in Figure 7.8.

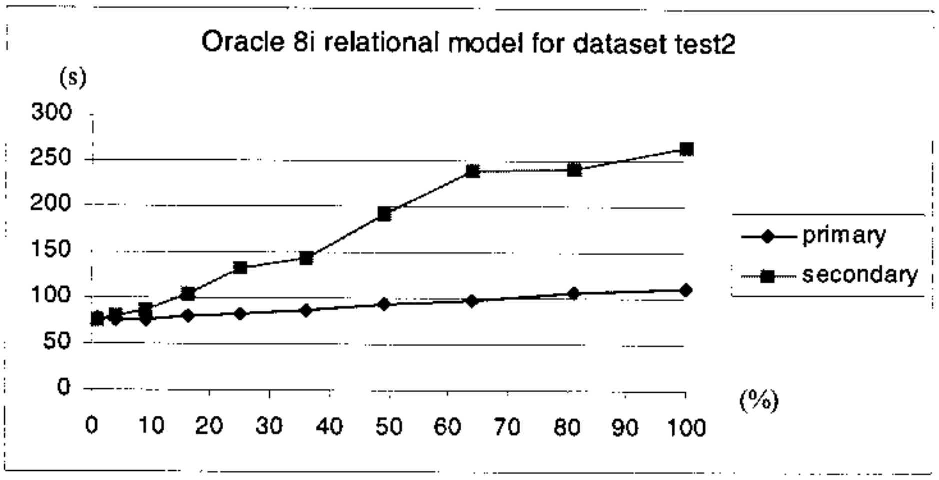


Figure 7.7 Times for range search using Oracle 8i relational model for test2 dataset.

```
EXECUTE SDO_ADMIN.POPULATE_INDEX_FIXED('LAYER17',10,FALSE,FALSE,FALSE);

ERROR ORA-13041: failed to compare tile with element LAYER17.3698.0

EXECUTE SDO_ADMIN.POPULATE_INDEX('LAYER17');

ERROR ORA-00604: error occurred at recursive SQL level 1
```

Figure 7.8 Two procedures and error messages
for generating the spatial index on dataset test1.

The second procedure SDO_ADMIN.POPULATE_INDEX() is recommended in the user guide.

Table 7. 9 Times for range search using Oracle 8i relational model for test2 dataset.

Qw size (%)	Primary (s)	Secondary (s)
1	77.661	75.399
4	76.450	81.447
9	75.739	87.686
16	79.704	103.900
25	81.798	132.641
36	86.755	142.725
49	92.843	191.586
64	97.019	238.834
81	105.472	241.267
100	110.699	265.602

Since the number of objects reported from the secondary range search with a 100% query window size was 15,000 instead of 20,000, there seems to be a fundamental problem with the Oracle 8i relational model. It does not seem to find all the objects in range. This model was used before Oracle 8i Spatial was released. For performance reasons, its use is not recommended in the latest release [Oracle8i, 1999].

CHAPTER EIGHT

SUMMARY AND CONCLUSION

In this research we compared the performance of four spatial indexing methods for range searches. One of the four methods is a new spatial indexing method based on the grid file index. Two sets of data from the National Topological Database were used to perform the experimental comparisons. A 1:50,000 scale data set with 20,000 objects as well as a 1:250,000 scale data set with 17,000 objects were used. All the data sets contain two-dimensional non-point objects.

In order to do the comparison, we first built spatial indices for objects. There are two ways to create a spatial index. These include 1) use the spatial data support in commercially available databases such as Oracle 8i and O_2, and 2) create the spatial data-indexing scheme directly using spatial data structures such as the R-tree or grid file. Oracle 8i Spatial has two models for spatial indexing, namely the object-relational model and the relational model. We built spatial indexes using both models from Oracle 8i. Due to the lack of support and functionality, we abandoned the spatial indexing method from O_2. For the direct spatial indexing method, we used the R-tree and modified grid file spatial data structures. The time required for building the spatial indexes for the four methods are 11.667 and 6.329 seconds for the R-tree and grid file, and approximately 660 and 480 seconds for the Oracle8i object-relational model and the Oracle8i relational models, respectively.

We tested the four indexing methods by performing various range searches on the test data. We tested using both primary range query (based on the bounding box of the objects only) and a secondary range query (based on the real data). Ten differently sized query windows (i.e. 1%, 4%, 9%, 16%, 25%, …, 100% of the data extent) were used. For each size of query window, we ran 50 random searches. According to the results, the R-tree outperforms the other three methods, and the grid file outperforms Oracle 8i Spatial. The Oracle 8i Spatial object-relational model outperforms the relational model.

For the query windows with size less than or equal to 49% of data extent, R-tree and the modified grid file take approximately same amount of time to perform a range search. The R-tree with order m=50 required an average of 0.497 seconds for a query window size of 49% on dataset test2 whereas the modified grid file required an average of 0.628 seconds under the same condition. As the query window increases, the modified grid file takes more time to perform a range search than the R-tree (see Figure 8.1). To perform the same range search with a query window size of 49% on dataset test2, Oracle object-relational and Oracle relational required approximately 220, 400 times more time, on average.

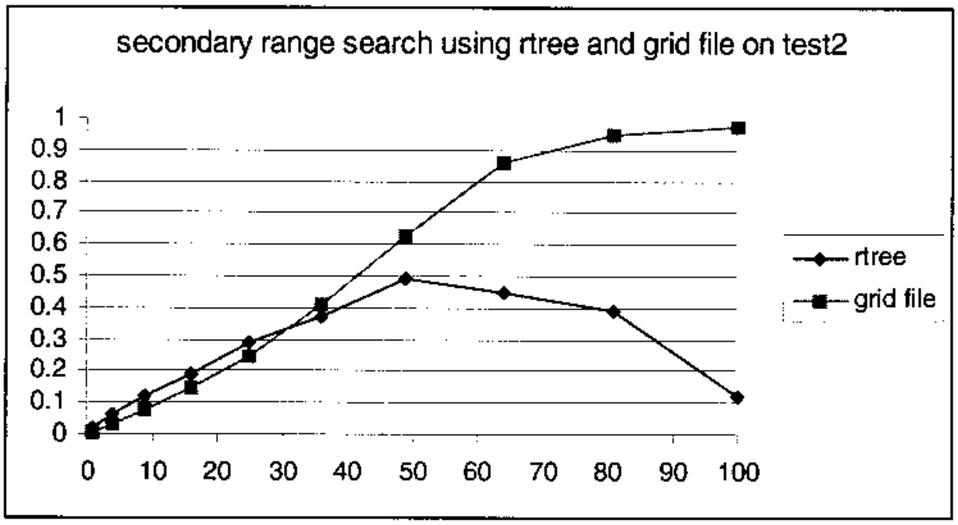


Figure 8.1 Secondary range search using the R-tree (m=50)

and the modified grid file on dataset test2.

From what we have experienced, the R-tree and grid file seem to be very good for our spatial datasets. Oracle8i Spatial data cartridge and O_2 Spatial are very difficult to set up and get running. There are quite a few spelling mistakes in the Oracle documentation that are misleading, and their technical support is limited. It took me three months to get any results. For O_2, it seems that the system is not stable, and the documentation and technical support is very limited.

This research has given rise to the following open questions:

1. Will the performance for Oracle 8i be better if we use OCI instead of ProC/C++?
2. How well does the same range search perform if we use a GIS database or other database system with spatial option, such as ArcInfo or Jasmine II?

3. There is a new spatial data structure called data replication [Kanth, 1998]; how well does this new method perform for indexing non-point spatial objects?
4. As the number of objects increases by orders of magnitude, will the modified grid file be faster than the R-tree?

REFERENCES

Abel, J. D. and Smith, L. J., "A data structure and algorithm based on the linear key for a rectangle retrieval problem", *Computer Vision, Graphics, and Image Processing,* vol. 24, 1983, pp.1-13.

Abel, J. D., "SIRO-DBMS: a database tool-kit for geographical information systems", *Int. J. Geographical Information Systems,* vol. 3, No. 2, 1989, pp.103-116.

Ardent software, "O_2 spatial user manual", The O_2 system: database solutions for object developers (CD), release 5.0.2, 1998.

Elmasri, R., and Navathe, B. S., *Fundamentals of Database systems*, Second Edition, Addison Wesley, 1994.

Eppstein, D., "Average case analysis of dynamic geometric optimization", *Computational Geometry*, vol. 6, 1996, pp. 45-68.

Gao, F., "Spatial Indexing of Large Volume Bathymetric Data Sets", UNB, Faculty of Computer Science, Technical report TR93-081, Dec. 1993.

Greene, D., "An Implementation and Performance Analysis of Spatial Data Access Methods", *Proceedings Fifth Internaltional Conderence on Data Engineering*, Los Angeles, USA, Feb. 6-10, 1989.

Gunther, O. and Gaede, V., "Oversize shelves: a storage management technique for large spatial data objects," *Int. J. Geographical Information Science,* vol.11, No.1, 1997, pp.5-32.

Guttman, A., "R-tree: a dynamic index structure for spatial searching", ACM, 1984, pp 47-57.

Hinrichs, K., Finke, U., Becker, L., "The binary-radix bucket-region-directory: a simple new directory for the grid file", Proceedings PANEL, 1992, pp. 460-472.

Kanth, K.V., Singh, A., "Efficient dynamic range searching using data replication", *Information processing letters 68*, 1998, pp. 97-105.

Knuth, D.E., "*The art of computer programming*, Vol. 1, *Foundamental algorithms*", Second edition, Addison-Wesley, Reading, MA, 1973

Kriegel, P.H., "Performance comparison of index structures for multikey retrieval", *Proceedings of the SIGMOD Conference*, Boston, June, 1984, pp.186-196.

Nickerson, G. B. and Gao, F., "Spatial indexing of large volume swath data sets", *Int. J. Geographical Information Systems*, vol. 12, 1998, in press.

Nievergelt, J., Hinterberger, H. and Sevcik, C. K., "The grid file: an adaptable, symmetric multikey file structure", *ACM Transactions on Database Systems*, vol. 9, 1984, pp.38-71.

Oracle Corp., "Oracle8 spatial data option user's guide and reference", release 8.0.3, Redwood Shores, CA, June 1997.

Oracle Corp., "Oracle8 spatial data option user's guide and reference", release 8.0.4, Redwood Shores, CA, November 1997.

Oracle Corp., "Oracle8i spatial data option user's guide and reference", release 8.1.5, Redwood Shores, CA, February 1999.

Samet, H., *The Design and Analysis of Spatial Data Structures*, Addison Wesley, Reading, MA, 1990.

Sinha, K. A., and Waugh, C. T., "Aspects of the implementation of the GEOVIEW design", *Int. J. Geographical Information Systems*, vol. 2, No. 2, 1988, pp.91-99.

Smith, T. R. and Gao, Peng, "Experimental performance evaluations on spatial access methods," *Proceedings of the 4th International Symposium on Spatial Data Handling*, Zurich, Switzerland, vol.2, July 23-27, 1990.

Appendix 0

Histograms of the object size and plots of the objects for two datasets

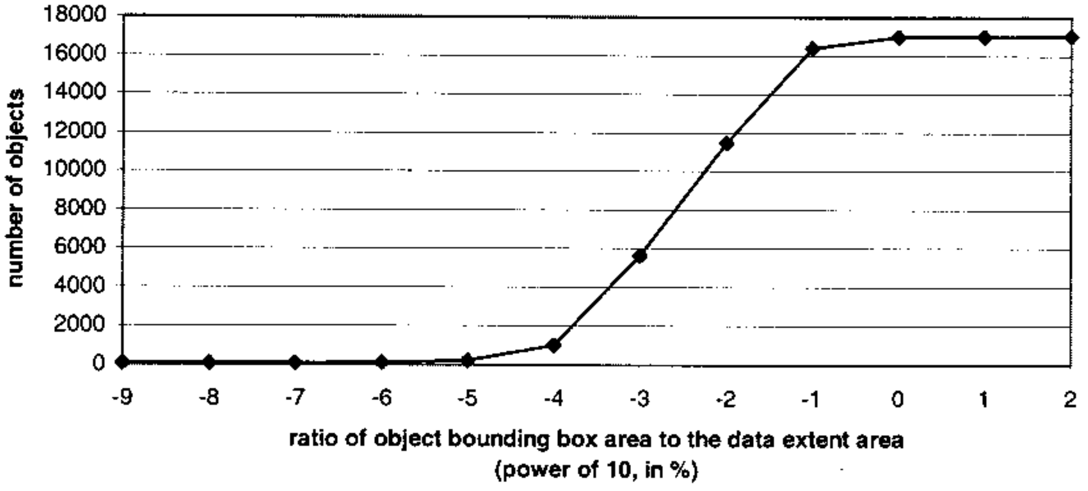


Figure A0.1. Cumulative histogram of the non-point object sizes for dataset test1 (nts21g15).

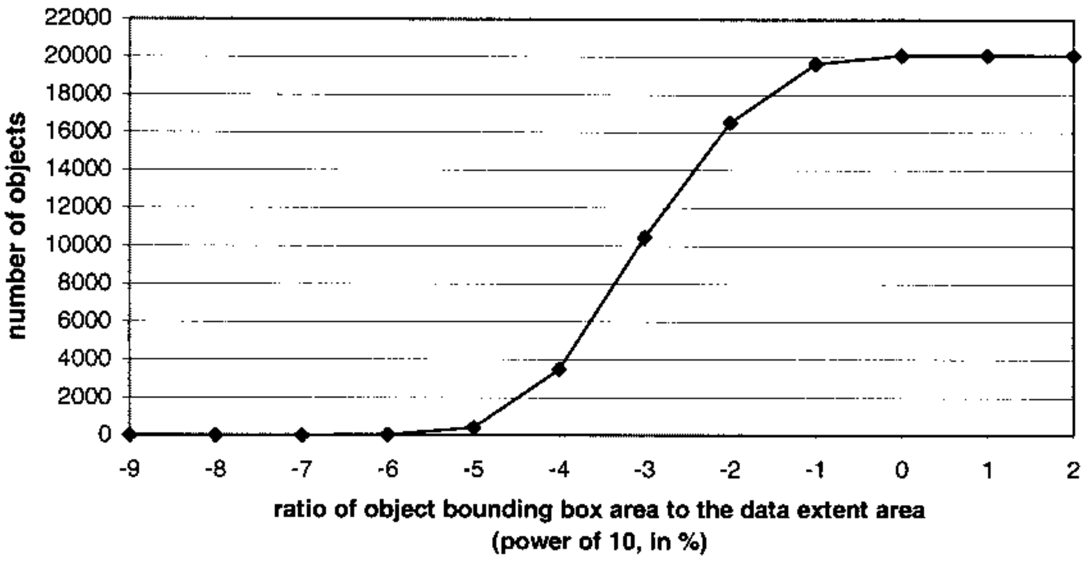


Figure A0.2. Cumulative histogram of the non-point object sizes for dataset test2 (nts21g).

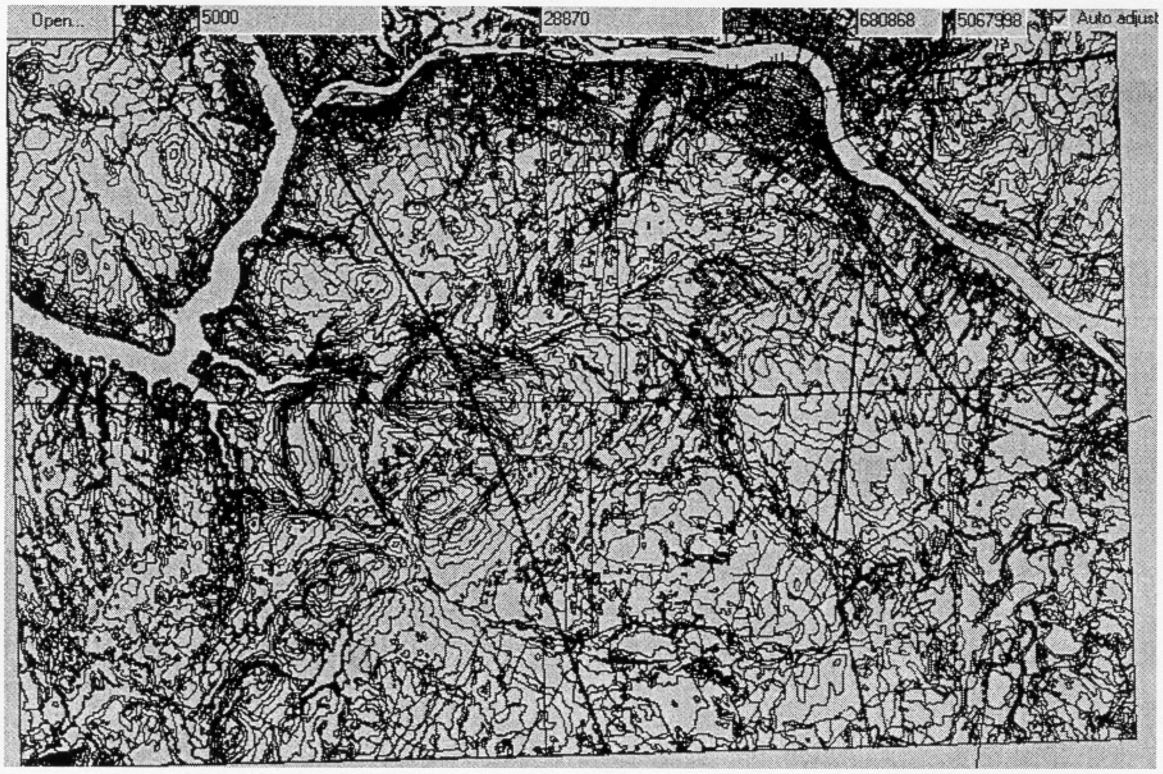


Figure A0.3. Plot of objects for dataset test1 (nts21g15).

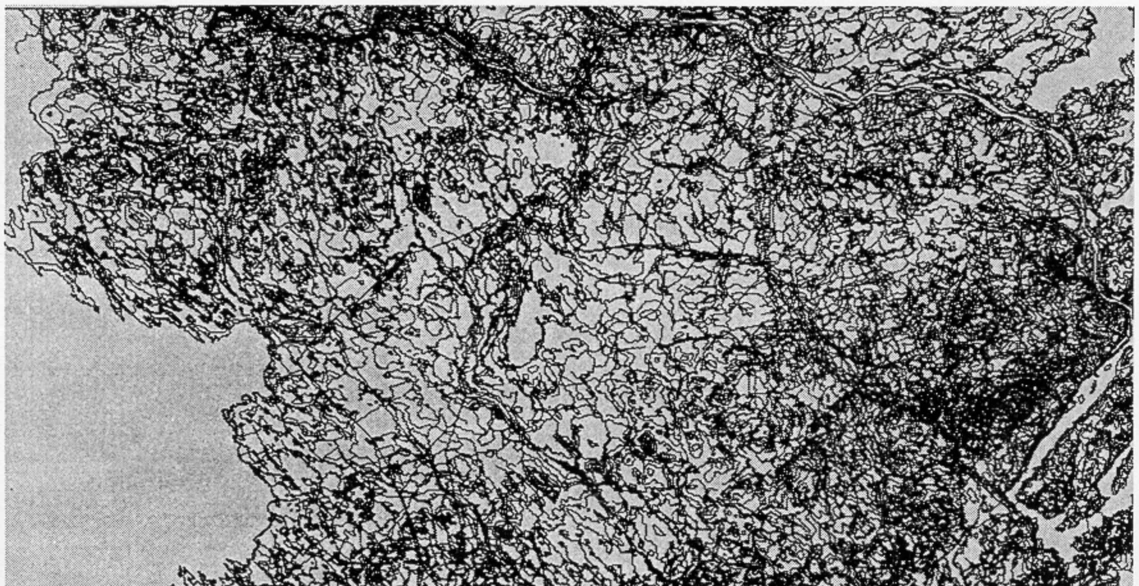


Figure A0.4. Plot of objects for dataset test2 (nts21g).

For the data in Figures A0.1 and A0.2, we bin objects into 12 categories according to their ratio (r = 100* object bounding box area / data extent area). We have the count of objects for each of the following bins: $0\% < r \leq 1*10^{-9}\%$, $1*10^{-9}\% < r \leq 1*10^{-8}\%$, $1*10^{-8}\% < r \leq 1*10^{-7}\%$, $1*10^{-7}\% < r \leq 1*10^{-6}\%$, $1*10^{-6}\% < r \leq 1*10^{-5}\%$, $1*10^{-5}\% < r \leq 1*10^{-4}\%$, $1*10^{-4}\% < r \leq 1*10^{-3}\%$, $1*10^{-3}\% < r \leq 1*10^{-2}\%$, $1*10^{-2}\% < r \leq 1*10^{-1}\%$, $1*10^{-1}\% < r \leq 1*10^{0}\%$, $1*10^{0}\% < r \leq 1*10^{1}\%$, $1*10^{1}\% < r \leq 1*10^{2}\%$. The count of object in each bin are plotted (cumulatively) against r, where r is labeled on the histogram as -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2. Each label corresponds to the power of ten of the upper bound for each of the 12 bins.

Appendix 1

Email Contacts Made with Ardent Support Stuff

***** Email number 1*****

-----Original Message-----
From: Liping Xie [mailto:g93c@unb.ca]
Sent: Monday, February 01, 1999 6:44 PM
To: o2support@ardentsoftware.com
Cc: bgn@unb.ca
Subject: O2 spatial

Hello,

I am a graduate student at the University of New Brunswick doing the research with spatial data under the direction of Dr. Nikerson, we have choosen to use O2 for spatial index testing. However, I have two questions about O2 spatial:

1) Where can I find a simple example for creating spatial index for few objects and performing range search on them.

2) Could you please give me some explanation for R-Quad tree which is used for the spatial indexing.

Thanks for the early reply
Liping Xie

Date: Tue, 2 Feb 1999 14:44:24 -0700
From: o2 Support <o2support@ardentsoftware.com>
To: 'Liping Xie' <g93c@unb.ca>
Subject: Case 102638: O2 spatial

Hello Liping,
We received your request for support. Unfortunately, our O2 expert analyst is out sick today but we'll make you obtain a quick response. If you need any additional information please contact us at 1-800-729-3553 or send e-mail to o2support@ardentsoftware.com.

Regards,
Ardent Software Inc
O2 Support

***** Email number 2*****

>> -----Original Message-----
>> From: Reynal Cocaign <reynal.cocaign@ardentsoftware.com>
>> To: 'Liping Xie' <g93c@unb.ca>
>> Date: Tuesday, February 16, 1999 6:31 PM
>> Subject: RE: O2 spatial
>>
>>
>> >Hi Liping,
>> >
>> >> Hello,
>> >>
>> >> I am a graduate student at the University of New Brunswick doing the research with spatial data under the direction of Dr. Nikerson, we have choosen to use O2 for spatial index testing. However, I have two questions
>> >> about O2 spatial:
>> >>
>> >> 1) Where can I find a simple example for creating spatial index for few objects and performing range search on them.
>> >>
>> >There are no samples delivered with the O2spatial index however please find enclosed a short example.
>> >
>> >> 2) Could you please give me some explanation for R-Quad tree which is used for the spatial indexing.
>> >>
>> >
>> >R-Quad Trees mix R-Tree representation for easy storage and QuadTree representation for fast retrieval.
>> >
>> >The QuadTree divides the space into Quadran each quadran being itself in Quadrans.
>> >The R-Tree (Rectangle Tree) is a set of rectangle (Which can be assimilated to the bounding box of some objects) that contains a set of spatial object.
>> >
>> >I hope these explanation and the sample found in attachment will help
>> >Regards,
>> >
>> >Reynal Cocaign

-----Original Message-----
From: Brad Nickerson <bgn@unb.ca>
To: o2support@ardentsoftware.com <o2support@ardentsoftware.com>; Reynal Cocaign <reynal.cocaign@ardentsoftware.com>
Cc: Liping Xie <g93c@unb.ca>
Date: Friday, February 19, 1999 9:15 AM
Subject: O2 spatial

>Reynal:
>
> Thanks for your response to Liping Xie, a graduate student working with me. We need more explanation about the R-Quadtree. I fully understand the R-tree and the many variations of the quadtree that are in the literature. Can you point us to a paper or patent that explains the R-Quadtree? Your explanation below is very vague, and does not explain how the R-tree and quadtree are "mixed". Do the R-tree and quadtree act independently? If you are indexing with a quadtree, how do you index linear and areal objects? Do you use a version of the PM-quadtree?

Thanks in advance for any light you can shed on the R-Quadtree.
>
>Regards, Brad Nickerson
>Dr. Brad Nickerson
>Professor and Director of the Information Technology Centre
>University of New Brunswick
>Faculty of Computer Science
>P.O. Box 4400
>Room GE-119, Gillin Hall, 540 Windsor Street
>Fredericton, N.B. E3B 5A3
>Canada
>phone: (506) 458-7278 fax: (506) 453-3566 E-mail: bgn@unb.ca
>
>
>On Wed, 17 Feb 1999, Liping Xie wrote:
>
>> 9:00am on Friday is ok with me, and the following is what I got from O2 support:
>>

***** Email number 3*****

> -----Original Message-----
> From: Liping Xie [mailto:g93c@unb.ca]
> Sent: Sunday, April 11, 1999 2:40 PM
> To: Reynal Cocaign
> Subject: RE: O2 spatial
>
>
> Hi Reynal,
>
> I was running the sample code you sent to me, it worked once, and I got some part of the results printed out. But when I tried to run it again one week ago, it gave the error 71256. Do you have any idea about this? Would you please tell me where I can find the reference for the error messages?
>
> Thanks in advance for your reply
>
> Liping
>

From: Reynal Cocaign <reynal.cocaign@ardentsoftware.com>
To: "'Liping Xie'" <g93c@unb.ca>
Subject: RE: O2 spatial
Date: Mon, 12 Apr 1999 12:15:12 -0600

Hi Liping,

Usually the error message would display the text associated to it. If not you can find the label of the message as follows:

```
teksun{reynal}:cd $O2HOME/message//disk3/o2/5.0/message
teksun{reynal}:grep 71256 *.eom_error.e:71256 Deletion of an indexed object is not allowed
```

Can you be a bit more precise about what you are doing and when it occurs while running the spatial index a second time. Do you select any special options from the menu?

Thanks,
Reynal

Reynal Cocaign

Customer Support Analyst
reynal.cocaign@ardentsoftware.com
Tel: 303-294-4768

The Art of Data Management... and All That Jazz
Register today for Ardent's 1999 Customer Conference
October 19 - 23, 1999
New Orleans, Louisiana, USA
http://www.ardentsoftware.com/conference

***** Email number 4*****
Tuesday, April 13, 1999 5:08PM

Hi Liping,

>
> Thank you for your quick response on my last message.
>
> I have a few questions regarding the compilation and execution of the sample program you sent me.
>
> 1. In the file o2spatial_sample.cf, two of the three lines below apparently are wrong since the makegen program complained about ImpBag and ImpList. I figured the colons are supposed to be equal signs so I changed it and everything seemed fine. Could you tell me if this was an appropriate thing to do.
>
> ImpFiles= Point2D.hxx GeoObject.hxx GeoPolygon.hxx GeoEllipse.hxx
> ImpBag: GeoObject
> ImpList: Point2D
Yes that's exactly what you have to do. In ealier versions of O2 the separator was colons. Having ported O2 to windows we had to change the separator to avoid conflict with drive specification, ie C:

> 2. In the source code, the macro
> #if defined(O2_NEED_TMPL_CODE) || defined(O2_NEED_TMPL_SRC)
> # include <o2template_CC.cc>
> #endif
>
> seemed to be true and as such, the file o2template_cc.cc is required. However, I did not find that file in the sample program you sent. I commented the line #include out just so that the compilation process passes. Is this file needed for a sucessful run of the sample? If yes, could you please sent me a copy.
>
This file is part of the standard O2 delivery when needed. You should find the file in O2HOME/include if it really is needed. This should only be the case on HPUX 10.
The macro should be:

```
#if defined(O2_NEED_TMPL_SRC)
#include <o2template_CC.cc>
#endif
```

>
> 3. During the building process, the compiler complained about unconfirm classes. Is this significant ?
>
Yes, once you are making modification on a class you have to confirm this modification otherwise you can not set a base, open a base in C++.
Confirming classes enables to create a class history internal to the O2 system so that type convertion can applied if necessary. You can add the directive +UseConfirmClasses in the configuration file.

> 4.I have sucessfully run the program several times before (or so it seemed !:-). Now, what I get is an error message saying that the spatial index could not be created along with the error code 71256. From the documentation, "71256 Delete unallowed for indexed objects". I feel like I am missing some steps in making the sample program or there is
> something I should have done before remake the program from scratch.
> Could you please suggest me as to what to do?
If you could track down at which line of code of the sample program this happens I might be able to help. I cannot reproduce it by running this sample twice.

> I would greatly appreciate any help I can get.
>
Reynal

***** Email number 5*****

> -----Original Message-----
> From: Liping Xie [mailto:g93c@unb.ca]
> Sent: Tuesday, April 13, 1999 9:05 PM
> To: Reynal Cocaign
> Subject: RE: O2 spatial
>
> Hi Reynal,
>
> I will try my best to described what I did and the error occur... Since I could not run the shell script on WinNT so I just created a batch file that basically does the same thing. I created the directory OutputDIr manually before running the batch file. I paste the result of the run below so you can see everything that went on during
> the process. I modified the o2spatial_sample.cf so that +UseComfirmClass is not commented out anymore and I got rid of the warning about unconfirmed class. I also put some more cout so that it is easier to trace the
> execution of the program. The source code is appended at the end of this message. I know this is long but it is the only way I can make things clearer.
>
> Thanks.
>
> Liping

Thursday, April 15, 1999

Hi Liping,

Thanks for sending all the details this helped me orientating my research. Actually the error message is not clear (I have create an improvement request for engineering so that they change it). What hapens is that we are checking for any existing spatial index and the error corresponds to O2E_IS_INDEXED as described in the spatial manual.

In your case this error should not happen since you have restarted by creating a new schema and a new base!
I would suggest to delete your base/schema once again and restart from scratch to see if you still have the same behavior. Please let me know if you have any changes!

Reynal

***** Email number 6*****

on April 15,1999 Reynal Cocaign wrote:

Hi Liping,

Thanks for sending all the details this helped me orientating my research. Actually the error message is not clear (I have create an improvement request for engineering so that they change it). What hapens is that we are checking for any existing spatial index and the error corresponds to O2E_IS_INDEXED as described in the spatial manual.

In your case this error should not happen since you have restarted by creating a new schema and a new base! I would suggest to delete your base/schema once again and restart from scratch to see if you still have the same behavior.

Please let me know if you have any changes!

Reynal

From: Liping Xie
Date: Monday, May 17, 1999
To: Reynal Cocaign
Cc: bgn@unb.ca
Subject: O2spatial

Reynal,

Ok, I have gone over the files (attached) many times but I could not see why the same executable would work one time and not another time. At first I thought it was because the file .Geo*list, .Geo*bag, .Geo*set changed during the compilation process. However, this seems unlikely because they only contain the string "ECHO is on." (13 byte-files). I believe the executable itself is ok. However, something must have changed during the compilation process since the same executable will not run anymore if I run _make_ again.

Attached are all the files relevant to the problem. Could you please take a look at them? Maybe something just snap out from your experience with O2 database.

The file Copy of exec.bat is basically a DOS version of the run_all shell script that you provided along with the sample code.

Thank you so much for your time.

Liping

***** Email number 7*****

From: Liping Xie
Date: Friday, June 25, 1999
To: Reynal Cocaign
Cc: bgn@unb.ca
Subject: O2spatial

Hi, Reynal

I am wondering if you got my email on May 17, I am still waiting for your reply. Besides, I would like to know if you have any running sample code under Windows NT with visual c++. If so, would you please send me a copy? then I can do some testing after reinstalling O2.

Thanks in advance

Liping

***** Email number 8*****

From: Reynal Cocaign
To: 'Liping Xie'
Subject: RE: O2 spatial
Date: Friday, June 25, 1999 4:14 PM

> I am wondering if you got my email on May 17, I am still waiting for your reply. Besides, I would like to know
> if you have any running sample code under Windows NT with visual c++. If so, would you please send me a
> copy? then I can do some testing after reinstalling O2.

I did not receive your email! Could you resend it please? Regarding the Windows NT/Visual C++, I guess that you wish to have a sample that that includes the spatial index. I do not have that. However we have example of VC projects which explains the settings required to be able to use together with O2. This can be found in the following directory:
O2HOME\samples\o2cplusplus\VCProjects.

There you have 2 different project (one console and one MFC project). In each case you have README files that gives you lots of information.

Reynal

***** Email number 9*****

From: Reynal Cocaign
To: 'Liping Xie'
Subject: RE: O2 spatial
Date: Friday, June 25, 1999 4:14 PM

Hi I just noticed that email you sent on May, 17th was attached to your email. I am looking into it.

Reynal

Appendix 2

The code for Overlap(), Contained() and twoline_intersect()

```
float A[3][3];

struct P
{
  float x,y;
};

void Inverse()
{
  float det = A[2][2]*A[1][1] - A[1][2]*A[2][1];
  float save = A[1][1];
  A[1][1] = A[2][2] / det;
  A[1][2] = -A[1][2]/det;
  A[2][1] = -A[2][1]/det;
  A[2][2] = save/det;
}

// determine if two line intersect
// uses parametric line segment intersection (from spatial data
// structure class notes)
int twoline_intersect(p1, p2, q1, q2)
struct P p1;
struct P p2;
struct P q1;
struct P q2;
{
  struct P q[3];
  struct P p[3];
  struct P q_p[3];

  float u;
  float v;

  p[1].x = p1.x;
  p[1].y = p1.y;
  p[2].x = p2.x;
  p[2].y = p2.y;
  q[1].x = q1.x;
  q[1].y = q1.y;
  q[2].x = q2.x;
  q[2].y = q2.y;

  A[1][1] = p[2].x-p[1].x;
  A[1][2] = q[1].x-q[2].x;
  A[2][1] = p[2].y-p[1].y;
  A[2][2] = q[1].y-q[2].y;

  q_p[1].x = q[1].x-p[1].x;
  q_p[1].y = q[1].y-p[1].y;
  Inverse();
  u = A[1][1]*q_p[1].x + A[1][2]*q_p[1].y;
  v = A[2][1]*q_p[1].x + A[2][2]*q_p[1].y;

  if (u>=0 && u<=1 && v>=0 && v<=1 )
    return 1;
  else
    return 0;
}


/*-----------------------------------------------------------------------
| Decide whether two rectangles overlap.[Guttman,1984]
---------------------------------------------------------------------*/
int Overlap(Rect *r, Rect *s)
{
  int i, j;

  for (i=0; i<NUMDIMS; i++) // NUMDIMS=2 for two-dimensional data
  {
    j = i + NUMDIMS;  /* index for high sides */
    if (r->boundary[i] > s->boundary[j] || s->boundary[i] > r-
>boundary[j])
      return FALSE;
  }
  return TRUE;
}

/*-----------------------------------------------------------------------
| Decide whether rectangle r is contained in rectangle s.[Guttman,1984]
---------------------------------------------------------------------*/
int Contained(Rect *r, Rect *s)
{
  int i, j, result;

  /* undefined rect is contained in any other */
  if (Undefined(r))
    return TRUE;

  /* no rect (except an undefined one) is contained in an undef rect */
  if (Undefined(s))
    return FALSE;

  result = TRUE;
  for (i = 0; i < NUMDIMS; i++)
  {
    j = i + NUMDIMS;  /* index for high sides */
    result = result
      && r->boundary[i] >= s->boundary[i]
      && r->boundary[j] <= s->boundary[j];
  }
  return result;
}
```

Appendix 3

Space cost for Oracle8i relational and object-relational model

Relational model

Dataset test1:

Data table: layer3_sdogeom

Space per record: number (4bytes) x 8 = 32 bytes
Number of records: 343827
Space cost: 343827 x 32 = 11002464 bytes = 11Mbytes

Index table: layer3_sdoindex

Space per record: number (4bytes) + raw (255bytes) = 259 bytes
Number of records: 1,000,000
Space cost: 1,000,000 x 259 = 259 Mbytes

Data and index: 11Mbytes + 259 Mbytes = **270 Mbytes**

Dataset test2:

Data table: layer2_sdogeom

Space per record: number (4bytes) x 8 = 32 bytes
Number of records: 523613
Space cost: 523613 x 32 = 16755616 bytes = 16.8 Mbytes

Index table: layer2_sdoindex

space per record: number (4bytes) + raw (255bytes) = 259 bytes
number of records: 1,000,000
space cost: 1,000,000 x 259 = 259 Mbytes

Data and index: 16.8Mbytes + 259 Mbytes = **275.8 Mbytes**

Note: The number of records in the two datasets are approximate since I could not regenerate the index table using the relational model (see chapter 7). The number of records are what I recall when I first generated the index tables.

Object-relational model

Dataset test1:

Data table: layer_17kk

varchar2 (32bytes)
sdo_geomtry
number (4bytes) x 2 = 8 bytes
sdo_point_type
number (4bytes) x 3 =12 bytes
sdo_elem_info_array
number (4bytes) x 200 = 800 bytes
sdo_ordinate_array
number (4bytes) x 200 = 800 bytes
space per record: 32 + 8 + 12 + 800 + 800 = 1652 bytes
number of records: 16998
space cost: 16998 x 1652 = 28080696 bytes = 28.1 Mbytes

Index table: layer_17kk_index_hl5n4$

space per record: raw (11bytes) x 4 + rawid (256bytes)=300bytes
number of records: 69027
space cost: 69027 x 300 = 20708100bytes = 20.7 Mbytes

Data and index: 28.1Mbytes + 20.7 Mbytes = **48.8 Mbytes**

Dataset test2:

Data table: layer_20k

varchar2 (32bytes)
sdo_geomtry
number (4bytes) x 2 = 8 bytes
sdo_point_type
number (4bytes) x 3 =12 bytes
sdo_elem_info_array
number (4bytes) x 200 = 800 bytes
sdo_ordinate_array
number (4bytes) x 200 = 800 bytes
space per record: 32 + 8 + 12 + 800 + 800 = 1652 bytes
number of records: 20104
space cost: 20104 x 1652 = 33211808 bytes = 33.2 Mbytes

Index table: layer_20kk_index_hl7n4$

space per record: raw (12bytes) x 4 + rawid (256bytes)=304bytes
number of records: 97106
space cost: 97106 x 304 = 29520224bytes = 29.5 Mbytes

Data and index: 33.2Mbytes + 29.5 Mbytes = **62.7 Mbytes**

VITA

Candidate's full name:

Liping Xie

University attended:

9/1986 – 7/1990
Department of Petroleum Geology, Jianghan Petroleum Institute, Shashi, Hubei, China.
B.Sc. received in July 1990.

Publications:

N/A

Conference Presentations:

"Spatial data indexing for relational databases", The Atlantic institute student research conference (8th annual). 20th-21st February 1998. University of New Bruncswick, Fredericton.

"A comparison of non-point spatial data indexing methodologies", The Atlantic institute student research conference (9th annual). 8th – 9th June 1999. Laval University, Quebec City.