SPATIAL INDEXING OF LARGE VOLUME BATHYMETRIC DATA SETS

by

Feng Gao

TR93-081 December 1993

This is an unaltered version of the author's
M.Sc.(CS) Thesis

Faculty of Computer Science
University of New Brunswick
P.O. Box 4400
Fredericton, N.B.
Canada E3B 5A3

Phone: (506) 453-4566
Fax: (506) 453-3566

SPATIAL INDEXING OF LARGE VOLUME BATHYMETRIC DATA SETS

by

Feng Gao

M.Sc.E., Tsinghua University, Beijing, China, 1987

B.Sc.E., Beijing College of Architecture and Civil Engineering, Beijing, China, 1984

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science (Computer Science)

in the Faculty

of

Computer Science

This thesis is accepted

....

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

December 1993

© Feng Gao, 1993

SPATIAL INDEXING OF LARGE VOLUME BATHYMETRIC DATA SETS

by

Feng Gao

M.Sc.E., Tsinghua University, Beijing, China, 1987

B.Sc.E., Beijing College of Architecture and Civil Engineering, Beijing, China, 1984

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science (Computer Science)

in the Faculty

of

Computer Science

This thesis is accepted

....

Dean of Graduate Studies and Research

THE UNIVERSITY OF NEW BRUNSWICK

December 1993

© Feng Gao, 1993

ABSTRACT

Three spatial index structures based on Morton code sequence and R-tree indices were designed and implemented. Indices for the Hydrographic Data Cleaning System (HDCS) were built with Morton sequence and R-tree structures directly implemented in the C language, and with Morton sequence structures managed by the INGRES relational database management system. A modified deletion algorithm for the R-tree index structure was developed.

To test the software and to evaluate the implementation of the structures, experiments were carried out with a bathymetric data set surveyed in Conception Bay, Canada. Spanning four days, the survey includes 46 lines consisting of 54,192 profiles each of which contains 32 sounding points. The total amount of data is 128.9 megabytes. It is shown that R-tree indexing is superior to Morton sequence indexing by providing five to nine times faster range search speed, occupying less disk space, and better supporting the range deletion operation. Range deletion with the modified algorithm was 11.7 times faster, on average, than Guttman's original deletion algorithm. The experiment also showed that, on average, the INGRES RDBMS build time was 31.7 times slower than building the Morton sequence implemented in C, search time is 176 times slower than searching the Morton sequence, and occupied 4.4 times more space.

The three spatial index structures, Morton sequence, R-tree and INGRES Morton sequence, required additional 1.9%, 1.7% and 7.9% of the original file storage for the HDCS data, respectively.

TABLE OF CONTENTS

ABSTRACT i
TABLE OF CONTENTS ii
LIST OF FIGURES v
LIST OF TABLES vii
ACKNOWLEDGMENTS viii

1 INTRODUCTION 1
1.1 Range Queries and Spatial Indexing 1
1.1.1 Spatial Data 1
1.1.2 Range Query 2
1.2 Data Structures for Spatial Indexing 4
1.2.1 Introduction 4
1.2.2 Grid File Structures 5
1.2.3 Tree Indexing Structures 6
1.2.3.1 Quad Trees 7
1.2.3.2 k-d Trees 7
1.2.3.3 Range Trees 8
1.2.3.4 R-Trees 9
1.2.4 Linear Indexing Structures 9
1.2.5 Regular Tessellation 11
1.3 Thesis Outline 13

2 THE HDCS GENERAL STRUCTURES 16
2.1 Ocean Mapping Project 16
2.2 HDCS Data and File Structures 16
2.2.1 HDCS File Organization 18
2.2.2 HDCS File Structures 19
2.3 Spatial Indexing in HDCS 21
2.4 Spatial Searches in HDCS 24

3 MORTON CODE INDEXING 26
3.1 Morton Codes 26
3.1.1 Encoding Morton Codes 26
3.1.2 Decoding Morton Codes 27
3.2 Morton Sequence of Points 28
3.2.1 Morton Sequence 28
3.2.2 Characters of Morton Sequences 29
3.2.3 Morton Sequence of Data Points 32
3.3 Range Search Using Morton Sequence 33
3.3.1 Linear Range Search Algorithm on Morton Sequence 34

3.3.2 Over-search Problems.....35
3.3.3 Modified Range Search Algorithm.....37
3.4 Morton Sequence Indexing of HDCS Profiles.....38
3.4.1 HDCS Spatial Indexing.....39
3.4.2 Spatial Resolution of 64-bit Morton Code.....39
3.4.3 Encoding and Decoding of Morton Codes in HDCS.....40
3.4.4 HDCS Index File Structures.....41
3.4.5 Creating Spatial Index Files.....42
3.4.6 Range Searches Using the Index.....44

4 RELATIONAL DATABASE MORTON CODE INDEXING.....47
4.1 Introduction.....47
4.2 Relational Tables for Morton Code Indexing.....47
4.3 Range Search Using the INGRES Relational Database.....50
4.4 Space Requirement Calculation.....51

5 R-TREE SPATIAL INDEXING.....53
5.1 R-tree Structure and Its Variations.....53
5.2 R-tree File Structures for HDCS.....57
5.3 Building R-tree Indices.....63
5.4 Range Search Using R-tree Indices.....64
5.5 Linking Deleted Records in R-trees.....65
5.6 Deleting Profiles From R-tree Indices.....65
5.7 Modified Deletion Algorithm.....66

6 COMPARISON OF THE INDEXING METHODS.....70
6.1 Experimental Description.....70
6.2 Times for Building Morton Code and R-tree Spatial Indices.....75
6.3 Times for Searching Morton Code and R-tree Spatial Indices.....76
6.4 Times For Deleting Profiles from R-tree Spatial Indices.....80
6.5 Times for Building and Searching INGRES Morton Code Indices.....82
6.6 Space Requirement of the Three Spatial Indexing Approaches.....83
6.7 Advantages and Disadvantages of the Three Indexing Approaches.....86

7 SUMMARY AND CONCLUSION.....88

REFERENCES.....91

APPENDIX A: STRUCTURES ACCOMMODATING SEARCH RESULTS 94

APPENDIX B: ALGORITHMS FOR SEARCHING MORTON SEQUENCE INDICES IN HDCS 99

APPENDIX C: AN EXAMPLE OF USING DYNAMIC SQL TO RETRIEVE MORTON CODES FROM AN INGRES TABLE 104

APPENDIX D: THE MODIFIED R-TREE DELETION ALGORITHMS 109

APPENDIX E: A DETAILED LIST OF HDCS DATA USED FOR THE EXPERIMENT 112

VITA 114

LIST OF FIGURES

Figure 1.1 Range query types....3
Figure 1.2 Grid file representation of a data space....6
Figure 1.3 Liner ordering of 2D space....10
Figure 1.4 Space decomposition using three regular tessellations....12
Figure 1.5 Tessellations at different resolutions and their Euclidean plane tiles....13
Figure 2.1 Hydrographic data cleaning process....17
Figure 2.2 A line and its soundings....18
Figure 2.3 Hierarchical organization of ocean mapping data in HDCS....19
Figure 2.4 Index and data file general structure in HDCS....20
Figure 2.5 Position/Depth file structure....21
Figure 2.6 Hierarchical spatial index structures....23
Figure 2.7 Three step access for inside profile testing....24
Figure 3.1 Points in binary coordinates and their Morton codes....27
Figure 3.2 Calculating Morton codes by interleaving bits....27
Figure 3.3 Two Morton sequences for 2D points....29
Figure 3.4 Quad tree related decomposition and recursion of a Morton sequence....30
Figure 3.5 Numbering scheme in base 4 digits....31
Figure 3.6 The walking pattern of a Morton sequence....31
Figure 3.7 Data points and the corresponding Morton sequence....33
Figure 3.8 The Morton sequence and a query window....34
Figure 3.9 A range search on a Morton sequence and its over-search problem....36
Figure 3.10 Relation of Morton sequence and query window....38
Figure 3.11 Possible cases Morton sequence touching or crossing QW edges....45
Figure 3.12 A range search on a Morton sequence....46
Figure 5.1 An R-tree and the corresponding MBRs....54
Figure 5.2 The R-tree and its coverage and overlay problem....55
Figure 5.3 An R^{+}-tree and the corresponding object MBRs....57
Figure 5.4 Directory spatial index files....61
Figure 5.5 A simple R-tree for profile MBRs....67
Figure 6.1 Surveyed area in Conception Bay off the north-east coast of Newfoundland....71
Figure 6.2 Algorithm for Morton code and R-tree index build and search experiments....73
Figure 6.3 Algorithm for range deletion experiment....74
Figure 6.4 Times for building R-tree indices for HDCS using different branch factors....76
Figure 6.5 Times for searching Morton code and R-tree indices in HDCS....79
Figure 6.6 Average time used in the deletion of different ranges....81

Figure 6.7 Times used in searching INGRES Morton code indices. 82
Figure 6.8 Space requirement for Morton code, R-tree and INGRES MC indices. 85
Figure 6.9 Space requirement for Morton code and R-tree indices. 85

LIST OF TABLES

Table 1.1 Comparison of linear orderings of 2D space....11
Table 3.1 A type definition for a 64-bit Morton code. 39
Table 3.2 Directory spatial index data file header structure....42
Table 3.3 Directory spatial index data file record structure....43
Table 3.4 Profile spatial index data file header structure. 43
Table 3.5 Profile spatial index data file record structure. 44
Table 4.1 INGRES table for projects. 48
Table 4.2 INGRES table for directories....49
Table 4.3 INGRES Table for Profiles....49
Table 5.1 Directory spatial index file (File1) header structure....59
Table 5.2 Directory spatial index file (File1) data structure for R-tree nodes 60
Table 5.3 Directory spatial index file (File2) header structure....60
Table 5.4 Directory spatial index file (File2) data structure for R-tree leaf nodes of File1....60
Table 5.5 Header record structure for the profile spatial index file....62
Table 5.6 Structure of one record of the profile spatial index file....63
Table 6.1 Query windows and their coverage....72
Table 6.2 Profile ranges to delete for profile deletion experiment....74
Table 6.3 Times (in seconds) for building Morton code and R-tree indices. 75
Table 6.4 Times (in seconds) for searching Morton code indices. 77
Table 6.5 Times (in seconds) for searching R-tree (M=4) indices. 77
Table 6.6 Times (in seconds) for searching R-tree (M=5) indices. 78
Table 6.7 Times (in seconds) for searching R-tree (M=6) indices. 78
Table 6.8 Times (in seconds) for searching R-tree (M=7) indices. 79
Table 6.9 Times (in seconds) for deleting R-tree indices using Guttman's algorithm. 80
Table 6.10 Times (in seconds) for deleting R-tree indices using modified algorithm. 81
Table 6.11 Times (in seconds) for searching INGRES Morton code indices. 82
Table 6.12 Number of kilobytes required by the three approaches. 84
Table 7.1 Comparison of times (in seconds) for building and searching index files and the index file's space requirements (in KB)....89

ACKNOWLEDGMENTS

I wish to express sincere thanks to my supervisor, Dr. Bradford G. Nickerson, for his suggestion of this subject and consistent guidance through each phase of this research. His valuable advice and insightful thoughts have greatly contributed in shaping this research work. I would also like to thank Dr. Lev Goldfarb, Dr. Joseph. D. Horton, and Dr. Y. C. Lee, who read the thesis drafts and provided valuable comments and suggestions.

Thanks are also due to the many individuals who contributed toward this thesis, especially Mr. Leonard Slipp for his assistance in acquiring the HDCS data, and Ms. Tracy Gulliver for her help in solving my problems with the INGRES database management system.

Great appreciation should also be extended to Ocean Mapping Group for funding my research work.

Especially, I would like to express my deep indebtedness to Dr. C. Ann Cameron for her encouragement and help through the days, and maternally turning her house into my home after I arrived in Fredericton.

CHAPTER ONE

INTRODUCTION

1.1 Range Queries and Spatial Indexing

1.1.1 Spatial Data

Computer processing of spatial data has grown substantially in recent years as the power of computers has increased. We will continue to demand more spatial data processing from computers since visual interpretation of our surroundings is so natural for humans. The rapid advance in computing technology also provides the ability to handle large volumes of spatial data. This increasing ability makes it possible to organize, store, retrieve, generate and distribute spatial data more efficiently.

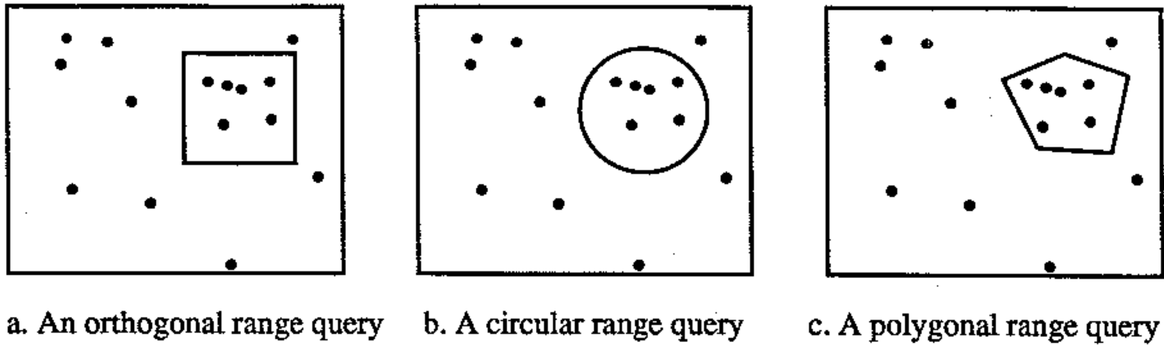
Data capturing techniques are becoming more and more sophisticated, and spatial data with higher resolutions are being acquired. The rapid advance in technology yields tremendous amounts of spatial data. Generally, a fourfold increase in data volume results by doubling the spatial resolution [Goodchild, 1989].

Spatial data are 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.

Five primitive types can be used to represent spatial objects: a point, a line, a polygon, a surface and a volume. Points can be considered as a basic type, since, no matter how complicated an object is, it can be ultimately decomposed into points. Mathematical methods can also be employed to approximate any object using point data. A complicated object can also be represented by a representative point such as the centroid of the object or the corners of its minimal bounding rectangle. Because of this, spatial operations for points can also be used for the objects of the other four types.

1.1.2 Range Query

The range query is a frequent operation performed on spatial objects. It can be described as retrieving or counting a collection of spatial data. In processing a query, two types of activities are involved. Firstly, given a set of spatial data, a search is required. The search, containing a sequence of comparisons, leads to a subset of the spatial data. Researchers in this area are trying to minimize search time by employing different search algorithms and different underlying data structures. Secondly, retrieval is involved to assemble the query results. For example, a typical one-dimensional range search requires retrieval of all the points within the range $[x_1, x_2]$ among N points ordered along the x-axis. The two activities can be described as using binary search to find x_1, and sequentially collecting each point along the x-axis in increasing x order, until a point whose value is greater or equal to x_2 is reached.



a. An orthogonal range query   b. A circular range query   c. A polygonal range query

Figure 1.1 Range query types.

A query has standard geometric shape [Preparata, F. P., and Shamos, M. I, 1985]. In a two-dimensional space, three forms of range queries can be specified: *orthogonal*, *circular*, and *polygonal range query* (Figure 1.1). The first of the three is the simplest. It only needs comparison in the axis directions.

Overhead is always a problem in range search, since spatial data must be physically stored on a medium such as a disc. Reading data from the medium or writing data to it involves relative movement of the disc heads. By the nature of this disc movement, spatial data can only be stored on the disc as a sequence of binary digital numbers. Two locationally close data, therefore, 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.

There are two resolutions to this problem. At a lower level, spatially close data should be kept in a physically adjacent location, typically in a disc page; at a logical level, a spatial indexing mechanism can be used. Usually those two methods are used jointly to reduce the overhead in performing a range query.

1.2 Data Structures for Spatial Indexing

1.2.1 Introduction

Carrying locational and topological meaning, spatial data are far more complicated than non-spatial data. These implicit meanings arise from human interpretation of the spatial data. It is realized that traditional linear indexing methods are not well suited to the non-zero size objects such as lines and polygons [Preparata and Shamos 1985; Samet 1990a; Abel and Mark 1990]. Because of this, we cannot simply handle the spatial data in the way we process non-spatial data. For example, we cannot simply sort the spatial data and still keep its property in the space [Wood, 1993].

In order to speed up range searches in geographic information systems and digital mapping, advanced indexing mechanisms for spatial data are used. Numerous hierarchical spatial data structures have been introduced in indexing multi-dimensional objects. Generally, spatial data structures are more complicated than non-spatial data structures. This is required by the applications, e.g., range searches, on the spatial data, since there is no total ordering for them [Wood, 1993]. The field of spatial data structures is a very active research area. New proficient data structures are continuously being created; old structures are being improved so that they can handle more sophisticated spatial data at a higher efficiency.

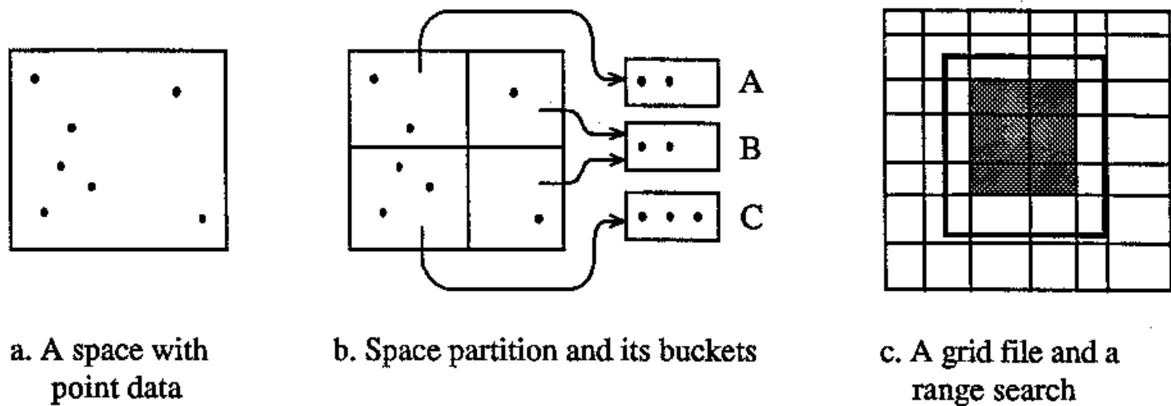
Spatial data can be represented in two approaches: *object-space representation* and *image-space representation*. The former is more object-oriented than the latter, and both representations tend to approximate each other [Samet and Webber 1988]. The first approach is based on the observation that complex objects consist of sub-objects which, in

turn, are complex objects of other relatively simple sub-objects. Hierarchical structures can be used to represent objects. An R-tree is an example of this method (see section 1.2.3.4). A typical image-space representation in two-dimensional space is a quadtree. Regions close to each other are organized in the tree as one object. Both object-space and image-space representations are similar to each other in some extent.

Spatial structures, according to how they organize the data, fall into the following four categories: grid files, tree structures, linear orderings, and tessellations. These structures and their usage in range searches are viewed briefly with emphasis on the tree structures and linear orderings.

1.2.2 Grid File Structures

Grid file structures were first introduced by Nievergelt [1984]. The idea behind grid file structures is to organize data space. Figure 1.2a is a space housing point data. It is separated by *grid lines* into *grid blocks*. Usually blocks are rectangles, having maximum m points in each. With data being held in a physical unit called a *bucket*, a block maintains the address of the unit. One bucket may store points of more than one block. Updating a grid file involves splitting and merging blocks and buckets as data are added into or deleted from the space.



a. A space with point data

b. Space partition and its buckets

c. A grid file and a range search

Figure 1.2 Grid file representation of a data space.

A range search spans a set of grid blocks. The blocks intersecting the range borders, or *bordering grid blocks*, may have data outside the query, while all the *non-bordering grid blocks* (the shaded grid blocks in Figure 1.2c) accommodate within-the-range points.

An adaptive grid file can also be used to find line intersections and to solve point-in-polygon problems [Franklin, 1984].

1.2.3 Tree Indexing Structures

Another commonly used approach is building spatial indices using hierarchical tree structures. The major advantage of these structures is they can rapidly focus on the data subsets of interest. This ability "results in an efficient representation and in improved execution times [Samet, 1990a]."

The most well-known tree structures are quad trees, k-d trees, segment trees, range trees, and R-trees.

1.2.3.1 Quad Trees

Constructing a quad tree for point data requires recursively decomposing a data space into four disjoint congruent square regions, or quadrants, until each quadrant has less than *m* points. A tree can be built accordingly with its leaf nodes accommodating addresses to the point data and its non-leaf ones pointing to the nodes of a lower level [Samet 1990a, Wood 1993].

As pointed out by Wood [1993], a range query on a quad tree, starting at the root, may be preserved as long as it is completely within a quadrant, or eventually split into *semi-range* query, *quarterplane* query, *slab* query, or *half plane* query. If we have n points, therefore, a range search reporting r points takes $O(r + \sqrt{n})$ time, in the worst case, on a quad tree of minimum height $\lceil \log_4 n \rceil$.

1.2.3.2 *k*-d Trees

Samet [1990a] identified the major disadvantages of the *k*-dimensional point quad trees: comparing *k* dimensional keys rather than only one at each node in a search, requiring a large amount of space for each node, and wasting space at non-leaf nodes due to the NIL pointers. Those deficiencies, however, can be reduced by using a *k*-d tree.

A *k*-d tree can be built by recursively subdividing the space in only one dimension. For example, we create a 2-d tree in two dimensions by alternatively using the x-coordinate and y-coordinate as discriminators at different levels of the tree. The space for a *k*-d tree is somewhat less than for a point quadtree since the level of a tree implies its discriminator.

Range searches are performed similar to the way we conduct a binary search. If a node is outside a query, we can avoid searching the branch of the tree rooted at this node. Only when the node is in the range, further searches for both ranges are needed. The time required for a worst case range search is $O(k\, n^{1-1/k})$ with a k-d tree of n points [Samet 1990a].

1.2.3.3 Range Trees

Range trees are designed to retrieve all the points falling within a given range. Comparing to the quadtree and the k-d tree, the range tree is "an asymptotically faster search structure [Samet, 1990a]." As a trade off, a significantly larger volume of storage is required since duplicated data are used in this tree-containing-tree structure.

In constructing a two-dimensional range tree, for example, we first sort all the points in one dimension, say x, and store those points at the leaf nodes of a binary tree T. Attached at each non-leaf node N is another binary tree T' with it's external nodes housing all the points in the subtree of T rooted at N. This time the tree T' is sorted in the other dimension y.

A range search for $([x_{min}, x_{max}], [y_{min}, y_{max}])$ starts with marking the paths from the root T to x_{min} and x_{max}. Ascending along the path to x_{min} from the closest common non-leaf node Q of the two paths, if a non-leaf node N's left son is also in the path, search its right son's binary tree T' sorted on y. On the other path to x_{max} from Q, if N's right son is also in the path, search its left son's binary tree T'.

A k-dimensional range tree of n points requires $O(n \log_2^{k-1} n)$ storage and a range search reporting r points takes $O(\log_2^k n + r)$ [Samet, 1990a].

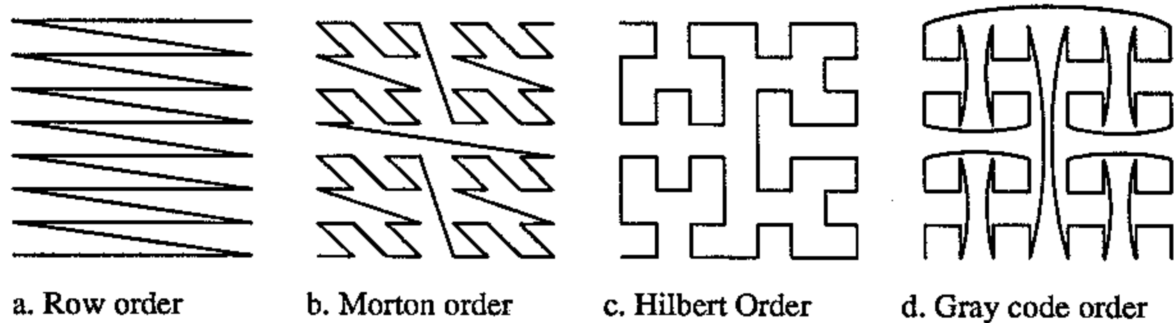
1.2.3.4 R-Trees

Guttman [1984] first introduced R-tree structures. The fundamental idea of an R-tree is for each object space to generate a minimal bounding rectangle (MBR) to cover the object. Those MBRs, according to their locations in the space, are divided into groups which, in turn, are bounded by larger MBRs hierarchically. The number of MBRs in a group is the number of entries in a node of the tree. Each entry in a leaf node stores the identification and the MBR of an object; an entry in non-leaf nodes contains the pointer to its child and the rectangle tightly bounding all the MBRs of the child. In a range search, by checking MBRs at the low level of the tree, only data near the search range are examined and irrelevant regions can be eliminated.

1.2.4 Linear Indexing Structures

Spatial indexing and search algorithms can be simplified by reducing multi-dimensional problems to one dimension problems, which in turn makes it possible to benefit from the existing data structures and algorithms in the one-dimensional world. One of the main approaches in spatial indexing is mapping an m-dimensional image into a one-dimensional line, known as the *space-filling curve* [Witten and Wyvill 1983]. The orderings are a list of single valued keys derived from the coordinates of the objects. Among many such curves, the Morton sequence, or Morton order [Samet 1990a, Samet 1990b, Peuquet 1984], is one of the conversions widely used in geographic information

systems. Other well-known linear orderings are row order [Samet 1990a], Hilbert order [Samet 1990a, Peuquet 1984], and gray code order [Samet 1990a] (see Figure 1.3).



a. Row order b. Morton order c. Hilbert Order d. Gray code order

Figure 1.3 Liner ordering of 2D space.

Identified by Abel and Mark [1990] are three properties of two-dimensional spatial orderings of a cellular model of space. They also apply to points at the resolution of the space.

(a) An ordering is *continuous* if, and only if, every cell pair with consecutive keys has four neighbours.

(b) An ordering is *quadrant-recursive* if "the cells in any valid quadtree subquadrant of the matrix are assigned a set of consecutive integers as keys."

(c) An ordering is *monotonic* if, and only if, for a fixed coordinate x or y, the key varies monotonicly with the other coordinate y or x, respectively.

Table 1.1 Comparison of linear orderings of 2D space.

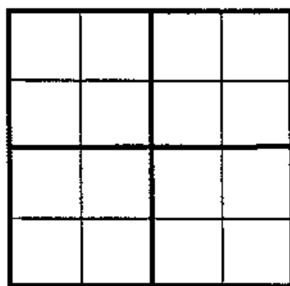
Ordering	Properties	Applications
Row Order	Monotonic.	Raster data in remote sensing and image processing.
Morton Order	Monotonic and quadrant-recursive	Spatial indexing in geographic information systems
Hilbert Order	Quadrant-recursive and continuous	Run-encoding tests and colour-space processing
Gray Code Order	Quadrant-recursive	Spatial data handling.

Range searches on the orderings sometimes involve coding and decoding the keys. Take a simple range search on a Morton sequence for example. Morton codes for the lower-left (MC_{LL}) and upper-right (MC_{UR}) corners of a given query window need to be calculated. All the points of their Morton codes smaller than MC_{LL} or greater than MC_{UR} are definitely outside the query window. For the Morton codes between MC_{LL} and MC_{UR}, coordinates have to be derived from the code so that we can check each point against the query. In a general picture, the basic range algorithm employs binary searches to find the range of MC_{LL} and MC_{UR} on the order, and uses a linear search to retrieve the points within the window.

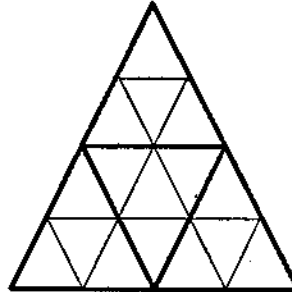
1.2.5 Regular Tessellation

Regular tessellations are space decomposition using equilateral and equiangular polygons. Three well-understood regular tessellations are the square, triangle, and hexagon. Bell et al. [1983] described these tessellations according to the type and number of polygons around each vertex of the atomic polygon. For example, [6^3] means there are six other triangles around a vertex, and the atomic triangle has three such vertices. A regular tessellation can be regarded as a dual of the vector data type. Unlike the vector

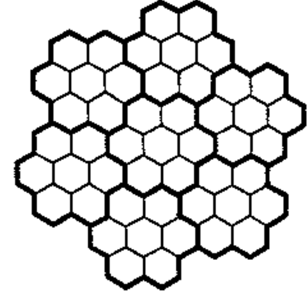
models, which use points as their logical units, regular tessellation models use polygonal meshes as their logical units [Peuquet 1984].



a. [4^4] Square Hierarchy



b. [6^3] Equilateral Triangle Hierarchy



c. [3^6] Hexagon Hierarchy

Figure 1.4 Space decomposition using three regular tessellations.

Hierarchies can be constructed based on the three regular meshes (Figure 1.4), which can be infinitely repeated to cover the whole image space. Unlike the other two, a hexagon is *limited* since we cannot subdivide it into infinitely finer hexagons for a better resolution. This is because hexagons have jagged edges. Squares and triangles can be recursively decomposed into four small areas of the atomic pattern; however only subdivision of squares maintains the shape and the orientation at the same time while the triangles' orientation difference is 60° [Samet 1990a, Ahuja 1983].

The primary advantage of a hexagonal mesh is this *uniform adjacency*; that is the distances between the centroid of the hexagon and the centroids of all its adjacent hexagons are equal. Each hexagon is adjacent to six other hexagons by sharing one-sixth of their boundaries [Gibson and Lucas, 1982]. Square and triangular tessellations do not have this property because the distances between the centroids of the common-side neighbour and the common-vertex neighbour are not equidistant.

Quad trees can be built on the tessellations based on both squares and triangles, and Ahuja [1983] proved that image operations on those two quad tree representations appeared to have the same complexity. Efforts have been made by Gibson and Lucas [1982] to develop what they called Generalized Balanced Ternary structures and the algorithms analogous to the ones for quad tree. The Euclidean plane tiles for quad trees and septrees are shown in Figure 1.5.

Figure 1.5 Tessellations at different resolutions and their Euclidean plane tiles.

1.3 Thesis Outline

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 spatial data collected in the Ocean Mapping Project are basically groups of soundings. In this thesis, the general objective is to build spatial indices for profiles and to test them by using a large number of profiles of real swath data from the EM100 depth system. It includes the following specific objectives:

1. Develop a method using the Morton sequence to index profiles of swath data, both directly and in a relational database management system (RDBMS) environment under INGRES. A timing comparison between the direct and RDBMS approaches will be made.

2. Investigate the use of R-trees to directly index profiles of swath data and answer the questions: "Is the performance of R-tree indexing better than that of the Morton code for indexing?" "Can the R-tree be better adapted to a dynamic update of the spatial index compared to the Morton code index?" and "How can the R-tree index be stored in a file?"

3. Investigate a hierarchical approach using R-trees to index groups of profiles, with each group having a sub-index such as another R-tree to index the data within the group.

Chapter Two gives the background of the Ocean Mapping Project and Hydrographic Data Cleaning System. Emphasis is given on the need for spatial indexing in HDCS, the surveyed data, data organization, file structures, and the assumptions for spatial indices and range searches.

Morton order indexing played a significant role in this thesis. Different Morton sequences on the same data space can be generated by using different code computing methods. Chapter Three discusses those methods and also explores the range search problem on Morton sequences. The search algorithm is described in this chapter. Index file structures are designed to store Morton indices for profiles and indices for directories of lines, days, vessels and projects under HDCS.

In Chapter Four, a relational database is used to realize the Morton sequence indexing. Continued from Chapter Three, this chapter is dedicated to the INGRES implementation of the spatial indexing and range searching method. Relations, or tables in INGRES terminology, are designed for indexing purposes.

Chapter Five opens a new page for spatial indexing in HDCS. We study the R-tree structures and the algorithms for building, deleting and searching the trees. The tree structure makes it possible to dynamically maintain the indices as the data is being changed or moved to other locations. A modified deletion algorithm is given based on the property that profiles are geographically close to each other and they are kept in the neighbouring leaf nodes in an R-tree structure.

Experiments based on the three implementations are performed using a large amount of real data. Chapter Six describes and compares the experimental results that lead to the conclusions in Chapter Seven. A summary is also included in the last chapter.

CHAPTER TWO

THE HDCS GENERAL STRUCTURES

2.1 Ocean Mapping Project

The Ocean Mapping Project at the University of New Brunswick started in the late 80's. Its objectives are [Ware et al., 1990]

> "to address current problems associated with processing the high volumes of data which are being generated by modern ocean bathymetry mapping systems. This data often cannot be fully utilized because of the difficulty of organizing and displaying this data using conventional techniques. Using a multidisciplinary approach implemented on a high performance workstation, we propose to devise techniques for error detection and correction, for seabed visualization, and for data interpretation."

2.2 HDCS Data and File Structures

The Hydrographic Data Cleaning System (HDCS) is a tool to clean or edit multibeam bathymetric data sets collected from a range of sensors such as the positioning system, gyro, vessel dynamics, tide gauges, and salinity/temperature/depth profiles along with the multibeam data [Ocean Mapping Group, 1991]. In order to make proper data cleaning decisions, all the observed data must be saved. The size of the four day survey data used in the experiment of Chapter Six for example, is 130 megabytes, including 57,192 profiles on 49 lines. It is required, therefore, that the HDCS be able to handle large volumes of data.

Surveyed data of one day are stored in a *daily file*. The raw data are converted to soundings at the very beginning of the data cleaning process, which is performed on a *working file*. This file may be read from parts of several daily files and the cleaned data will be written back into the daily files. The large volume of data makes spatial indexing a must when retrieving the data to form a working file (Figure 2.1).

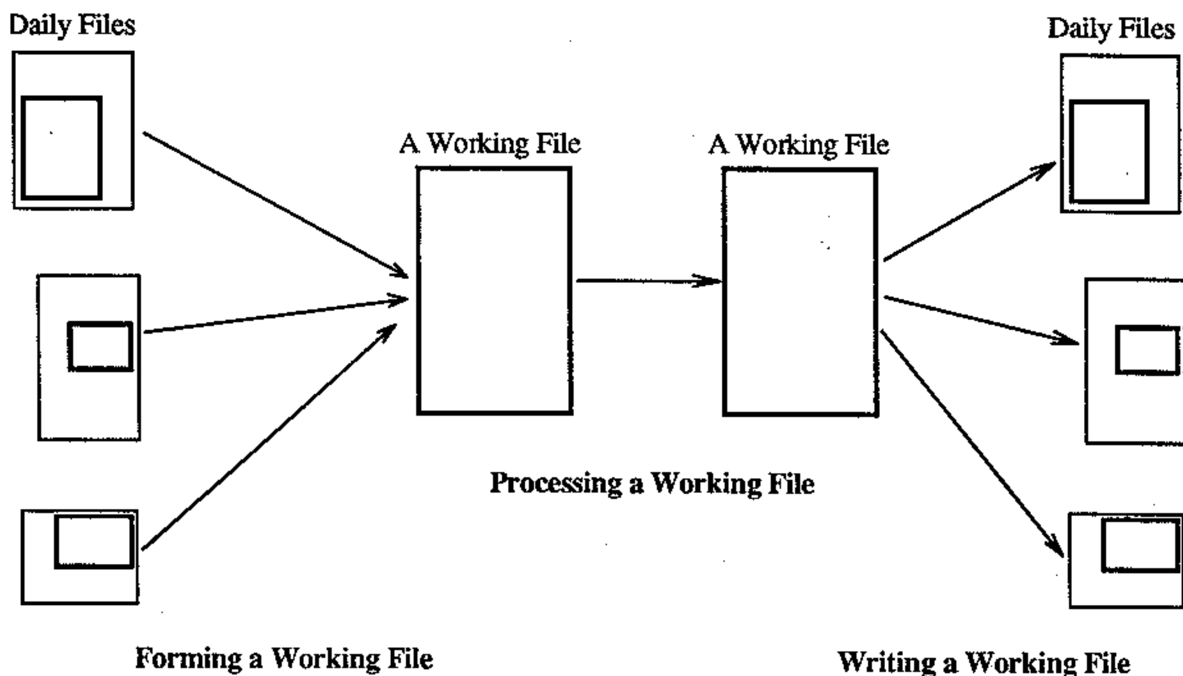


Figure 2.1 Hydrographic data cleaning process.

A daily file contains lines which represent a survey vessel's tracks at sea during the day. Moving along the path, the equipment on the boat makes multibeam bathymetric surveys. An array of depths perpendicular to the line can be calculated using the reflected sonar signals. Each array is called a *profile* and the depths are called *soundings* (Figure 2.2). Varying with the equipment used, each profile may have from 12 to 132 soundings [Ware et al., 1992].

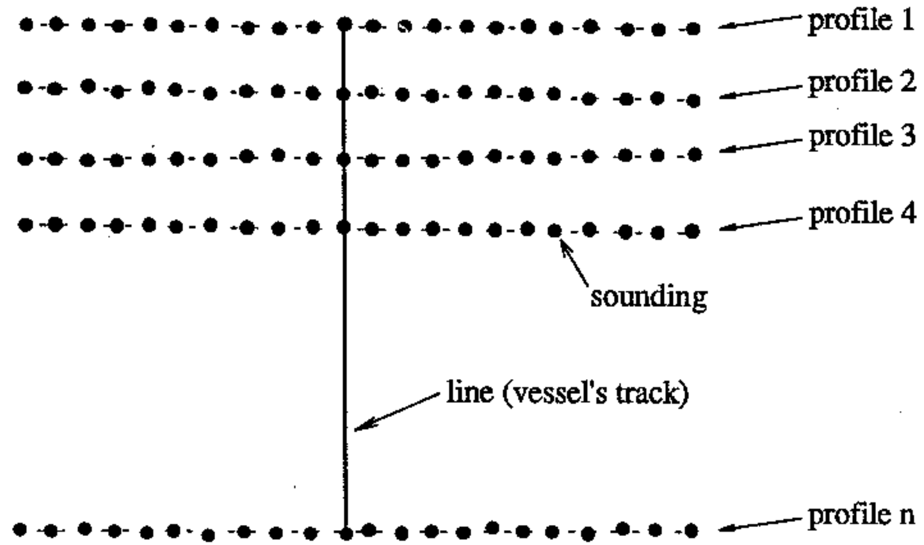


Figure 2.2 A line and its soundings.

2.2.1 HDCS File Organization

In the Hydrographic Data Cleaning System, surveyed data are organized hierarchically [Ware et al., 1992]. All the data are stored under the directory HDCS, which consists of directories named after each surveying project. In each project, several vessels may be used. Under each vessel directory are days, followed by line directories, storing the data surveyed in a day (Figure 2.3).

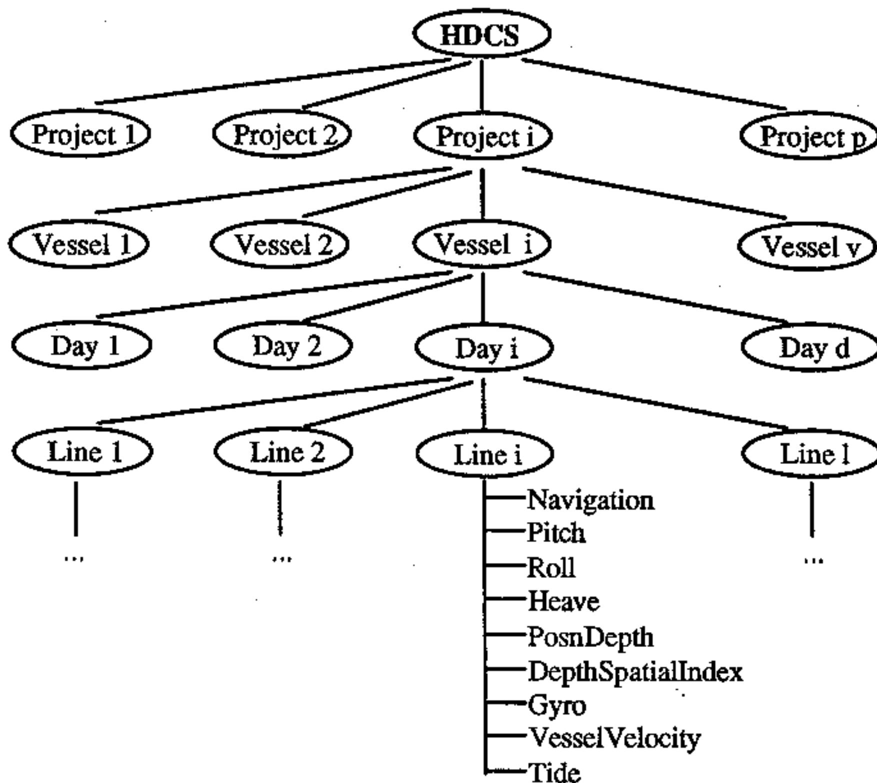


Figure 2.3 Hierarchical organization of ocean mapping data in HDCS.

2.2.2 HDCS File Structures

The HDCS file structures are depicted in Figure 2.4. Generally, surveyed data and configuration data are kept in data files. Given in the document [Ocean Mapping Group 1991] are the data files used in HDCS for the information of project, station, vessel, navigation, pitch/roll, heave, heading or course, vessel velocity, tide, depth, vertical offset, sound velocity, and position/depth. Each data file has a header record defining the header record size, data record size, number of data records, and other related information about the file. Data records have flags indicating the status of the data.

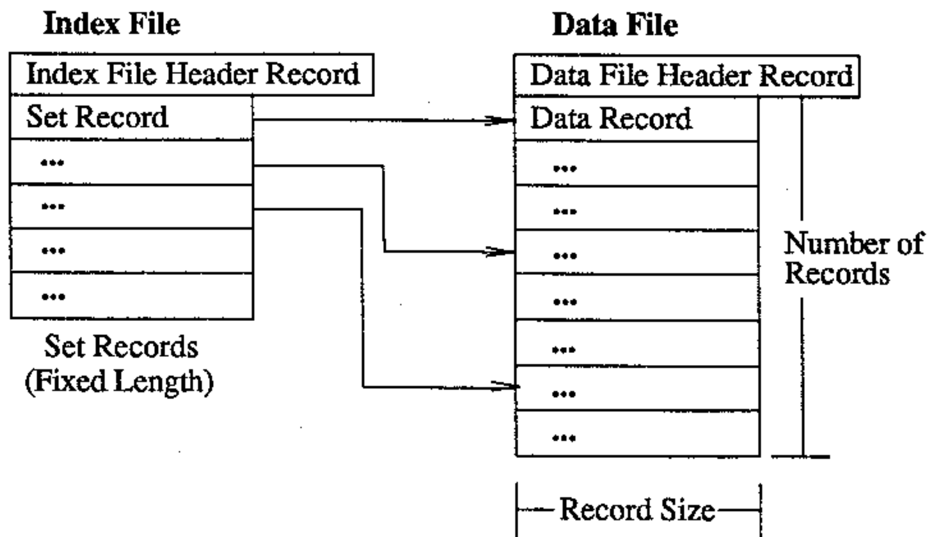


Figure 2.4 Index and data file general structure in HDCS.

Each data file has an index file of fixed structure; accessing individual data records requires a visit to the set record containing information about the data record. The size of a data set varies according to the type of data.

Of all the data files mentioned above, only the Position/Depth data file is of interest here. The data in this file have been corrected for tide, vessel dynamics, sound velocity profiles, and sensor placement. The conceptual file structure for the Position/Depth data is shown in Figure 2.5.

Posn/Depth Index

Posn/Depth index header
Posn/Depth set
Posn/Depth set
...
...
Posn/Depth set

Posn/Depth Data

Posn/Depth data header
Profile data
Depth data
...
...
Terminator
Profile data
Depth data
...
...
Terminator

Figure 2.5 Position/Depth file structure.

2.3 Spatial Indexing in HDCS

Spatial indexing files are used in the HDCS to make it efficient to access profiles falling within or overlapping a rectangular query window. A range search is used to form a working file and to support interactive queries. Some assumptions are made in the Data Structure Design [Ocean Mapping Group 1991] which are listed as follows:

1. Query windows (QW) used in range searches are rectangular.
2. A range search returns a whole profile intersecting with the QW; it leaves profile-query-window clipping to the system graphic functions.
3. A query window must be larger than any profile in the HDCS data. A constant has to be imposed on the minimum size of a QW based on the maximum length of all the profiles.

4. Sounding data must be converted to latitude/longitude/depth format before spatial indices are built. Range searches are also performed on data of this format.
5. Spatial searches should support interactive queries.

Guided by the assumptions, spatial index files are built on two kinds of objects: profiles and directories. The basic objects we need to index are the profiles on each line. A profile consists of a constant number of sounding points whose geographic locations can be described by latitudes, longitudes and depths. Minimal bounding rectangles (MBRs) are used in building indices. A profile's MBR can be obtained through finding the minimum latitude and longitude, and maximum latitude and longitude of all the soundings on the profile. A line's MBR is a rectangle that covers all the profile MBRs. The MBRs for a day, vessel and project directory can also be easily obtained by computing a bigger MBR covering the smaller MBRs under the directory.

The hierarchical nature of the HDCS tree structures provide us an excellent base for building hierarchical indexing structures. We can not only establish index files at the line level for profiles, but also build index files at the lower levels of the tree, i.e., in the day, vessel, and project directories (Figure 2.6).

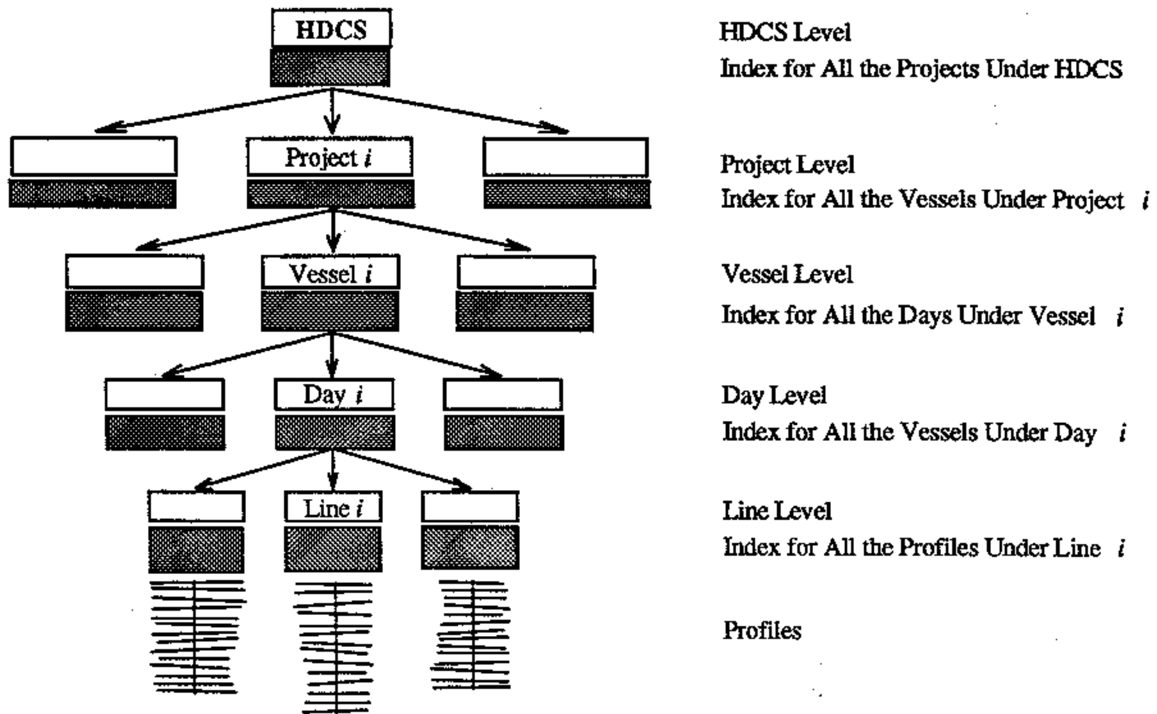


Figure 2.6 Hierarchical spatial index structures.

Since one MBR is needed for each directory and the number of directories is limited, sequential files are used for directory indexing. Sequential scans are used to perform range searches on those files. On the contrary, sophisticated indexing mechanisms are used for indexing profiles. Either Morton sequences or R-trees are used for profile indexing.

Like the structure of a data file, each index file also has two parts: file header and file data. The file header is used to store the summary information for the file. Each record in the file data stores the actual indexing data. Chapters Three, Four and Five elaborate on the index file structures for directories and profiles.

2.4 Spatial Searches in HDCS

Spatial searches in HDCS involve retrieval *inside profiles* of a query window, described using four corner points in pairs of latitude and longitude. Given such a range, a query can be carried out in three-step-access (see Figure 2.7) [Ocean Mapping Group 1991], as follows:

1. Search the directory spatial index files for vessels, days and lines, and find out the lines whose MBRs fall into or overlap with the QW.

2. Search the profile spatial index files under such lines, and obtain the profiles whose MBRs are within or overlap with the QW.

3. Use the actual sounding data of the profiles to rule out the profiles which do not intersect the query window.

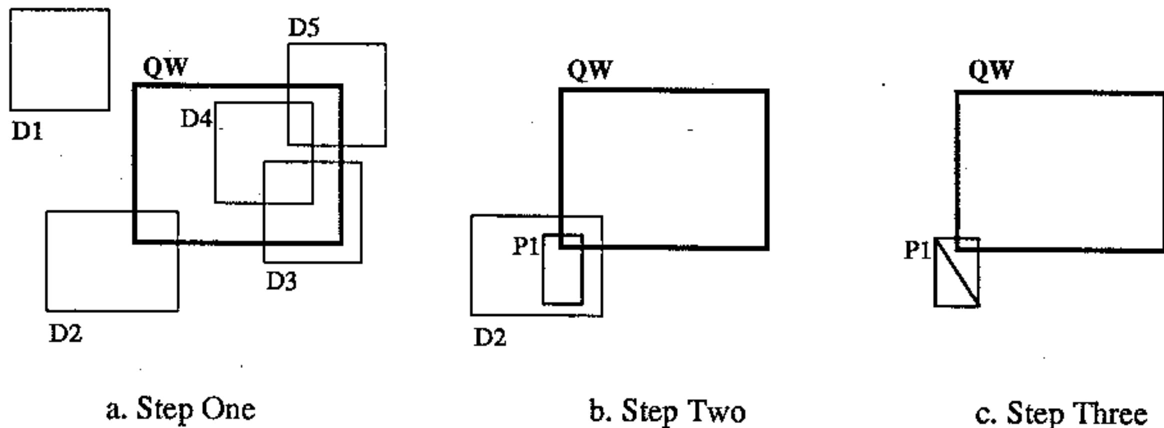


Figure 2.7 Three step access for inside profile testing.

The range search sequentially scans a directory spatial index file, and checks each minimal bounding rectangle to find the relation between the QW and the MBR. Four test results can be obtained as follows:

1. The QW overlaps an MBR;
2. The QW is enclosed by an MBR;
3. The QW encloses an MBR; or
4. The QW and an MBR are separate.

In the first two cases, further tests of the relations between the QW and the directory's sub-directory as well as profile MBRs must be done. On the other hand, when a directory MBR is entirely within the search range as in the third possibility, all the sub-directory and profile MBRs under this directory can be included in the search results without any extra examination. For example, if it is tested positive that a project directory minimal bounding rectangle is totally within a given query window, then, as a matter of fact, all the MBRs of the project's vessel, day and line directories, as well as the profiles of each line, must be within the query window. Therefore all the spatial data under the project can be counted as search results. Finally when a directory MBR is totally outside the search range, the directory itself, the subdirectories and the profiles under this directory are totally outside the query window. No further test is necessary either.

The input for the range search program are a query window and the name of a project under which searches for the profiles are required. The search results, returned by the program, are organized hierarchically according to the vessels, days, lines, and the profiles under the specified project. Only the directories and the profile numbers which are within or overlap the query window are kept in the SpatialSubset structure listed in Appendix A.

CHAPTER THREE

MORTON CODE INDEXING

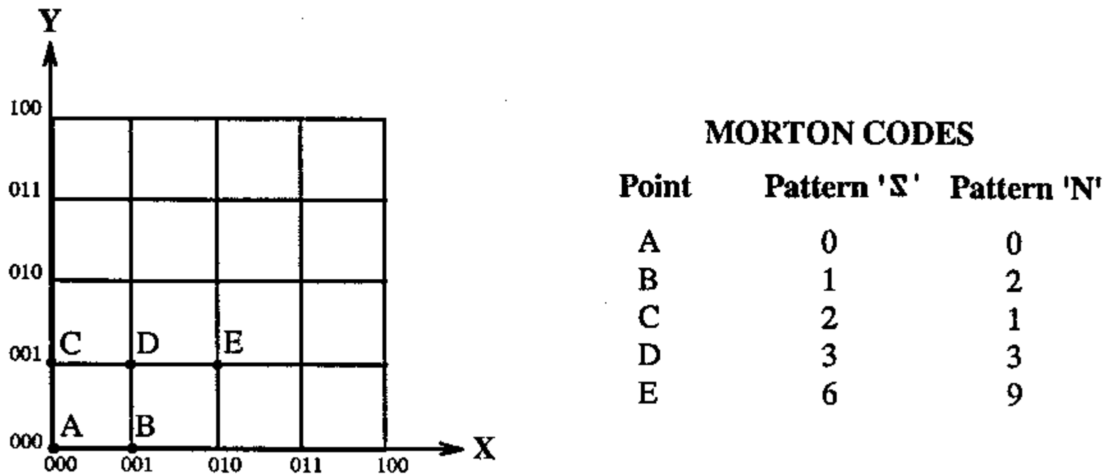
3.1 Morton Codes

Morton codes are named after G. M. Morton who first introduced the codes and used them in the development of the world's first operational geographic information system called Canadian Geographic Information System (CGIS) [Morton 1966, Tomlinson 1976].

A spatial point can be represented by a Morton code, which is a decimal number computed by interleaving the binary representations of the point's coordinates.

3.1.1 Encoding Morton Codes

The process of constructing a Morton code is called *encoding*. In order to illustrate how a Morton code is calculated, a $2^2 x 2^2$ grid space is used (Figure 3.1) with each grid line intersection being addressed by x- and y-coordinates. Three bit binary numbers are needed in order to represent all the coordinates in the space. For example, points A, D, and E have coordinates $(0,0)_{10} = (000, 000)_2$, $(1,1)_{10} = (001,001)_2$, and $(2, 1)_{10} = (010, 001)_2$, respectively.



MORTON CODES

Point	Pattern 'Z'	Pattern 'N'
A	0	0
B	1	2
C	2	1
D	3	3
E	6	9

Figure 3.1 Points in binary coordinates and their Morton codes.

Two ways of encoding Morton codes of point E are given in Figure 3.2a and b. Taking the lowest bit of x first in a Morton code gives us decimal number 6. Otherwise, using the lowest bit of y first generates a Morton code of 9, instead.

a)

$$
\begin{array}{r}
x = (2)_{10} = (\quad 1 \quad\; 0\,)_2 \\
y = (1)_{10} = (0 \quad 1 \quad\;\,)_2 \\
\hline
\text{Morton code} = (0\ 1\ 1\ 0)_2 = (6)_{10}
\end{array}
$$

b)

$$
\begin{array}{r}
x = (2)_{10} = (1 \quad 0 \quad\;\,)_2 \\
y = (1)_{10} = (\quad 0 \quad\; 1\,)_2 \\
\hline
\text{Morton code} = (1\ 0\ 0\ 1)_2 = (9)_{10}
\end{array}
$$

Figure 3.2 Calculating Morton codes by interleaving bits.

3.1.2 Decoding Morton Codes

In a range search, we cannot tell if a point is within a query or not by looking only at the Morton code of the point. The coordinate information carried in the code, however,

can be used in the test. As a reverse process of encoding, *decoding* uses the bit representation of a Morton code and restores them, from the lowest bit to the highest bit, to the original coordinates.

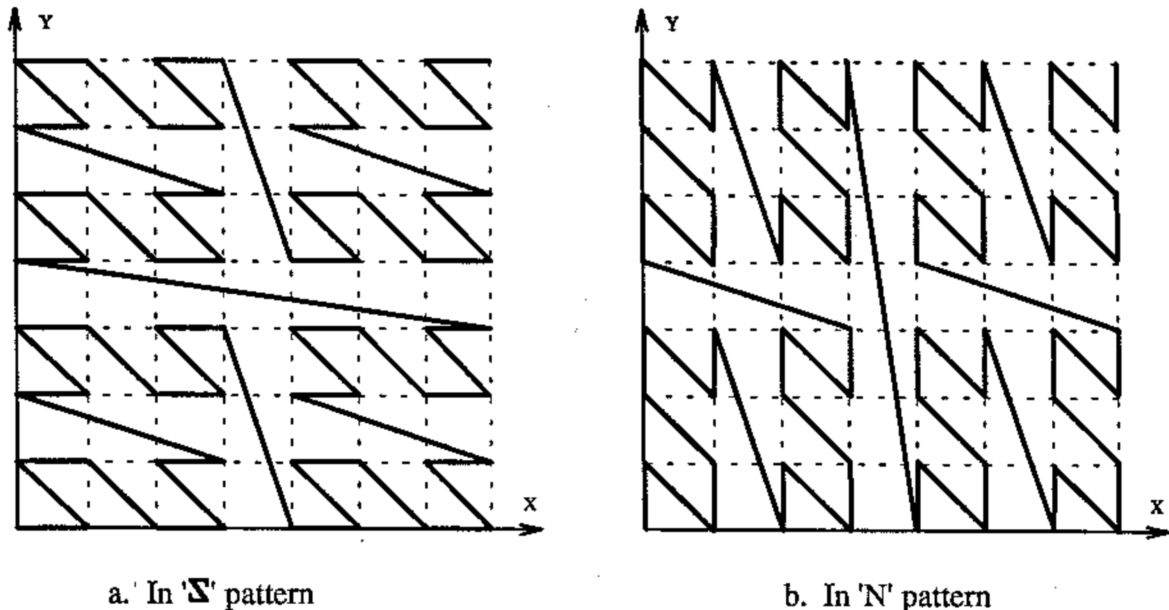
3.2 Morton Sequence of Points

3.2.1 Morton Sequence

A Morton sequence is a sorted order of Morton codes. Figure 3.3 gives examples of Morton sequences for all the coordinate points in a 2D space. In the examples, Morton codes are first calculated for each point, and then sorted in an increasing order.

In Section 3.1, we mentioned that there are two ways to calculate a point's Morton code. Those two methods yield two Morton sequences of the same space. The Morton sequence in Figure 3.3a walks through the space in a 'Σ' pattern while the sequence in Figure 3.3b traverses the space in an N pattern.

These two appearances of the Morton ordering are identical in characters, and are widely used in spatial data handling for indexing purposes. The 'Σ' pattern is chosen as the indexing mechanism in this thesis.



a. In 'Z' pattern b. In 'N' pattern

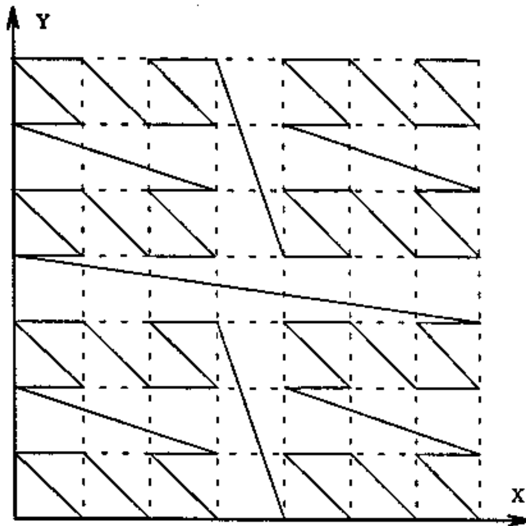
Figure 3.3 Two Morton sequences for 2D points.

3.2.2 Characters of Morton Sequences

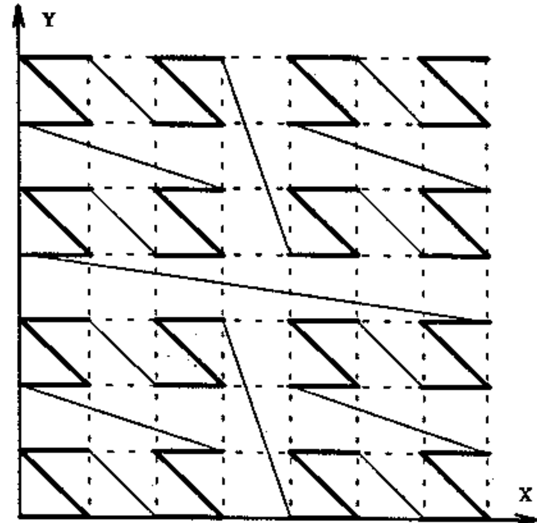
The characters of Morton sequences make them very useful space-orderings. Morton codes, as the keys in a Morton sequence, are position related. A Morton code is defined only by the coordinates of a point. Encoding and decoding of a Morton code are also easy to specify. As long as the pattern is pre-defined, points of different coordinates have different Morton codes.

As for all space-filling curves, a Morton sequence transforms two-dimensional space into a linear order which keeps the neighbours in the original space as close as possible in the sequence. This filling is exhaustive; it enters the space at the origin of the space, and passes through every point before it exits.

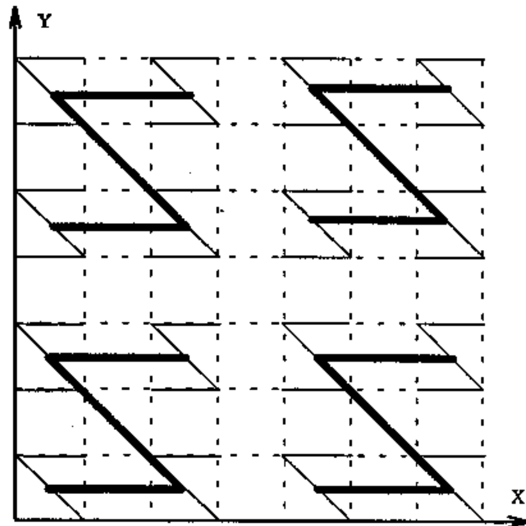
Another major advantage of this zigzag order is its quad tree related decomposition and recursion. Each corner of a 'Σ' pattern can be regarded as a quadrant, or a representation of small such patterns at a higher resolution. This property can be clearly seen in Figure 3.4a-d.



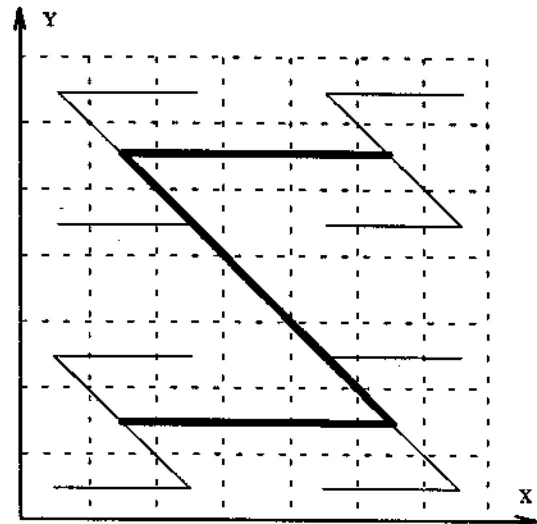
a) A Morton Sequence for 2D Points



b) The Highest Level of 'Σ' Pattern



c) A Lower Level of 'Σ' Pattern



d) The Lowest Level of 'Σ' Pattern

Figure 3.4 Quad tree related decomposition and recursion of a Morton sequence.

Quadrants can be coded into base 4 numbers in a scheme such as that drawn in Figure 3.5 with the south-west quadrant denoted as 0, the south-east denoted as 1, the north-west as 2 and the north-east as 3. A Morton code in Figure 3.4 can be represented in three digits with the first digit representing the lowest level of 'Σ' pattern, and the last digit the highest level. Morton code 27_{10}, for example, is 123_4, and its corresponding point can be found located at quadrant 1 at the lowest level, quadrant 2 at the second level, and quadrant 3 at the highest level (see Figure 3.7).

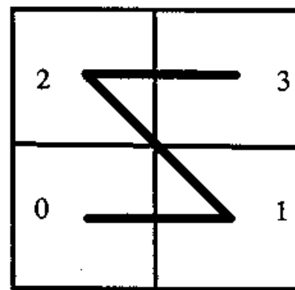


Figure 3.5 Numbering scheme in base 4 digits.

This character of quad tree related decomposition and recursion can be used to determine the direction when walking through a Morton sequence. It can be clearly seen that the walking order is 0, 1, 2 and 3, and this order holds at all the levels.

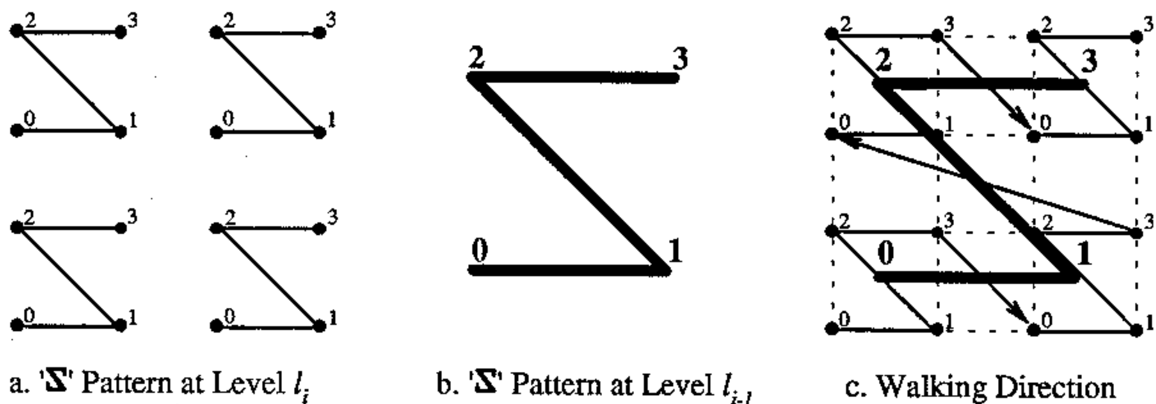
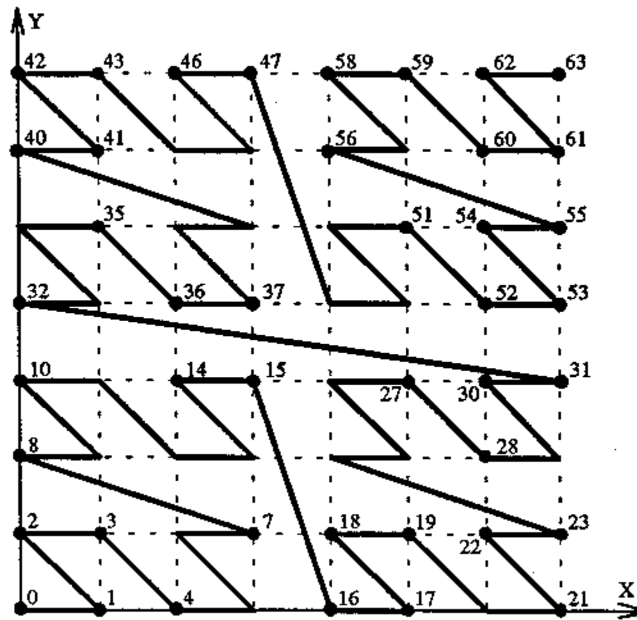


Figure 3.6 The walking pattern of a Morton sequence.

Deciding the direction after quadrant 3 is subtle. We have to look one level lower, and consider the quadrant number of the current 'Σ' in this lower level. Suppose that the current 'Σ' pattern is at level l_i and the immediate lower level is l_{i-1}. If the 'Σ' at level l_i is in quadrant 0 of level l_{i-1}, the Morton sequence goes to the first Morton code in quadrant 1 of level l_{i-1} (Figure 3.6). Similarly, the first Morton code at quadrant 3 of level l_{i-1} follows the last Morton code at quadrant 2. When the 'Σ' pattern of level l_i is quadrant 1 of level l_{i-1}, the Morton sequence jumps to quadrant 2 of level l_{i-1}. The walking directions are arrowed in Figure 3.6c.

3.2.3 Morton Sequence of Data Points

So far, we have been describing Morton sequences based on the points at the interval of the coordinate units. This unit is the resolution of such points in the space, and the points themselves are called *resolution points*. In a particular application, not every resolution point is occupied by an actual *data point*. A Morton sequence can still be built without any modification. Figure 3.7 gives an example of data points in solid dots and its corresponding Morton sequence.



0, 1, 2, 3, 4, 7, 8, 10, 14, 15, 16, 17, 18, 19, 21, 22, 23, 27, 28, 30, 31, 32,

35, 36, 37, 40, 41, 42, 43, 46, 47, 51, 52, 53, 54, 55, 56, 58, 59, 60, 61, 62, 63

Figure 3.7 Data points and the corresponding Morton sequence.

3.3 Range Search Using Morton Sequence

We have mentioned in Chapter One that a range search involves retrieving or counting all the spatial objects within a given range. In the HDCS, such search ranges are assumed to be rectangular. We, therefore, mainly consider orthogonal query windows whose edges are parallel to the coordinate axes.

Before we get into the details of a range search on a Morton sequence, we should bear in mind that such a sequence is *monotonic*, that is if, and only if, for a fixed coordinate x or y, the Morton sequence varies monotonicaly with the other coordinate y or x, respectively [Abel and Mark 1990]. It is not hard to prove that among the four

corners of a query window, the upper-right one has the biggest Morton code MC_{UR} while the lower-left has the smallest Morton code MC_{LL}. All the points inside the query window have their Morton codes bigger than MC_{LL} and smaller than MC_{UR}. The Morton code of point P in Figure 3.8, for example, is bigger than MC_A which is bigger than MC_{LL}, and MC_P is smaller than MC_B which is smaller than MC_{UR}.

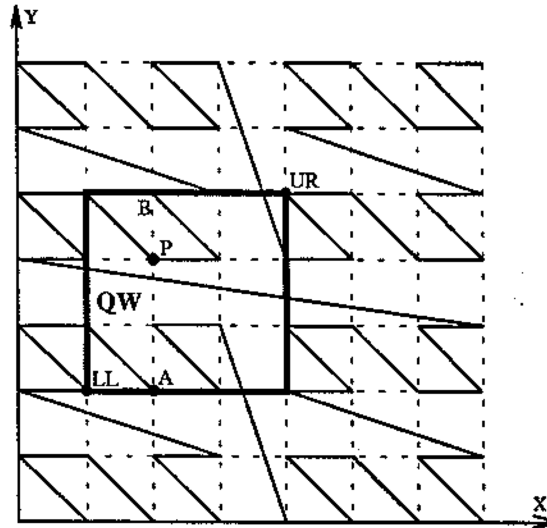


Figure 3.8 The Morton sequence and a query window.

3.3.1 Linear Range Search Algorithm on Morton Sequence

Once the Morton sequence of a point set has been built, a range search can be performed in the following way:

1. The Morton codes of the query window's lower left and upper right corners, MC_{LL} and MC_{UR}, are calculated. Those two numbers are used as a search boundary with the Morton codes of in-window points falling in between.

2. A binary search is employed to find the location of MC_{LL} in the Morton sequence. If there is no data point at the lower-left corner of the QW, the location of a Morton code just greater than MC_{LL} should be used.

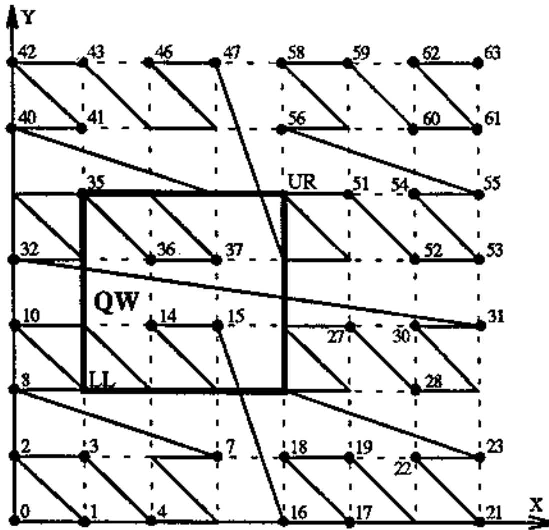
3. Morton codes larger than MC_{LL} will be visited linearly, until the end of the sequence or a Morton code larger than MC_{UR} is reached, whichever comes first. All the points within the QW will be reported.

In the example of Figure 3.9a, the Morton codes for the query window are MC_{LL}=9 and MC_{UR}=50, and the Morton codes we test are 10, 14, 15, 16, 17, 18, 19, 21, 22, 23, 27, 28, 30, 31, 32, 35, 36, 37, 40, 41, 42, 43, 46, 47. Notice that only five points, i.e. 14, 15, 35, 36 and 37, are in the query range (see Figure 3.9b).

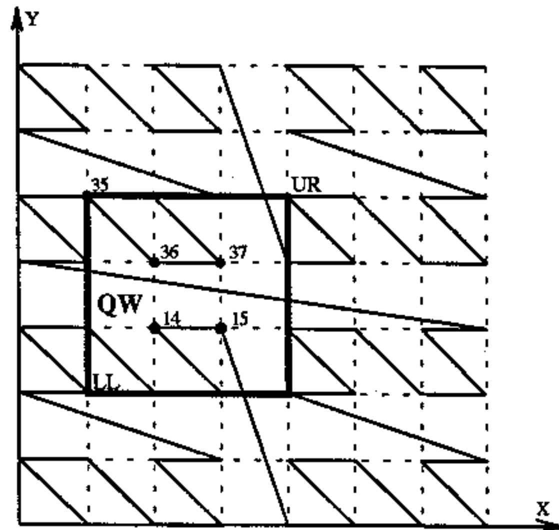
3.3.2 Over-search Problems

Any visit to unqualified data in a range search is an *over-search.* Over-searches should be avoided as much as possible.

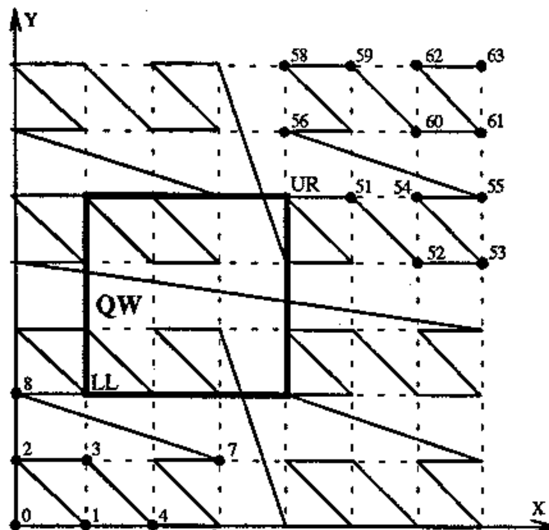
Caused by the points outside a query window, the over-search problem exists when using the linear search algorithm. This problem varies with the location of data and the query window, as well as the distribution of data. The linear search algorithm partially eliminates the over-search problem by quickly finding the Morton code greater than MC_{LL}, and by skipping the Morton codes greater than MC_{UR} on the Morton sequence (Figure 3.9c). All the Morton codes with values between MC_{LL} and MC_{UR}, however, have to be visited, even though there are only five points within the window (Figure 3.9d).



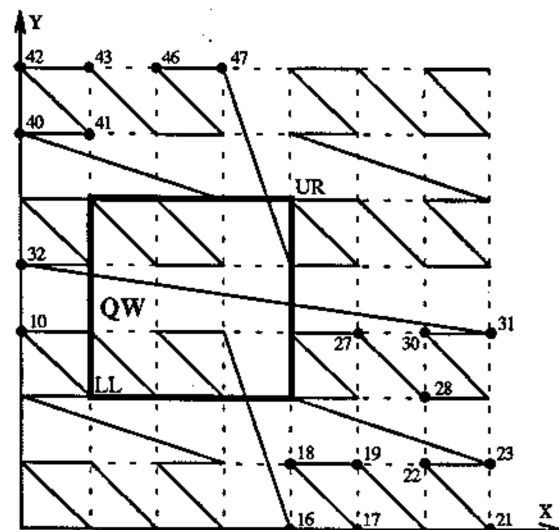
a. Data Points and a Query Window



b. Data Points within the Query Window



c. Data Points Immediately Excluded From the Query Window



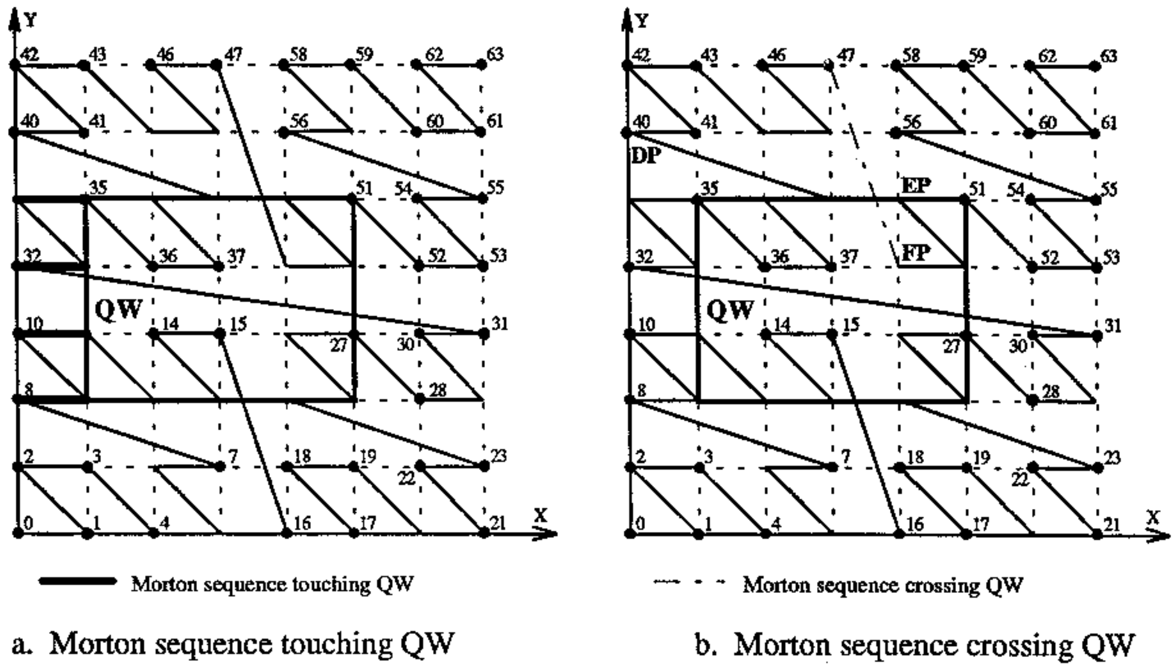
d. Data Points Causing Over-search of the Query

Figure 3.9 A range search on a Morton sequence and its over-search problem.

3.3.3 Modified Range Search Algorithm

Trying to minimize the number of disk accesses, Yang's algorithm [1992] stops the linear search as long as the Morton sequence leaves the query window, calculates the next point where the sequence enters the QW, and resumes the linear search from the entering point. For example, in Figure 3.9a, after the sequence leaves the query window at Morton code 16, the entering point 24 is calculated. Starting the search again at point 24, allows points 17, 18, 19, 21, 22, and 23 to be skipped.

Based on the recursive character of Morton sequences, two important calculations are performed in the algorithms. The first is to detect on which side of the query window the Morton sequence is re-entering the range and meanwhile decide if the Morton sequence touches the side of the query window, i.e., stops at the same quadrant (Figure 3.10a), or crosses the side, i.e., changes quadrants at a level less than the highest (Figure 3.10b). The second is to compute edge point EP for a crossing bridge and obtain the first point FP referring to EP and the data point DP.



a. Morton sequence touching QW

b. Morton sequence crossing QW

Figure 3.10 Relation of Morton sequence and query window.

3.4 Morton Sequence Indexing of HDCS Profiles

This section introduces the HDCS structures, index file structures and the way Morton sequences are used in indexing profiles. A hierarchical indexing structure is used in the HDCS with index files for directories and profiles. The Morton sequence indexing technology is used only at the profile level since the number of directories is relatively small and scanning the directory index files is quite trivial.

3.4.1 HDCS Spatial Indexing

In each project directory, a spatial index file is built for the MBRs of vessel directories under this project; in each vessel directory an index file for MBRs of day directories under each vessel; in each day directory an index file for the MBRs of line directories under each day. A directory name is the key in the index file.

Spatial index files for profiles are built in the line directories. For each profile's MBR, four Morton codes are calculated for the four corners. The Morton codes for profiles on a line are sorted into a linear Morton sequence and stored into an index file. A Morton code is a key for an index file record, and the corresponding profile number is its attribute.

3.4.2 Spatial Resolution of 64-bit Morton Code

A 64-bit unsigned integer is required for each Morton sequence, since four-byte integers are used to represent the coordinates of each sounding point. A 64-bit integer is beyond the limit of a 32-bit workstation. A structure with two four-byte unsigned integers is employed. The first unsigned integer represents the lower bits of a Morton code, while the second holds the higher bits.

Table 3.1 A type definition for a 64-bit Morton code.

```
typedef unsigned long MortonCode[2];
```

Since the earth's radius can be roughly taken as 6378 km, if a 32-bit unsigned integer is used to represent the longitude, which has the range from [0, 2π), the minimum distance an unsigned integer can represent is calculated as:

$$\text{longitude resolution} = \frac{\text{The earth's perimeter}}{\text{Maximum number of an unsigned 32-bit integer}}$$

$$= \frac{2 \times \pi \times 6378 \times 10^6 mm}{2^{32} - 1}$$

$$= \frac{40074155889.191399 mm}{4294967295}$$

$$= 9.33 mm$$

which is enough for the precision of longitude representation.

The range for latitude is [-π/2, +π/2] which is half the range of longitude, and the resolution for latitudes is 4.67mm. Therefore a 32-bit unsigned integer variable is precise enough for both latitudes and longitudes.

3.4.3 Encoding and Decoding of Morton Codes in HDCS

The 'Σ' pattern encoding of Morton codes are employed in HDCS spatial indexing. It is mentioned in Section 3.4.2 that the first element in the two element array of Table 3.1 stores the lower bits of a Morton code and the second element of the array stores the higher bits of the code. The first bit of MortonCode[0] is set to be the first bit of latitude, and the second bit of MortonCode[0] is set to be the first bit of longitude. The interleaving goes on till all the bits in MortonCode[0] are set to the lower bits of the latitude and

longitude. The first bit of MortonCode[1] is set to be the 17th bit of the latitude and the second bit of the MortonCode[1] is set to be the 17th bit of the longitude, and so on.

Decoding Morton codes is really a reverse process of encoding. Bits in a two element Morton code are taken from lower position to higher position, and are put first in latitude then longitude variables starting from their lower positions.

Algorithms for encoding and decoding are given in Appendix B.

3.4.4 HDCS Index File Structures

As mentioned in Chapter 2, the spatial index files have two parts: header and data records. A header record contains information about a file, and the data records store the spatial indexing data.

Tables 3.2-3.5 list the structures for the header and data of directory and profile spatial index files. The profile spatial index files are sorted according to the Morton codes of the profile MBR corners, while the directory spatial index file records are randomly listed.

A Morton code in Tables 3.4 and 3.5 is actually a two element array described in Table 3.1.

3.4.5 Creating Spatial Index Files

All the directories are processed in post-order. Spatial index files are first created for profiles, meanwhile the MBRs are found for those line directories whose profile spatial index files are being built. In turn, MBRs are found for day directories, vessel directories, and project directories while building spatial index files for line directories, and finally, vessel directories.

Table 3.2 Directory spatial index file header structure.

Item	Type	Size	Units
Size of this header record	integer	32 bits	Bytes
Size of a record in this data file	integer	32 bits	Bytes
Number of records in this data file	integer	32 bits	
HDCS file type	integer	32 bits	
File version number	integer	32 bits	
Reference time for this header file	integer	32 bits	100 seconds
Data time scale for time offsets in this file	integer	32 bits	No. of microseconds
Minimum time (offset w.r.t. reference time)	integer	32 bits	Data time scale
Maximum time (offset w.r.t reference time)	integer	32 bits	Data time scale
Position scale for position offsets in this file	integer	32 bits	No. of nano-radians
Reference latitude for this header file	integer	32 bits	100 nano-radians
Minimum latitude (offset w.r.t. reference latitude)	integer	32 bits	Position scale
Maximum latitude (offset w.r.t. reference latitude)	integer	32 bits	Position scale
Reference longitude for this header file	integer	32 bits	100 nano-radians
Minimum longitude (offset w.r.t. reference longitude)	integer	32 bits	Position scale
Maximum longitude (offset w.r.t. reference longitude)	integer	32 bits	Position scale
Maximum latitude difference for one profile's bounding rectangle	integer	32 bits	100 nano-radians
Maximum longitude difference for one profile's bounding rectangle	integer	32 bits	100 nano-radians

Table 3.3 Directory spatial index file record structure.

Item	Type	Size	Units
Directory MBR's minimum time (offset w. r. t. reference time)	integer	32 bits	Data time scale
Directory MBR's maximum time (offset w. r. t. reference time)	integer	32 bits	Data time scale
Directory MBR's minimum latitude (offset w.r.t. reference latitude)	integer	32 bits	Position Scale
Directory MBR's maximum latitude (offset w.r.t. reference latitude)	integer	32 bits	Position Scale
Directory MBR's minimum longitude (offset w.r.t. reference longitude)	integer	32 bits	Position Scale
Directory MBR's maximum longitude (offset w.r.t. reference longitude)	integer	32 bits	Position Scale
Directory name	char	256 characters	

Table 3.4 Profile spatial index file header structure.

Item	Type	Size	Units
Size of this header record	integer	32 bits	Bytes
Size of a record in this data file	integer	32 bits	Bytes
Number of records in this data file	integer	32 bits	
HDCS file type	integer	32 bits	
File version number	integer	32 bits	
Reference time for this header file	integer	32 bits	100 seconds
Data time scale for time offsets in this file	integer	32 bits	No. of microseconds
Minimum time (offset w.r.t. reference time)	integer	32 bits	Data time scale
Maximum time (offset w.r.t reference time)	integer	32 bits	Data time scale
Minimum Morton code stored in the data file	unsigned	64 bits	
Maximum Morton code stored in the data file	unsigned	64 bits	
Maximum latitude difference for one profile's bounding rectangle	integer	32 bits	100 nano-radians
Maximum longitude difference for one profile's bounding rectangle	integer	32 bits	100 nano-radians

Table 3.5 Profile spatial index file record structure.

Item	Type	Size	Units
Morton code for this profile minimal bounding rectangle corner point	unsigned	64 bits	
Sequential number of this profile in the line	integer	32 bits	

3.4.6 Range Searches Using the Index

The modified search algorithm [Yang 1992] was adapted to the HDCS application and used in this thesis to perform range search on profile Morton sequences. Pseudo code for the search is listed in Appendix B after the following explanations are given.

A query window divides the data space into nine areas (Figure 3.11). Several cases must be considered depending on where the Morton sequence enters the QW from. For example, if the outside-window point is in Area Eight, the Morton sequence may touch the window on the top or bottom edge, or may cross the top edge. Function FindEnteringPoint handles these cases.

When a Morton sequence crosses an edge of the query window (Figure 3.10b), a Morton code on the edge, MC_{EP}, must be calculated first by function GetQWEdgeMC, then the first Morton code in the QW, MC_{FP}, can be calculated by FallIntoQW. MC_{EP} is the smallest Morton code among all the on-edge Morton codes which are larger than the Morton code of the data point outside the query window, MC_{DP}. If the Morton sequence touches an edge, the touching point's Morton code is also the smallest among all the on-edge codes lager than MC_{DP}. It can also be calculated using function GetQWEdgeMC.

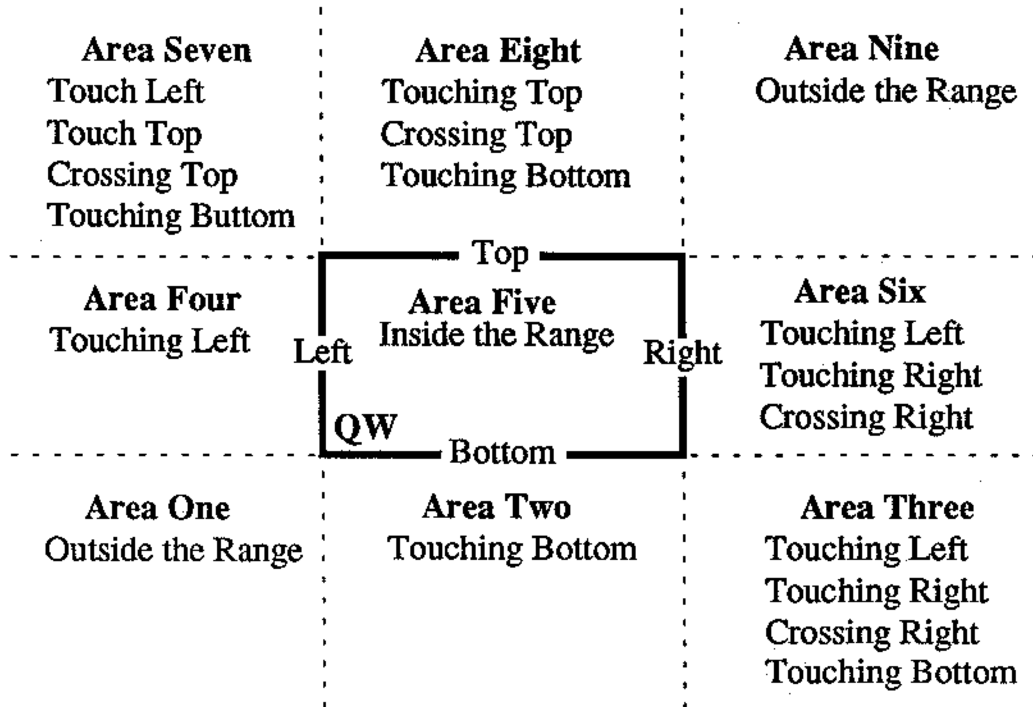


Figure 3.11 Possible cases Morton sequence touching or crossing QW edges (Adapted from Yang [1992]).

Figure 3.9(a) is repeated in Figure 3.12 and used as an example to explain the range search algorithms on Morton code indices. First the Morton codes for the corners of the QW are calculated: $MC_{LL} = 9$, $MC_{UL} = 35$, $MC_{LR} = 24$, and $MC_{UR} = 50$. Binary searches on the Morton sequence are involved to find the first MC which is not smaller than MC_{UR}, and the first MC which is not small than MC_{LL}. These two Morton codes are 51 and 10, respectively.

The first test point is 10 which is smaller than MC_{UR}. So Morton code 10 is decoded to obtain the point's latitude and longitude used in detecting if the point is within the QW. Since point 10 is not within the QW, Morton code 11 at which the Morton

sequence enters the QW is calculated by calling function FindEnteringPoint. The Morton code on the Morton sequence which is just larger than 11 is Morton code 14. Since 14 is less than MC_{UR}, it is decoded and tested against the QW. The test is positive so the counter for the profile represented by Morton code 14 is increased by one, and next Morton code, 15, on the sequence is obtained. Similar tests are carried on with Morton code 15 and 16, and then an entering Morton code 24 is calculated. The loop from Step 6 to Step 16 in Algorithm 1 of Appendix B continues until a Morton code larger than MC_{UR} is reached.

Finally all the profiles whose counter are larger than zero are reported as within the query window.

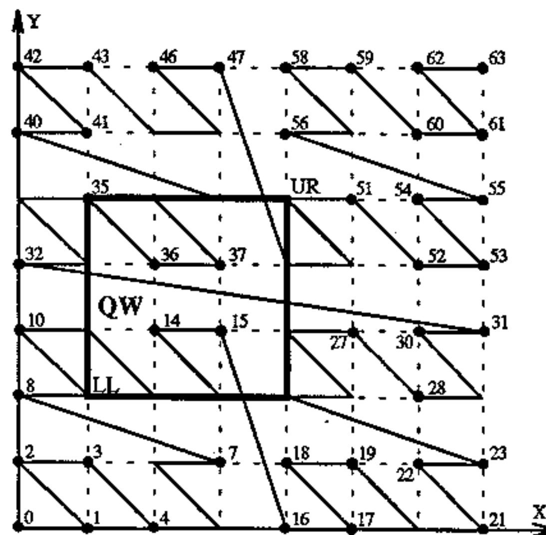


Figure 3.12 A range search on a Morton sequence.

CHAPTER FOUR

RELATIONAL DATABASE MORTON CODE INDEXING

4.1 Introduction

Database management systems (DBMSs) are powerful tools in handling data of large volume. Commercial DBMSs provide the facilities to define, insert, update and retrieve data with the added value of reducing redundancy, controlling concurrency, supporting data independence, and enforcing data integrity. With its flexibility in data structures and system designs [Date 1981] and its rapid improvement in performance, relational database management systems (RDBMSs) are one of the main methods for large scale data storage. Researchers are also exploring the use of RDBMSs to deal with geographical information [Waugh and Healey 1987, Abel 1989].

In order to compare the performance with direct Morton sequence indexing, a commercial RDBMS, INGRES, was used in this thesis. Tables are designed as close as possible to the direct indexing file structures described in Chapter Three. The C language was used as host for embedded SQL to build and visit those tables.

4.2 Relational Tables for Morton Code Indexing

Under the INGRES environment, all the spatial index information is kept in relations, or tables using INGRES terminology. To have a unique schema for each table,

the spatial index file headers (see Tables 3.2 and 3.4) are separated from the files. Profile spatial index file headers are combined with the Directory spatial index file data (Table 3.3) into a new table (Table 4.2). Directory spatial index file headers are no longer necessary.

All the tables are maintained in a database. In order to compare the timing for building and searching the INGRES tables with those in the direct Morton code indexing, hierarchical index mechanisms are also preserved in INGRES spatial indexing. For each line, there is a table containing all the profile Morton codes of the line, and for each directory, a table accommodating all the MBRs of the immediate sub-directory. For example, a project has a table for all the vessel directories under this project.

The tables for the directories have the same schema, and so do the tables for the profile Morton sequences of different lines. The reason for organizing the indices in different tables is that if a directory is found totally outside the query, then the tables containing indices for the directories and profiles under this directory can be ignored.

The database schemata are listed in Tables 4.1-4.3 with their keys in italic font. Only the data necessary for indexing from Tables 3.2-3.5 are included in the design.

Table 4.1 INGRES table for projects.

Item	Type	Size	Units
Project name	char	64	Characters
Table name for the project	char	32	Characters

Table 4.2 INGRES table for directories.

Item	Type	Size	Units
Directory name	char	64	Characters
Reference time for this header file	integer	32 bits	100 seconds
Data time scale for time offset in this file	integer	32 bits	No. of microseconds
Minimum time (offset w.r.t. reference time)	integer	32 bits	Data time scale
Maximum time (offset w.r.t. reference time)	integer	32 bits	Data time scale
Position scale for position offsets in this file	integer	32 bits	No. of nano-radians
Reference latitude for this file	integer	32 bits	100 nano-radians
Minimum latitude (offset w.r.t. reference latitude)	integer	32 bits	Position scale
Maximum latitude (offset w.r.t. reference latitude)	integer	32 bits	Position scale
Reference longitude for this file	integer	32 bits	100 nano-radians
Minimum longitude (offset w.r.t. reference longitude)	integer	32 bits	Position scale
Maximum longitude (offset w.r.t. reference longitude)	integer	32 bits	Position scale
Maximum latitude difference for one profile's bounding rectangle	integer	32 bits	100 nano-radians
Maximum longitude difference for one profile's bounding rectangle	integer	32 bits	100 nano-radians
Table name for the lower directory	char	32	

Table 4.3 INGRES Table for Profiles.

Item	Type	Size	Units
Morton code for this profile minimal bounding rectangle corner point	float	128 bits	
Sequential number of this profile in the line	integer	32 bits	

Compared to the tables in Chapter Three, some modifications have been made to the new tables, as follows:

1. Since INGRES lacks the *unsigned* integer type, in order to represent a 64-bit unsigned Morton code listed in Table 3.1, two INGRES *double* precision fields are used.

INGRES *double* type takes 64 bits, and is the closest data type in INGRES for 32-bit unsigned integer. This takes extra eight bytes for a Morton code. However, the algorithms to encode and decode a Morton code are not affected since a Morton code in memory is still stored in a two element array as described in Table 3.1. A Morton code needs to be converted into two double values before written into a INGRES table; these two double values of a Morton code needs to be converted into two unsigned integers after they are read into the memory. Another option would be to convert an unsigned integer into a character string and store the string in INGRES tables. This, however, would definitely slow down the interactive retrieval by the extra CPU time spent on conversion.

2. One extra table (Table 4.1) is needed in the database to bring together all the project names and their corresponding tables.

3. In the INGRES environment, hierarchical indexing structures are represented in tables kept in one database. Each index for directories in Table 4.2 must contain a table name telling the index table of the lower directory or the table of a Morton sequence.

4.3 Range Search Using the INGRES Relational Database

Range searches are performed in a way similar to that used for direct Morton code indexing. The same search algorithms are used, and the search results are hierarchically organized in the vessel, day, line and profile order. The only difference between the range searches in direct Morton code indexing and INGRES Morton code indexing is how the data are retrieved from the index files.

Since query conditions vary with the change of the Morton code values, SELECT statements must be formed and executed during run-time. Dynamic SQL was used to handle this situation [INGRES Manual, 1990].

The basic idea of dynamic SQL is using PREPARE FROM and DESCRIBE INTO statements to obtain the type and size information of an SQL statement. A structure called SQLDA (SQL Descriptor Area) is used to house the information and to pass it into the program. After each FETCH statement retrieves data from the table into a cursor, items of interest can be correctly interpreted using the type and size information for the elements of the table.

Given in Appendix C is an example of using dynamic SQL to retrieve a Morton code from an INGRES table.

4.4 Space Requirement Calculation

The space required for an INGRES table can be calculated in the way described in the INGRES manual [1992]. Three steps are needed to calculate the number of INGRES pages used by a table.

Firstly, the number of rows, *NumRows*, and the width of a row, *RowWidth*, of a table must be known. The latter is the total number of bytes of a row including three bytes overhead.

Secondly, the number of rows per page, *NumRowsPerPage*, is calculated. Each INGRES page is 2048 bytes, of which only 2008 are available for user data. The *NumRowsPerPage* must be rounded down to the nearest integer.

$$NumRowsPerPage = \lfloor 2008 / (RowWidth + 2) \rfloor$$

Finally, the total number of pages, *NumPages*, can be computed by dividing the *NumRows* with *NumRowsPerPage*. The result should be rounded up to the nearest integer.

$$NumPages = \lceil NumRows \ / \ NumRowsPerPage \rceil$$

As an example, take the table of a profile Morton sequence on the line *13:46:06* of the day *1991311*. These 1145 profiles are measured by the vessel *Matthew* for the project *ConceptionBay*. Since four Morton codes are calculated for the MBR corners of each profile, there are 4580 rows in the index table of the Morton sequence, with the width of the row as 43 bytes.

The number of rows per page can be computed as follows,

$$NumRowsPerPage = \lfloor 2008 / (43 + 2) \rfloor = \lfloor 44.6222222222 \rfloor = 44,$$

therefore, the total number of pages needed is

$$NumPages = \lceil 4580 / 44 \rceil = \lceil 104.0909090909 \rceil = 105,$$

or 215,040 bytes.

CHAPTER FIVE

R-TREE SPATIAL INDEXING

5.1 R-tree Structure and Its Variations

Guttman [1984] introduced the R-tree structure for dynamic indexing. This is a spatially organized tree; it reduces over-search by eliminating irrelevant data and checking only data which are close in space to the search area.

The basic idea of the R-tree indexing is using minimal bounding rectangles (MBRs) to enclose objects. According to their location in space, the MBRs are divided into groups which are covered by successively larger MBRs to form a hierarchical data structure. Each MBR belongs to one and only one higher rectangle, even though it may have a part in several other higher ones. A node in this height balanced tree can hold a maximum of M entries. Each entry in its leaf nodes stores the object's MBR and tuple-identification, while an entry at the non-leaf level has its MBR and a pointer to the child. Figure 5.1 is an example with $M = 4$.

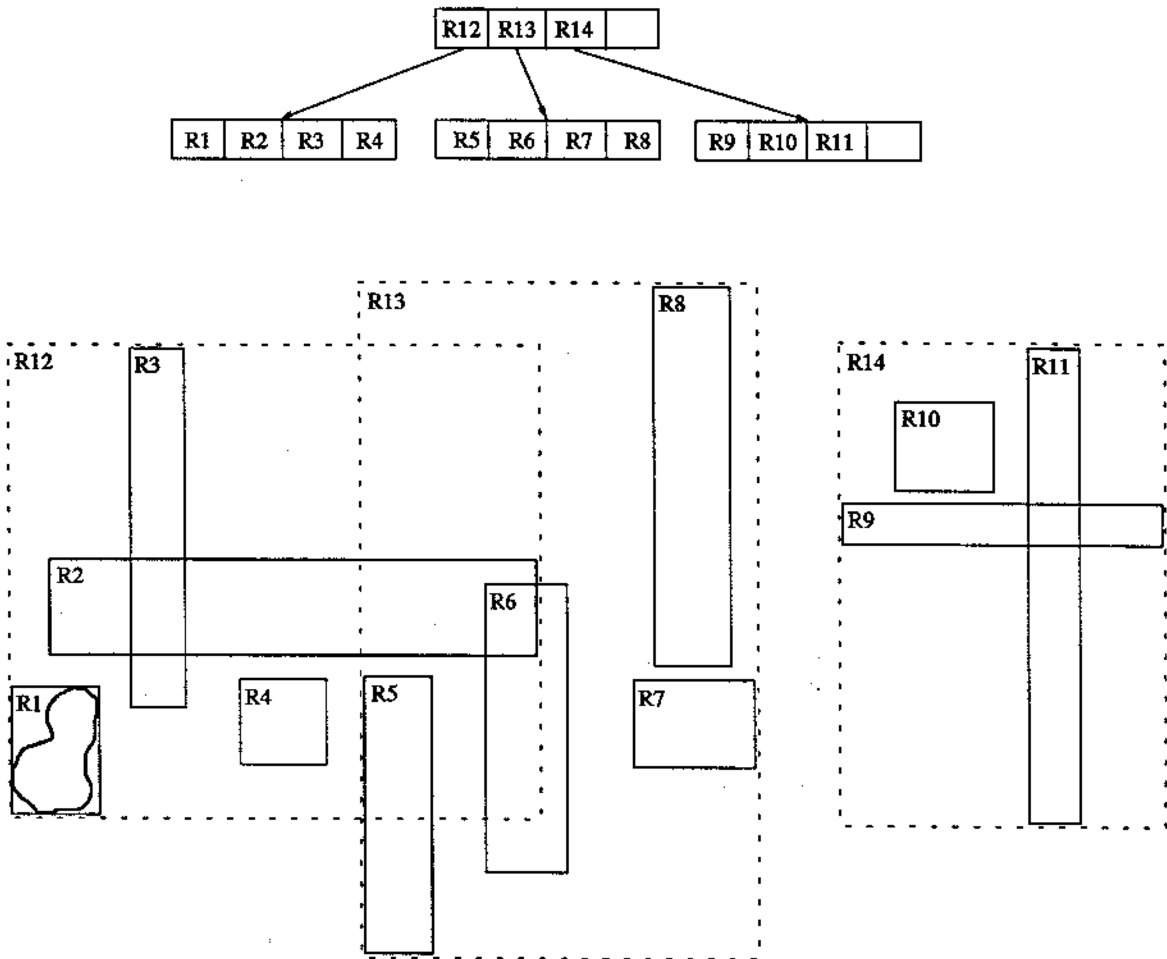


Figure 5.1 An R-tree and the corresponding MBRs.

The R-tree indexing structure has many advantages. Firstly, minimal bounding rectangles give us a rough idea about the range extension 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 full and non-atomic 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. Finally, paging and disk I/O buffering can be handled well using the R-tree structures.

R-trees have been shown to be an efficient dynamic indexing mechanism for spatial objects of non-zero size [Guttman 1984]. The R-tree has *coverage* and *overlay problems* [Roussopoulos and Leifker 1985]. The coverage is defined as the area in a higher rectangle which is not occupied by the MBRs of its descendants. The coverage is the irrelevant space in R-trees. This problem always exists, since not every higher rectangle can be fully filled by the MBRs of its descendants. This will cause a query window, QW1 in Figure 5.2, for example, to test positive in a higher rectangle R14, but all its descendants--R9, R10 and R11--test negative [Noronha 1988]. Guttman also noticed the coverage problem. He tried to make the excess area as small as possible in his *Splitting* algorithms and tried to optimize R-trees in the re-insertion consideration.

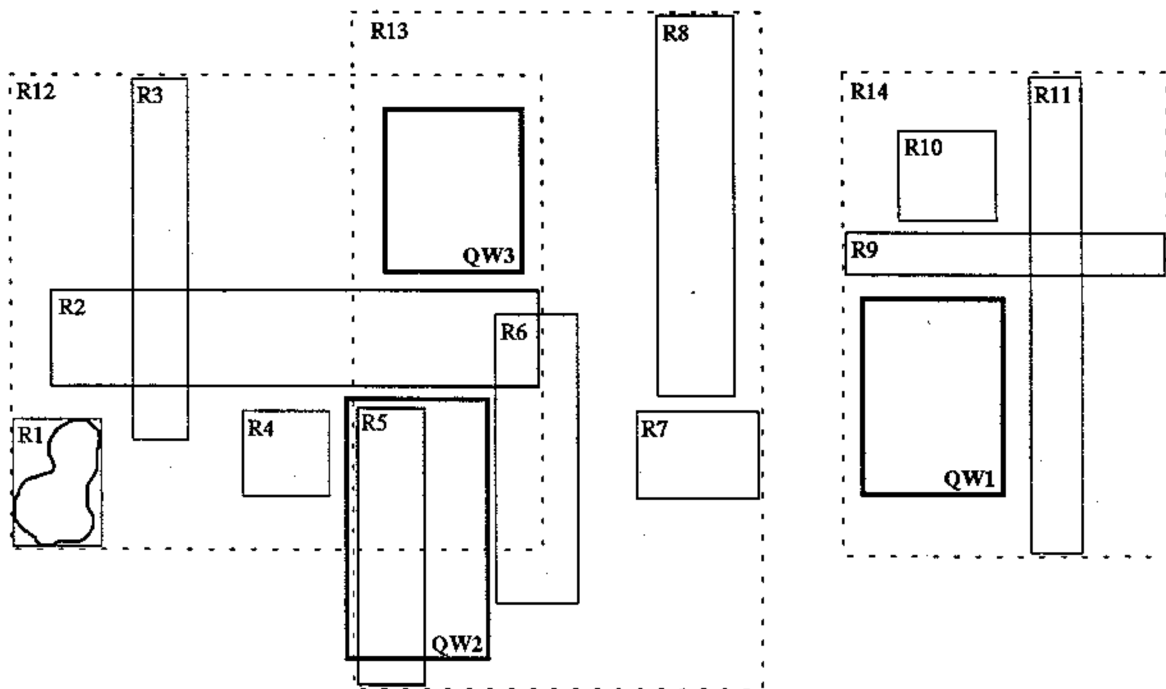


Figure 5.2 The R-tree and its coverage and overlay problems.

Overlap problems arise when MBRs intersect among themselves, as in the case of R12 and R13 in Figure 5.2. A query window in the overlapped area will caused the search algorithm to test all the overlapping rectangles. Overlap problems become critical when they occur in non-leaf nodes. In order to complete the search of QW2, both rectangles R12 and R13 and their descendants are tested. If the query window falls in n overlapping areas, n paths from the root have to be followed [Faloutsos et al., 1987].

The coverage and overlap problems lead to the worst case in searching an R-tree when they happen simultaneously as indicated by the example of QW3 of Figure 5.2. There is no data in the query window, but tests are done through more than one path until the algorithm finally convinces itself of the search failure.

It is not always possible to have zero overlap for polygons in an R-tree [Roussopoulos and Leifker 1985]. If we know the objects before we build the tree, those data can be tightly packed into the tree with minimum overlap and coverage of the leaf nodes. The so-called packed R-tree uses this idea. The nearest neighbours are picked out from the spatial data and are put together into a node. The packed R-tree is suitable to relatively static data which do not submit to frequent insertion and deletion. It "can result in significant savings in space and search time" [Roussopoulos and Leifker 1985].

The R^+-tree is also an improvement of the R-tree. R^+-trees eliminate all overlap by finding the problem object, say R2 in Figure 5.1, and assigning it to more than one neighbouring higher MBR. An R^+-tree is shown in Figure 5.3 where R2 belongs to both R12 and R13. It is easy to see that, for the same objects, an R^+-tree is higher than an R-tree, and the search path will also be longer. Since it is also possible in an R^+-tree to get the same object from different paths, the search algorithm needs some further test steps to

prevent an object being counted more than once. Those two disadvantages, however, are overshadowed by the efficiency of R^+-trees [Faloutsos et al., 1987].

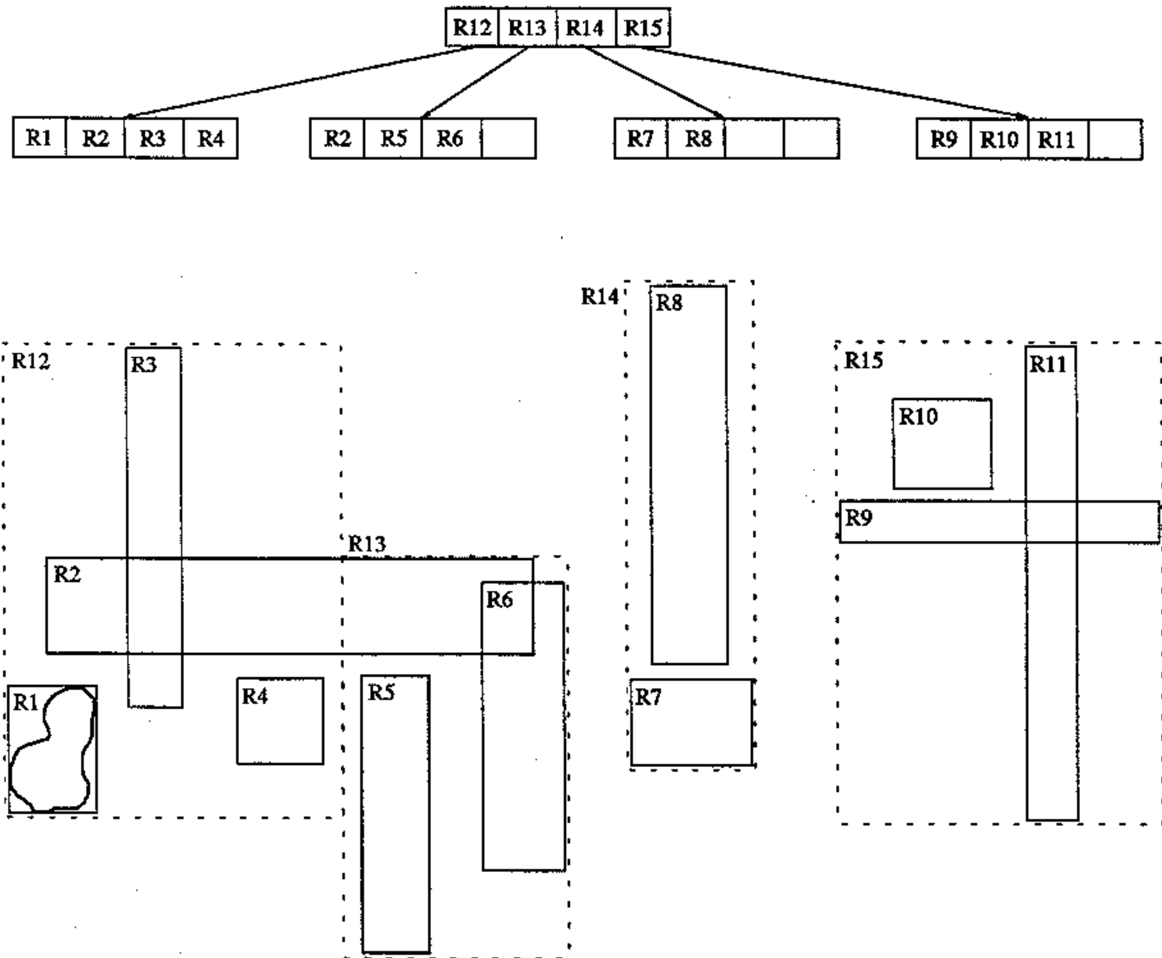


Figure 5.3 An R^+-tree and the corresponding object MBRs.

5.2 R-tree File Structures for HDCS

Packed R-tree structures are used for HDCS spatial indexing since profiles are collected in such a manner that two consecutive profiles are close in space, and those data are not subject to frequent change.

Two types of R-tree indices are built. The first is the tree for profiles, and the second is the tree for directories. In each line directory, an index tree is constructed for all the profiles on the line. The rectangle tightly bounding all the profile MBRs is the minimal bounding rectangle for the line directory. In a day directory an R-tree index is built using the MBRs of all the lines under the day. Similarly, in a vessel directory an R-tree spatial index is constructed for all the days, and in a project directory an index is built for all the vessels, as well. The general picture of R-tree spatial indexing in HDCS can be regarded as a tree-containing-tree structure.

Each file is divided into two parts: file header and file data. The file header is used to store the summary information for the file. Each record in the file data stores the information for one node in the R-tree. Each record contains m minimal bounding rectangles and m pointers. The maximum number of MBR entries in a node is M where the property $m \geq \lceil M/2 \rceil$ holds as for all R-trees of order M. The pointers in the leaf nodes are the addresses of the data tuples, and those in the non-leaf nodes are pointers to the other nodes. An indicator is used to discriminate the type of record in the file data. It has the value of either *l* (for leaf), or *n* (for non-leaf) or *d* (for a deleted record). The root in the tree structure may not be the first record in the file structure, and the file header records the root's byte offset in the file.

All the R-tree index structures for lower directories are identical. Each index consists of two separate files with extension *File1* and *File2*, respectively. They are described as follows:

Table 5.1 Directory spatial index file (File1) header structure.

Item	Type	Size	Units
Size of this header record	int	32 bits	Bytes
Size of a record in this data file	int	32 bits	Bytes
Number of data records in this data file	int	32 bits	
HDCS file type	int	32 bits	
File version number	int	32 bits	
Reference time for this header file	int	32 bits	100 seconds
Data time scale for time offsets in this header file	int	32 bits	No. of microseconds
Minimum time (offset w.r.t. reference time)	int	32 bits	Data time scale
Maximum time (offset w.r.t. reference time)	int	32 bits	Data time scale
Position scale for position offsets in this file	int	32 bits	No. of nano-radians
Reference latitude for this file	int	32 bits	100 nano-radians
Minimum latitude (offset w.r.t. reference latitude)	int	32 bits	100 nano-radians
Maximum latitude (offset w.r.t. reference latitude)	int	32 bits	100 nano-radians
Reference longitude for this file	int	32 bits	100 nano-radians
Minimum longitude (offset w.r.t. reference latitude)	int	32 bits	100 nano-radians
Maximum longitude (offset w.r.t. reference latitude)	int	32 bits	100 nano-radians
Maximum latitude difference for one profile's minimal bounding rectangle	int	32 bits	100 nano-radians
Maximum longitude difference for one profile's minimal bounding rectangle	int	32 bits	100 nano-radians
Maximum number (m) of bounding rectangles in a record	int	32 bits	
Byte offset to the root in this data file	int	32 bits	
Number of deleted records in this data file	int	32 bits	
Byte offset to the first record on the linked list for the deleted records in this data file	int	32 bits	

Table 5.2 Directory spatial index file (File1) data structure for R-tree nodes.

Item	Type	Size	Units
Indicator (l, n, or d) of the data record	char	8 bits	
Byte offset of this record's parent record	int	32 bits	
Number of MBRs in this record, *m*	int	32 bits	
Minimum latitude for MBR 1	int	32 bits	Position scale
Minimum longitude for MBR 1	int	32 bits	Position scale
Maximum latitude for MBR 1	int	32 bits	Position scale
Maximum longitude for MBR 1	int	32 bits	Position scale
Byte offset for child node of MBR 1	int	32 bits	Byte
Minimum latitude for MBR 2	int	32 bits	Position scale
Minimum longitude for MBR 2	int	32 bits	Position scale
Maximum latitude for MBR 2	int	32 bits	Position scale
Maximum longitude for MBR 2	int	32 bits	Position scale
Byte offset for child node of MBR 2	int	32 bits	Byte
. . .			
Minimum latitude for MBR *M*	int	32 bits	Position scale
Minimum longitude for MBR *M*	int	32 bits	Position scale
Maximum latitude for MBR *M*	int	32 bits	Position scale
Maximum longitude for MBR *M*	int	32 bits	Position scale
Byte offset for child node of MBR *M*	int	32 bits	Byte

Table 5.3 Directory spatial index file (File2) header structure.

Item	Type	Size	Units
Size of this header record	int	32 bits	Bytes
Maximum size of the records in this data file	int	32 bits	Bytes
Number of data records in this data file	int	32 bits	
HDCS file type	int	32 bits	
File version number	int	32 bits	
Total number of bytes in this data file	int	32 bits	Bytes

Table 5.4 Directory spatial index file (File2) data structure for R-tree leaf nodes of File1.

Item	Type	Size	Units
Number of characters (*N*) in this data record	int	32 bits	
Directory name of length *N*	char	N bytes	

In *File1*, when the record is not a leaf, its byte offset is the address of another record in *File1*. Every node is a child node, except for the root, which has its "Byte offset of this record's parent record" BO_p set to zero. When the record is a leaf, the byte offset is an address of a record in *File2* which is simply a list of all the sub-directory names. We can regard those directories as tuples we want to build the spatial index on.

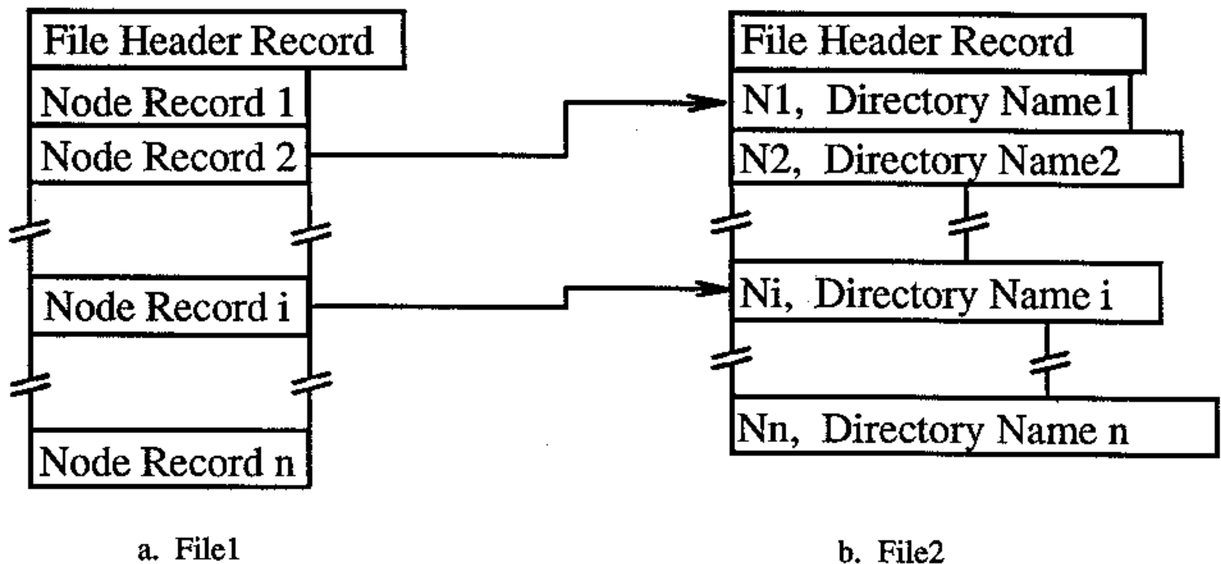


Figure 5.4 Directory spatial index files.

The structure for a spatial index file for profiles is given in Tables 5.5 and 5.6.

As for *File1*, the indicator in the data record also has one of the value: *l*, *n* and *d*. The only difference between *File1* and a profile spatial index is at a leaf node, where the byte offsets become the profile sequential numbers in a line instead of the addresses of records in *File2*.

Table 5.5 Header record structure for the profile spatial index file.

Item	Type	Size	Units
Size of this header record	int	32 bits	Bytes
Size of a record in this data file	int	32 bits	Bytes
Number of data records in this data file	int	32 bits	
HDCS file type	int	32 bits	
File version number	int	32 bits	
Reference time for this header file	int	32 bits	100 seconds
Data time scale for time offsets in this header file	int	32 bits	No. of microseconds
Minimum time (offset w.r.t. reference time)	int	32 bits	Data time scale
Maximum time (offset w.r.t. reference time)	int	32 bits	Data time scale
Position scale for position offsets in this file	int	32 bits	No. of nano-radians
Reference latitude for this file	int	32 bits	100 nano-radians
Minimum latitude (offset w.r.t. reference latitude)	int	32 bits	100 nano-radians
Maximum latitude (offset w.r.t. reference latitude)	int	32 bits	100 nano-radians
Reference longitude for this file	int	32 bits	100 nano-radians
Minimum longitude (offset w.r.t. reference latitude)	int	32 bits	100 nano-radians
Maximum longitude (offset w.r.t. reference latitude)	int	32 bits	100 nano-radians
Maximum latitude difference for one profile's minimal bounding rectangle	int	32 bits	100 nano-radians
Maximum longitude difference for one profile's minimal bounding rectangle	int	32 bits	100 nano-radians
Maximum number (m) of bounding rectangles in a record	int	32 bits	
Byte offset to the root in this data file	int	32 bits	
Number of deleted records in this data file	int	32 bits	
Byte offset to the first record on the linked list for the deleted records in this data file	int	32 bits	

Table 5.6 Structure of one record of the profile spatial index file.

Item	Type	Size	Units
Indicator of the data record	char	8 bits	
Byte offset of this record's parent record	int	32 bits	
Number of MBRs in this record	int	32 bits	
Minimum latitude for MBR 1	int	32 bits	Position scale
Minimum longitude for MBR 1	int	32 bits	Position scale
Maximum latitude for MBR 1	int	32 bits	Position scale
Maximum longitude for MBR 1	int	32 bits	Position scale
Byte offset for MBR 1	int	32 bits	
Minimum latitude for MBR 2	int	32 bits	Position scale
Minimum longitude for MBR 2	int	32 bits	Position scale
Maximum latitude for MBR 2	int	32 bits	Position scale
Maximum longitude for MBR 2	int	32 bits	Position scale
Byte offset for MBR 2	int	32 bits	
. . .			
Minimum latitude for MBR *m*	int	32 bits	Position scale
Minimum longitude for MBR *m*	int	32 bits	Position scale
Maximum latitude for MBR *m*	int	32 bits	Position scale
Maximum longitude for MBR *m*	int	32 bits	Position scale
Byte offset for MBR *m*	int	32 bits	

5.3 Building R-tree Indices

Index files are built using a combination of top-down and bottom-up approaches. The program starts from a project directory, visits each vessel directory under the project, then each day under the vessels, and finally all lines under each day. An index file is created for all the profiles on this line, and an MBR for all the profiles is also established. This MBR is the bounding rectangle for the data of this line. After all the line MBRs are found, index files can be created for those lines. Minimal bounding rectangles are propagated up to the project directory.

Guttman's algorithms are adopted after some modification. The modified algorithms take into account the difference in the data structures, file operations, as well as the deleted record collection which will be introduced in Section 5.5.

An R-tree is built up by inserting new rectangles into its leaf nodes. Coverage and overlay problems should be minimized when adding new rectangles. The right leaf node is determined by traversing the tree from the root to one of its leaves while at each node choosing the sub-tree whose MBR would have to be enlarged the least to enclose the new rectangle.

If there are M minimal bounding rectangles in the leaf node in which the new MBR is about to be added, it has to be split. The splitting propagates up till a non-leaf node with less than M rectangles is reached. Two rectangles with the greatest normalized separations along all the dimensions among the $M + 1$ rectangles are found and used as the first MBRs in the two split nodes. The rest of the MBRs are added arbitrarily into one of the two nodes which would be enlarged the least by the addition.

5.4 Range Search Using R-tree Indices

Given a project name and a query window, a range search starts from a project directory, and searches the R-tree index files in the hierarchically structured directories. Searching an R-tree is straightforward. The MBRs at each node are compared with the given range when traversing through the index tree starting from its root. If the QW encloses an MBR, all the MBRs of the sub-tree are within the query range, so that a further test can be skipped.

A sub-directory name can be obtained from *File2* whose address in the file is acquired from a leaf node of *File1*. Then the search program moves to the lower directory for the further search. The tuple-IDs in profile spatial index files whose MBRs are within the search range yields the search results: the in-window profile numbers.

5.5 Linking Deleted Records in R-trees

The fundamental operations for dynamic index file updating are inserting and deleting records. To avoid moving massive data within the file, deleted records are linked into a list through the last byte offset of the record. A pointer in the file header records the first record of the list, and is set to a negative value if the list is empty.

The linked list approach makes it easy to access all the deleted records and makes it flexible to dynamically update R-tree indices. A newly deleted record can be easily attached to the list without moving other records and modifying the pointers to the records. When a new record is about to be added into the file, the linked list is checked first. If the list is not empty, the first record on the linked list is released and the disk space is reused; otherwise the new record is appended to the end of the file.

5.6 Deleting Profiles From R-tree Indices

Whenever some profiles are deleted from the data set, the R-tree spatial index files based on this data set should be modified by deleting profile entries in the R-trees.

Deleting a profile from an R-tree is performed in the following four phases:

1. The profile's MBR is used to find the leaf-node containing the profile. Then the leaf is searched to find the entry whose tuple-ID matches the profile number.

2. Remove the entry from the leaf.

3. If the number of entries in the node is less than *M/2*, keep the entries in a list for re-insertion, and delete this node from the tree. Otherwise tighten the MBR for the node. Move up to the parent of the node and repeat the operations in this phase on the parent until it is a node. Re-insert all the entries in the list into the tree.

4. If the root has only one child, delete the root. Set the root pointer in the file header to the old root's child.

Suppose some profiles, defined by the range of profile sequential numbers, are removed from a line. To modify the indices, the line's profile spatial index file must be changed first. Then the deletion propagates up to the corresponding day, vessel and project directories, and the directory spatial index files under these directories are modified accordingly. Changing each directory R-tree index file (*File1*) involves the same steps mentioned above. If a whole directory is removed, a record in *File2* is deleted by changing the directory name length *N* to a negative value.

5.7 Modified Deletion Algorithm

A profile spatial index R-tree is well packed. The profiles close in space are close in the tree. They are stored in the same leaf-node or in the leaf-nodes of common

ancestors. An R-tree for the first 15 profiles on line 16:57:19, under project *ConceptionBay*, vessel *Matthew*, date *199134*, is shown in Figure 5.5.

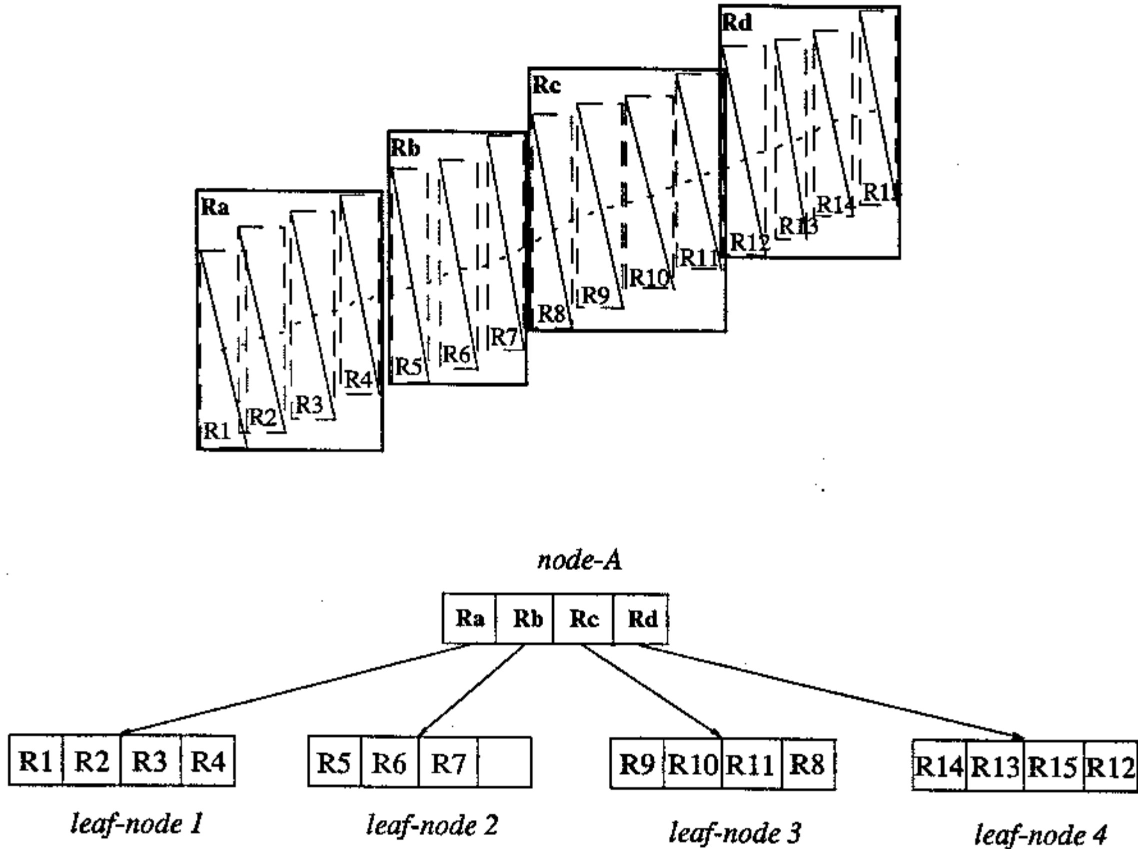


Figure 5.5 A simple R-tree for profile MBRs.

The deletion algorithm works in a profile-by-profile approach. Each time a deletion is requested, a search for the profile from the root to the leaf and a tree adjustment from the leaf to the root have to be performed. Since most of the deletions in HDCS are caused by removing a range of profiles, say from profile 100 to profile 500, by removing all the should-be-removed profiles from a leaf-node or the leaf nodes of the same close ancestors, a deletion can avoid substantial amounts of the search and adjustment.

The modified deletion algorithm takes into account that the profiles are packed into an R-tree. It finds the first profile in a leaf-node, deletes all the node's profiles which are within the profile range. Then it climbs up a limited level of the tree, searches in the siblings of the node just visited, and deletes all the profiles in the range.

A linked list, called *AvoidList*, is used to avoid repeatedly deleting a profile. It keeps the profile numbers being deleted, and changes as more profiles are deleted. In order to speed up searching the list, when a profile number is found on the list, the profile will be removed. Suppose the current to-be-deleted profile number is i, therefore all the profile numbers on the list are no less than i. The modified deletion algorithms are given in Appendix D.

For example, suppose we wish to delete profiles 4-14 from the R-tree in Figure 5.5. The deletion starts with profile 4, finds *leaf-node 1* containing an entry *R4* for the profile, and deletes *R4*. All other entries in *leaf-node 1* are checked for deletion, but no more removal occurs in the node. Then, the R-tree is condensed. The next profile to be deleted is profile 5 which has an entry *R5* in *leaf-node 2*. After *R5* is deleted, other entries in the node are checked. Evidently, entries *R6* and *R7* for profiles 6 and 7, respectively, are within the deletion range, and they are deleted from *leaf-node 2*. Now *leaf-node 2* is empty, and it is added into the deleted record linked list in the R-tree. After that function DeleteUpwards is called to delete *leaf-node 2* from its parent *node-A* and to delete the profiles 8-14 from *leaf-node 3* and *leaf-node 4*. Finally the R-tree is condensed starting from *leaf-node 4*.

The function **DeleteUpwards** first deletes *leaf-node 1* from *node-A*, then goes to *leaf-node 2*'s first sibling on the right, i.e. *leaf-node 3*. Entries for profiles 8-11 are deleted from *leaf-node 3*, and the deletion leaves the node empty. Again, **DeleteUpwards** is called

to delete *leaf-node 3* from the *node-A* and to remove the profiles in the first right sibling of *leaf-node 3*. Since only entry *R14* can be deleted from *leaf-node 4* and the node is not empty after the removal of *R14*, the deletion stops.

CHAPTER SIX

COMPARISON OF THE INDEXING METHODS

6.1 Experimental Description

Experiments for the Morton code, R-tree and INGRES Morton code indexing methods were conducted using data surveyed by the vessel *Matthew* for the Canadian Hydrographic Service in 1991. The area surveyed was in Conception Bay off the north-east coast of Newfoundland, Canada. Figure 6.1 shows a map of the surveyed area. The range covered by the survey is

minLat = 0.829973	(47°33′14″N)	minLong = -0.926763	(53°05′59″W)
maxLat = 0.831002	(47°36′46″N)	maxLong = -0.925504	(53°01′39″W)

Spanning four days, the survey includes 46 lines. The total number of profiles collected is 54,192, each of which contains 32 sounding points. The total amount of data is about 128.9 megabytes. Appendix E gives the numbers of projects, days, lines, profiles, and the data size of each line.

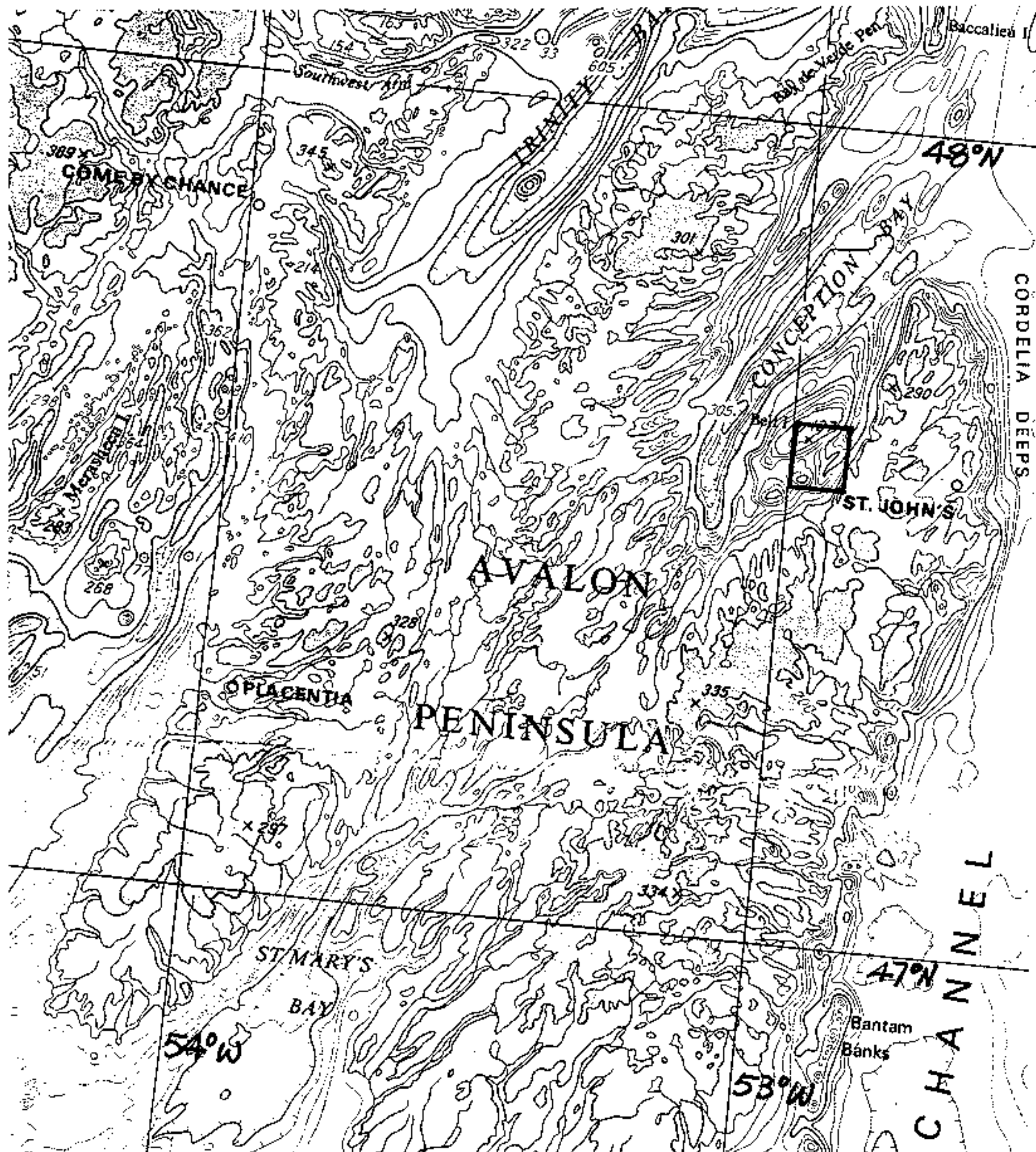


Figure 6.1 Surveyed area in Conception Bay off the north-east coast of Newfoundland. [from Canadian Hydrographic Chart 802, Newfoundland Shelf]

The programs for building and searching the indices were developed on a Sun workstation called *jupiter*, a model Sparc 670MP running Sun OS 4.1.3, with 128 MB main memory and 18GB of hard disk, in the Computing Services Department, University of New Brunswick.

The time for building and searching spatial indices using the three approaches was measured. R-trees were constructed using four different branch factors from M=4 to M=7. Range searches on the Morton code and R-tree indices were performed using thirteen query windows of different sizes. The windows, listed in Table 6.1, were chosen based on the percentage of profiles returned in the search results.

Table 6.1 Query windows and their coverage.

QW	Min. Lat.	Min. Long.	Max. Lat.	Max. Long.	Coverage (%)
1	0.83060	-0.9260	0.83060	-0.9260	0
2	0.83064	-0.9260	0.83065	-0.9259	5
3	0.83064	-0.9260	0.83070	-0.9259	11
4	0.83060	-0.9260	0.83070	-0.9259	15
5	0.83040	-0.9264	0.83060	-0.9260	21
6	0.83040	-0.9266	0.83060	-0.9259	29
7	0.82970	-0.9266	0.83070	-0.9260	40
8	0.82970	-0.9264	0.83060	-0.9259	50
9	0.82970	-0.9264	0.83090	-0.9259	60
10	0.83020	-0.9267	0.83060	-0.9256	70
11	0.83020	-0.9267	0.83070	-0.9255	80
12	0.82990	-0.9266	0.83090	-0.9257	90
13	0.82990	-0.9268	0.83110	-0.9255	100

An index file was built ten times, and each query window was also searched ten times. The averages for the building and searching were calculated, along with their standard deviation. In order to minimize the effect of other processes running on the

machine during the testing, the experiment was designed in such a way that the building and searching of Morton code and R-tree indices were interleaved as described below:

```
Loop 10 times
        Construct Morton code indices;
        Search the Morton code indices against the QW_j (j = 1, 13);
        Construct R-tree indices for M=4;
        Search the R-tree indices against the QW_j (j = 1, 13);
        Construct R-tree indices for M=5;
        Search the R-tree indices against the QW_j (j = 1, 13);
        Construct R-tree indices for M=6;
        Search the R-tree indices against the QW_j (j = 1, 13);
        Construct R-tree indices for M=7;
        Search the R-tree indices against the QW_j (j = 1, 13);
EndLoop
```

Figure 6.2 Algorithm for Morton code and R-tree index build and search experiments.

The timing results for building and searching the Morton code and R-tree indexing files are listed in Tables 6.3 - 6.8, and the file space requirements of the two methods are listed in Table 6.12.

Seven profile ranges (see Table 6.2) were used to test Guttman's deletion algorithm and the modified algorithm. Line 16:44:37 of day 1991314 was used in the experiment. There are 1076 profiles in this line. A deletion range is chosen according to the percentage of profiles it covers on the line, and is defined by two numbers giving the starting profile and the ending profile for the deletion.

Deletion timing was also taken ten times for each range using the two algorithms on the R-tree indices with the branch factor M equal to 4. Tables 6.9 and 6.10 list the timing results obtained using the following algorithm:

Table 6.2 Profile ranges to delete for profile deletion experiment.

Range	Start No.	End No.	Coverage (%)
1	0	0	0
2	500	608	10
3	400	616	20
4	300	732	40
5	200	848	60
6	100	964	80
7	1	1076	100

Loop 10 times

For each range R_j, j = 1, 7, loop

Build R-tree indices;

Delete R_j using Guttman's algorithm

Build R-tree indices;

Delete R_j using modified algorithm

EndLoop

EndLoop

Figure 6.3. Algorithm for range deletion experiment.

INGRES Morton code indices were built under the INGRES environment. The same query windows listed in Table 6.1 were used to test the search speed. The time required for building and searching using INGRES tables was very long, so the timing was taken only once and the results are listed in Table 6.11.

6.2 Times for Building Morton Code and R-tree Spatial Indices

The time used in building both the Morton code and R-tree index files was measured in the manner described in Section 6.1 between 9 AM and 7 PM, October 30, 1993. A Saturday was chosen since this is the day when the number of users is the least in a week.

The timing results in building R-trees varies with the branch factor M. The fastest building performance was obtained when M=7, and the slowest one when M=4, which have average execution times of 8.86 seconds and 11.16 seconds, respectively. As illustrated in Table 6.3 and Figure 6.4, there is no significant difference among the timings when M is 5, 6, and 7.

It is shown in Table 6.3 that building Morton code indices is almost two to three times faster than constructing R-tree indices.

Table 6.3 Times (in seconds) for building Morton code and R-tree indices.

	MC	**RT (M=4)**	**RT (M=5)**	**RT (M=6)**	**RT (M=7)**
1	4.32	10.65	8.70	10.83	7.82
2	3.27	10.92	10.62	9.88	8.67
3	3.50	10.08	10.05	9.07	8.67
4	5.40	11.80	10.88	12.15	11.53
5	4.32	13.38	9.33	9.02	9.67
6	3.68	10.52	8.33	11.22	9.00
7	3.55	11.78	9.35	8.78	8.62
8	3.48	11.00	8.03	8.42	7.17
9	3.62	11.62	9.30	8.02	7.72
10	3.20	9.85	7.90	8.60	9.68
Average	**3.83**	**11.16**	**9.25**	**9.60**	**8.86**
Standard Deviation	0.67	1.03	1.04	1.37	1.24

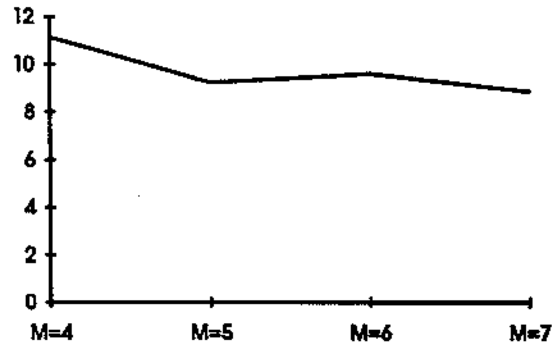


Figure 6.4 Times for building R-tree indices for HDCS using different branch factors.

6.3 Times for Searching Morton Code and R-tree Spatial Indices

The distribution of data, the position and the sizes of the query windows are three major factors that affect the search time if the other conditions are the same.

The times for searching Morton code and R-tree indices were taken in the same period as measuring the times for building the indices. Thirteen query windows of different sizes were used, and each query window was searched ten times against Morton code indices and R-tree indices of M= 4, 5, 6, and 7, respectively. The timing results, their average (AV) and the standard deviations (SD) are listed in Tables 6.4 - 6.8. The averages are used in Figure 6.5.

The figures in the tables display that the time for searching R-tree indices is very consistent. The search time on an R-tree index does not change substantially with the size and the location of the query windows. The query windows QW1 - QW4 use about 0.01 seconds and the query windows QW5 - QW13 use about 0.08 seconds. The search time

consistency holds for the R-trees of different branch factors. The change of branch factors affects the search time slightly, as shown in Figure 6.5, with M=4 giving the fastest times.

Table 6.4 Times (in seconds) for searching Morton code indices.

	QW 1	**QW 2**	**QW 3**	**QW 4**	**QW 5**	**QW 6**	**QW 7**	**QW 8**	**QW 9**	**QW 10**	**QW 11**	**QW 12**	**QW 13**
1	0.03	0.00	0.00	0.03	0.25	0.30	0.47	0.62	0.70	0.65	0.43	0.45	0.60
2	0.05	0.02	0.02	0.02	0.20	0.22	0.33	0.42	0.45	0.37	0.40	0.50	0.62
3	0.05	0.02	0.02	0.02	0.17	0.20	0.48	0.43	0.52	0.52	0.37	0.43	0.60
4	0.05	0.02	0.02	0.03	0.23	0.25	0.42	0.68	0.45	0.50	0.65	0.67	0.78
5	0.07	0.02	0.02	0.02	0.18	0.27	0.40	0.33	0.57	0.47	0.48	0.87	0.90
6	0.05	0.02	0.03	0.02	0.17	0.20	0.33	0.28	0.47	0.38	0.37	0.45	0.62
7	0.05	0.03	0.07	0.05	0.25	0.37	0.32	0.38	0.52	0.40	0.45	0.45	0.58
8	0.07	0.00	0.02	0.03	0.30	0.25	0.83	0.37	0.40	0.38	0.33	0.52	0.43
9	0.05	0.03	0.03	0.03	0.20	0.20	0.28	0.35	0.60	0.45	0.38	0.43	0.47
10	0.03	0.02	0.02	0.02	0.13	0.20	0.35	0.28	0.78	0.35	0.35	0.53	0.65
AV	**0.05**	**0.02**	**0.03**	**0.03**	**0.21**	**0.25**	**0.42**	**0.41**	**0.55**	**0.45**	**0.42**	**0.53**	**0.63**
SD	0.01	0.01	0.02	0.01	0.05	0.06	0.16	0.13	0.12	0.09	0.09	0.14	0.14

Table 6.5 Times (in seconds) for searching R-tree (M=4) indices.

	QW 1	**QW 2**	**QW 3**	**QW 4**	**QW 5**	**QW 6**	**QW 7**	**QW 8**	**QW 9**	**QW 10**	**QW 11**	**QW 12**	**QW 13**
1	0.02	0.00	0.00	0.02	0.05	0.05	0.12	0.07	0.05	0.07	0.05	0.12	0.08
2	0.05	0.00	0.00	0.02	0.08	0.13	0.13	0.08	0.08	0.08	0.08	0.10	0.10
3	0.02	0.00	0.00	0.00	0.10	0.10	0.08	0.07	0.10	0.13	0.17	0.10	0.08
4	0.03	0.00	0.02	0.00	0.08	0.08	0.12	0.10	0.12	0.15	0.13	0.32	0.20
5	0.02	0.00	0.00	0.00	0.07	0.07	0.13	0.07	0.13	0.10	0.08	0.08	0.10
6	0.03	0.00	0.00	0.00	0.07	0.08	0.10	0.10	0.08	0.12	0.10	0.18	0.18
7	0.02	0.00	0.00	0.00	0.10	0.08	0.15	0.07	0.08	0.07	0.10	0.10	0.12
8	0.03	0.00	0.00	0.00	0.08	0.10	0.10	0.13	0.10	0.13	0.10	0.10	0.13
9	0.02	0.02	0.00	0.00	0.07	0.10	0.07	0.07	0.07	0.13	0.12	0.10	0.10
10	0.03	0.00	0.00	0.02	0.07	0.08	0.10	0.10	0.10	0.10	0.10	0.10	0.12
AV	**0.03**	**0.00**	**0.00**	**0.01**	**0.08**	**0.09**	**0.11**	**0.09**	**0.09**	**0.11**	**0.10**	**0.12**	**0.12**
SD	0.01	0.01	0.01	0.01	0.01	0.02	0.02	0.02	0.02	0.03	0.03	0.08	0.04

Table 6.6 Times (in seconds) for searching R-tree (M=5) indices.

	QW 1	QW 2	QW 3	QW 4	QW 5	QW 6	QW 7	QW 8	QW 9	QW 10	QW 11	QW 12	QW 13
1	0.03	0.00	0.00	0.02	0.07	0.08	0.10	0.07	0.07	0.08	0.07	0.08	0.08
2	0.03	0.00	0.00	0.02	0.05	0.07	0.07	0.05	0.08	0.08	0.07	0.18	0.12
3	0.03	0.00	0.00	0.00	0.08	0.05	0.07	0.07	0.08	0.13	0.12	0.08	0.07
4	0.02	0.00	0.00	0.02	0.07	0.07	0.08	0.12	0.13	0.10	0.07	0.12	0.10
5	0.02	0.00	0.00	0.00	0.07	0.05	0.05	0.05	0.03	0.03	0.05	0.05	0.03
6	0.02	0.00	0.00	0.00	0.05	0.07	0.08	0.12	0.08	0.08	0.07	0.08	0.10
7	0.02	0.02	0.00	0.00	0.05	0.07	0.08	0.05	0.08	0.05	0.10	0.08	0.07
8	0.02	0.02	0.00	0.00	0.10	0.08	0.07	0.05	0.08	0.08	0.05	0.07	0.03
9	0.02	0.00	0.02	0.00	0.07	0.07	0.07	0.03	0.07	0.08	0.05	0.05	0.07
10	0.02	0.00	0.00	0.00	0.05	0.12	0.10	0.05	0.05	0.05	0.05	0.07	0.10
AV	**0.02**	**0.00**	**0.00**	**0.01**	**0.07**	**0.07**	**0.08**	**0.07**	**0.08**	**0.08**	**0.07**	**0.09**	**0.08**
SD	0.00	0.01	0.01	0.01	0.02	0.02	0.01	0.03	0.03	0.03	0.02	0.04	0.03

Table 6.7 Times (in seconds) for searching R-tree (M=6) indices.

	QW 1	QW 2	QW 3	QW 4	QW 5	QW 6	QW 7	QW 8	QW 9	QW 10	QW 11	QW 12	QW 13
1	0.02	0.00	0.00	0.00	0.05	0.03	0.05	0.03	0.05	0.08	0.08	0.08	0.08
2	0.03	0.00	0.00	0.00	0.08	0.07	0.08	0.08	0.10	0.08	0.10	0.08	0.17
3	0.02	0.02	0.00	0.00	0.07	0.07	0.07	0.07	0.08	0.07	0.07	0.08	0.07
4	0.02	0.02	0.00	0.02	0.07	0.07	0.08	0.07	0.17	0.12	0.13	0.08	0.07
5	0.03	0.00	0.02	0.02	0.05	0.10	0.08	0.08	0.12	0.10	0.07	0.08	0.07
6	0.02	0.02	0.00	0.00	0.10	0.07	0.13	0.08	0.08	0.08	0.10	0.13	0.10
7	0.02	0.02	0.00	0.00	0.07	0.07	0.08	0.05	0.07	0.07	0.07	0.08	0.08
8	0.02	0.00	0.00	0.02	0.05	0.05	0.07	0.07	0.05	0.20	0.10	0.12	0.18
9	0.02	0.00	0.00	0.00	0.05	0.07	0.07	0.07	0.12	0.10	0.12	0.13	0.05
10	0.02	0.02	0.00	0.00	0.07	0.05	0.07	0.07	0.08	0.07	0.07	0.08	0.08
AV	**0.02**	**0.01**	**0.00**	**0.01**	**0.07**	**0.07**	**0.08**	**0.07**	**0.09**	**0.10**	**0.09**	**0.09**	**0.10**
SD	0.00	0.01	0.01	0.01	0.02	0.02	0.02	0.02	0.04	0.04	0.02	0.02	0.04

Table 6.8 Times (in seconds) for searching R-tree (M=7) indices.

	QW 1	QW 2	QW 3	QW 4	QW 5	QW 6	QW 7	QW 8	QW 9	QW 10	QW 11	QW 12	QW 13
1	0.02	0.02	0.00	0.02	0.05	0.05	0.07	0.05	0.07	0.07	0.07	0.07	0.12
2	0.03	0.02	0.00	0.02	0.05	0.12	0.12	0.10	0.13	0.12	0.08	0.08	0.07
3	0.02	0.02	0.00	0.02	0.10	0.07	0.08	0.08	0.08	0.12	0.12	0.12	0.08
4	0.02	0.00	0.00	0.02	0.07	0.08	0.10	0.10	0.08	0.07	0.07	0.05	0.07
5	0.03	0.02	0.00	0.02	0.07	0.07	0.07	0.08	0.08	0.07	0.07	0.07	0.07
6	0.02	0.00	0.00	0.00	0.08	0.03	0.07	0.05	0.08	0.07	0.10	0.05	0.05
7	0.02	0.00	0.00	0.00	0.07	0.07	0.05	0.07	0.05	0.05	0.10	0.07	0.05
8	0.03	0.00	0.00	0.02	0.05	0.02	0.05	0.07	0.08	0.07	0.07	0.08	0.13
9	0.02	0.02	0.02	0.02	0.08	0.12	0.10	0.05	0.05	0.05	0.05	0.03	0.03
10	0.02	0.00	0.02	0.00	0.08	0.08	0.07	0.08	0.07	0.05	0.05	0.05	0.05
AV	**0.02**	**0.01**	**0.00**	**0.01**	**0.07**	**0.07**	**0.08**	**0.07**	**0.08**	**0.07**	**0.08**	**0.07**	**0.07**
SD	0.00	0.01	0.01	0.01	0.02	0.03	0.02	0.02	0.02	0.03	0.02	0.02	0.03

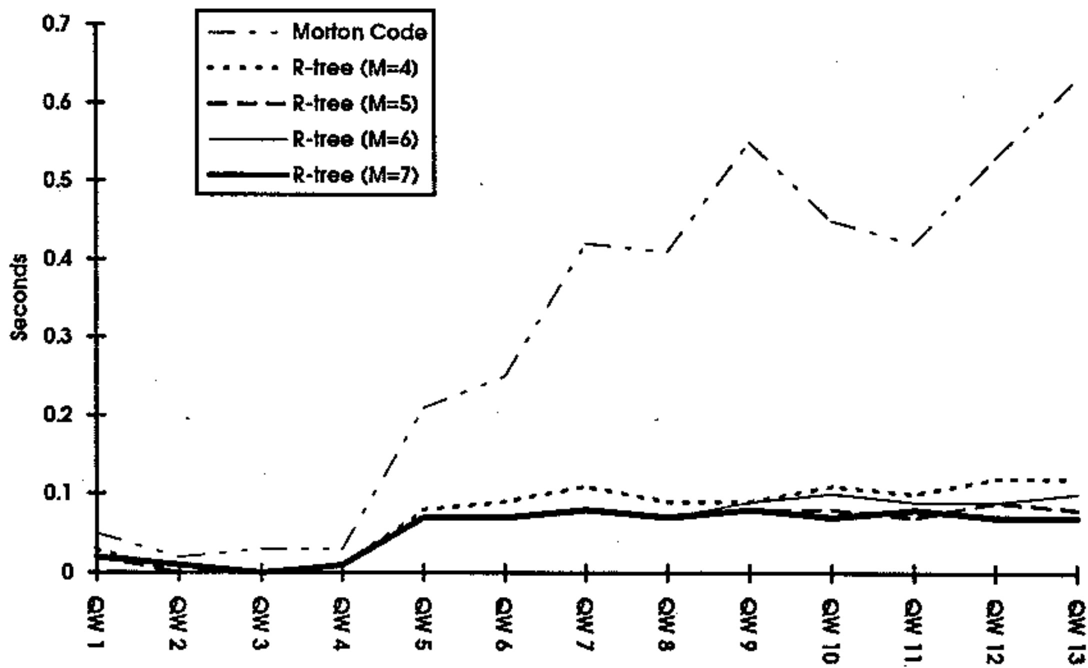


Figure 6.5 Times for searching Morton code and R-tree indices in HDCS.

Generally, the larger the query window is, the longer the search takes on a Morton code spatial index file. The search time, however, also varies with the change of the

window's locations. QW10 and QW11 give better search speeds because of their location in the data set.

6.4 Times For Deleting Profiles from R-tree Spatial Indices

R-tree indices with a branch factor M equal to 4 are used in timing the Guttman's deletion algorithm and the modified deletion algorithm. All the deletions take place on one line: 16:44:37 of day 1991314. Seven deletion ranges covering different percentages of profiles on the line were used (see Table 6.2).

The experiment started with no profile being deleted, and ended with the whole line being removed. For each range, ten timing results were recorded on each algorithm, and are listed in Tables 6.10 and 6.11, as well as their averages and standard deviations. The timing averages are used in Figure 6.6.

Table 6.9 Times (in seconds) for deleting R-tree indices using Guttman's algorithm.

(M=4)	Range 1	Range 2	Range 3	Range 4	Range 5	Range 6	Range 7
1	0.00	0.23	0.43	0.88	0.93	1.82	1.57
2	0.00	0.15	0.35	0.68	1.47	1.52	1.97
3	0.00	0.28	0.37	0.93	1.22	2.05	2.15
4	0.00	0.07	0.28	0.93	1.57	1.65	1.80
5	0.00	0.10	0.27	0.77	1.07	1.97	1.75
6	0.00	0.08	0.37	1.03	1.45	1.95	1.78
7	0.00	0.17	0.33	0.80	1.05	1.90	1.98
8	0.00	0.12	0.43	0.77	1.08	1.88	1.73
9	0.00	0.10	0.25	0.78	1.02	1.88	2.45
10	0.00	0.08	1.27	1.35	1.13	3.17	3.15
AV	0.00	0.14	0.44	0.89	1.20	1.98	2.03
SD	0.00	0.07	0.30	0.19	0.22	0.45	0.46

Table 6.10 Times (in seconds) for deleting R-tree indices using modified algorithm.

(M=4)	Range 1	Range 2	Range 3	Range 4	Range 5	Range 6	Range 7
1	0.00	0.00	0.02	0.03	0.08	0.18	0.26
2	0.00	0.02	0.02	0.02	0.07	0.17	0.24
3	0.02	0.02	0.00	0.02	0.07	0.14	0.28
4	0.00	0.00	0.02	0.02	0.06	0.16	0.28
5	0.00	0.02	0.02	0.02	0.06	0.18	0.21
6	0.00	0.00	0.00	0.02	0.08	0.11	0.19
7	0.00	0.00	0.02	0.03	0.09	0.20	0.20
8	0.00	0.00	0.02	0.02	0.08	0.22	0.23
9	0.00	0.00	0.02	0.02	0.09	0.20	0.21
10	0.00	0.00	0.02	0.07	0.12	0.29	0.30
AV	0.00	0.01	0.02	0.03	0.08	0.19	0.24
SD	0.01	0.01	0.01	0.02	0.02	0.05	0.04

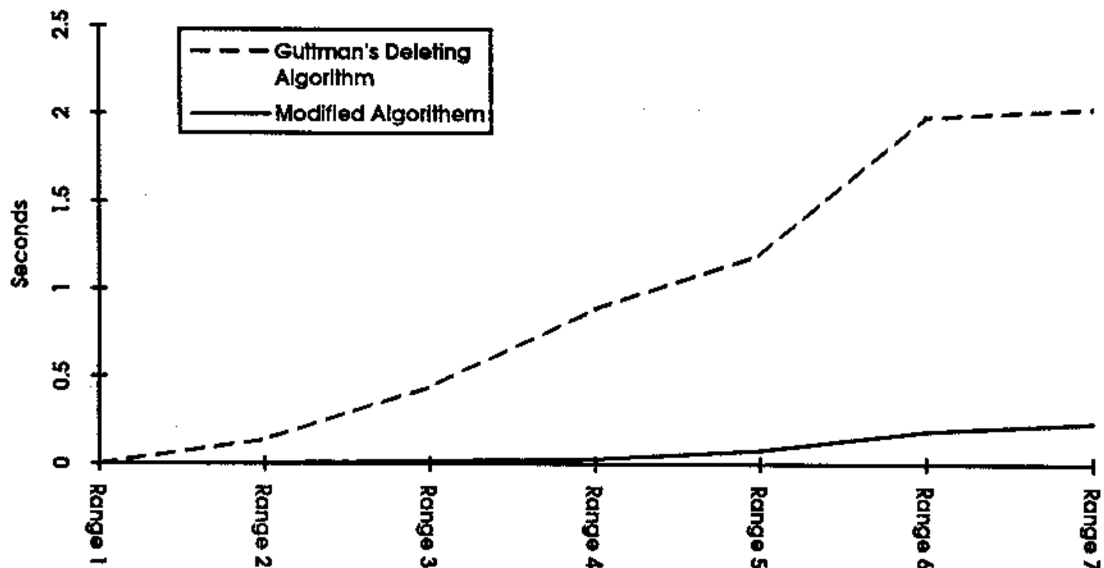


Figure 6.6 Average time used in the deletion of different ranges.

The time used by the modified deletion algorithm increases slightly with the enlarging of the profile deletion ranges. On the contrary, the deletion time of Guttman's algorithm increases rapidly, and the time used by Guttman's algorithm turns out to be 10 to 20 times that used by the modified algorithm.

6.5 Times for Building and Searching INGRES Morton Code Indices

The time used to build and search INGRES Morton code indices is very high. Building indices for HDCS took 121.32 CPU seconds. The timing results for the 13 QWs are listed in Table 6.11.

Table 6.11 Times (in seconds) for searching INGRES Morton code indices.

Query Window	Time
QW1	3.95
QW2	8.95
QW3	22.07
QW4	16.32
QW5	99.28
QW6	197.63
QW7	595.50
QW8	880.72
QW9	1183.05
QW10	1127.55
QW11	1046.65
QW12	1715.65
QW13	1853.72

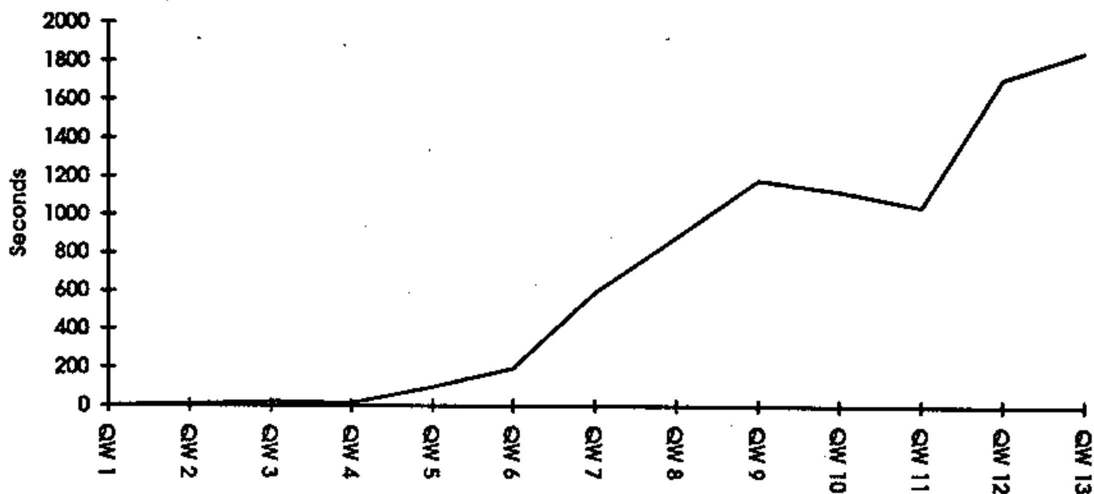


Figure 6.7 Times used in searching INGRES Morton code indices.

6.6 Space Requirement of the Three Spatial Indexing Approaches

The sizes of index files for directories or profiles change with the number of directories or profiles being indexed. The more the directories or profiles, the larger the index file sizes. Since the directory numbers are far less than the profile numbers in HDCS, the size of a directory spatial index file is far smaller than the size of a profile spatial index file. For example, under Morton code indexing, the directory spatial index file for vessels under project ConceptionBay has only one vessel, *Matthew*, and takes 224 bytes, while the profile spatial index file under day 1991314 and line 16:44:37 contains 1076 profiles and uses 51708 bytes. Because of this difference, only the sizes of profile spatial index files are considered here.

While the size of Morton code and R-tree spatial index files can be displayed by the UNIX command *ls*, the size of INGRES Morton code spatial index tables can be calculated using the formula given in Chapter Four. Table 6.12 gives the number of kilo-bytes for an index file or a table of different number of profiles contained in the spatial indices.

Table 6.12 Number of kilobytes required by the three approaches.

No. of Profiles	M=4	M=5	M=6	M=7	MC	INGRES
515	24	20	22	20	25	96
743	33	28	33	29	35	138
867	38	32	38	33	41	161
907	39	33	38	35	43	132
1002	43	37	42	37	48	187
1097	47	40	47	42	52	204
1195	52	44	50	44	57	222
1262	58	47	54	47	60	235
1370	59	50	58	51	65	255
1452	63	53	62	54	69	270
1710	74	63	73	63	81	318
Average KB/Profile	0.043	0.037	0.042	0.037	0.048	0.187

Figure 6.8 shows the disk space usage occupied by the INGRES tables, Morton code index files (MC) and R-tree index files. With the same profile number, an INGRES table takes four to five times the size of a Morton code or R-tree file. The space requirement of the three approaches increases linearly with the growth of the number of profiles being indexed, and the INGRES tables have the fastest growth rate.

The space used by Morton code index files is slightly more than that used by R-tree index files. The difference, however, is not significant. An interesting observation is that the space used by R-trees of even branch factors M is larger than that used by the trees of odd M, while the sizes of the trees are almost identical within each group. Figure 6.9 gives a better view of the space used by Morton code and R-tree index files.

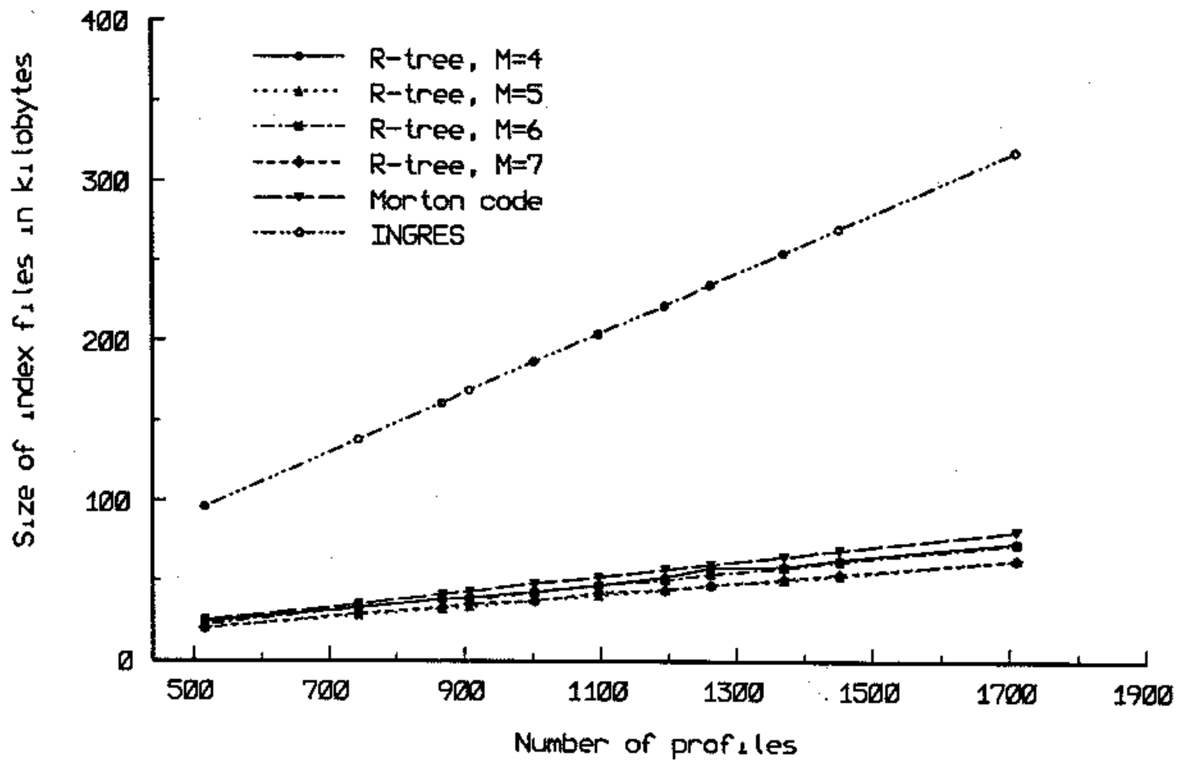


Figure 6.8 Space requirement for Morton code, R-tree and INGRES MC indices.

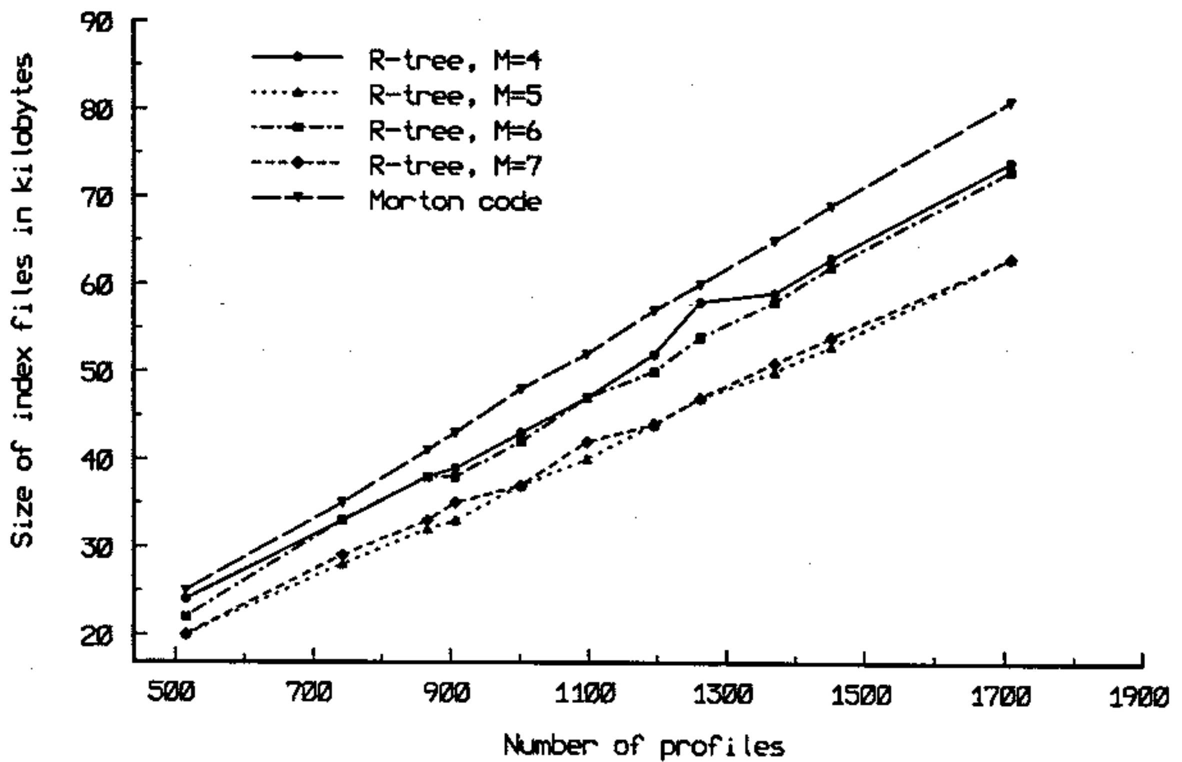


Figure 6.9 Space requirement for Morton code and R-tree indices.

6.7 Advantages and Disadvantages of the Three Indexing Approaches

From the experiment, the following conclusions are drawn,

1. Since Morton codes are easy to compute, it takes the smallest amount of time to build index files among the three spatial indexing approaches. Building an R-tree requires comparing of minimal bounding rectangles, splitting nodes, and re-inserting entries, so that it is two to three times slower than building Morton code index files.

2. From the range search point of view, R-tree indices are better than Morton code indices. The search time on the trees is almost not affected by the size and location of the query windows. On the contrary, the search time on the Morton code indices increases sharply when the query range becomes bigger. With the query window at a different location, the search algorithm appears to have a slightly different performance.

3. The Morton sequence is an indexing mechanism for static data. Since a Morton sequence is not organized through pointers, taking out one Morton code from the sequence will cause massive data movement. Linked by pointers, an R-tree can be easily reorganized when adding or removing some of its entries. The pointer's values are changed so that nodes are linked together in a new order. Deleted nodes can also be collected through the pointers, and the nodes can be reused when a new one is required. No data movement is needed.

4. For packed R-trees like these built here for HDCS, deletion time can be largely reduced since a deletion usually takes place with a range of profiles being removed and those profiles are kept in the sibling nodes of an ancestor in the trees.

5. Due to the overhead of managing an INGRES database, it is very costly to build and search Morton code indices under the INGRES environment. Building the index under INGRES required 32 times more CPU time than directly building Morton code index, and searching required an average of 176 times more CPU time compared to the Morton code index.

CHAPTER SEVEN

SUMMARY AND CONCLUSION

Spatial indexing is an important and active research topic in spatial data structures. Among the ever-growing numbers of spatial data structures, Morton sequences are widely used in geographic data related applications. The fast query speed on R-trees is attributed to their hierarchical structures. R-trees are also promising in indexing dynamically changing spatial data.

Morton sequences were used to index profiles of swath data, both directly and in a relational database management system (RDBMS) environment under INGRES. A timing comparison between the direct and RDBMS approaches was made.

R-trees were also used to index profiles of swath data, and a hierarchical approach was invented using R-trees to index groups of profiles, with each group having a sub-index such as another R-tree to index the data within the group.

Comparisons were made among Morton code and R-tree spatial indexing mechanisms, and Morton code implementations in both the C language and under the INGRES RDBMS with the C as its host language. It is the first time such a comparison was conducted.

The data structures designed for Morton sequences, R-trees and INGRES Morton sequences to index profiles in HDCS were all shown to be viable. The search results

include the sequential numbers of the profiles which are within or overlapping a query window, as well as the path from the project to these profiles. The path and the profile sequential numbers can be used to access the data associated with the profiles.

To decide the best structure for spatial indexing, three factors should be taken into account: the time used for building and searching indices, and the space required by the index files. Table 7.1 lists the construction time, the search time based on QW8 which covers about 50% of the data, and space used for index files of a line with 1710 profiles. The minimum and maximum values of time and space requirements are used for R-trees of different branch factors.

Table 7.1 Comparison of times (in seconds) for building and searching index files and the index file's space requirements (in KB).

	Average build time	Search time for QW8	Space required
Morton Code	3.83	0.41	81
R-tree (min.)	8.86 (M=7)	0.07 (M=5, 6, 7)	63 (M=5, 7)
R-tree (max.)	11.16 (M=4)	0.09 (M=4)	74 (M=4)
INGRES MC	121.32	1853.72	318

Based on these experimental results, the main findings of this research are

1. Morton code indices under the INGRES RDBMS cannot satisfy the needs of HDCS applications. Building and keeping INGRES Morton code indices takes too much computer resources. It is also extremely slow to search the INGRES tables.

2. Building Morton sequence indices is about three times faster than building R-tree indices.

3. R-trees take a slightly less amount of disk space than Morton sequences do, and, more importantly, the search time using R-trees is from five to nine times faster.

4. Improved efficiency of the modified deletion algorithm is definitely another asset when profile positions have to be modified due to e.g. navigation corrections.

5. R-trees with different branch factors have different characteristics. R-trees of branch factor 7 yield the best performance relative to M=4, 5, and 6 in building and searching, and also have the least file storage.

It can be concluded that the R-tree structure is the best among the three for the application of spatial indexing of bathymetric profiles.

The R-tree index construction algorithm makes substantial use of disk storage. One suggestion for future work would be to improve the speed by moving all parts of the algorithm to be in memory. The speed of the searching algorithm could also be improved in this way.

This research has given rise to the following open questions:

1. Why is the RDBMS so much slower?

2. How slow is the Morton code index on deletion?

3. How well can Morton code indexing based on a list structure (e.g. skip lists) support deletion?

4. How well does R-tree indexing generalize to more general types of spatial data, e.g., polygons, polylines, and volumetric data?

5. Are other RDBMS systems equally slow?

REFERENCES

Abel, D. J. (1989). "SIRO-DBMS: a database tool-kit for geographical information systems." *International Journal of Geographical Information Systems*, Vol. 3, No. 2, pp. 103-116.

Abel, D. J., and Mark, D M. (1990). "A comparative analysis of some two-dimensional orderings." *International Journal of Geographical Information Systems*, Vol. 4, pp. 21-31.

Ahuja, N. (1983). "On approaches to polygonal decomposition for hierarchical image representation." *Computer Vision, Graphics, and Image Processing*, Vol. 24, pp. 200-214.

Bell, S. B., Diaz, B. M., Holroyd, F., and Jackson, M., J. (1983). "Spatially referenced methods of processing raster and vector data." *Image and Vision Computing*, Vol. 1, No. 4, pp. 211-220.

Bentley, J. L. (1975). "Multidimensional binary search trees used for associative searching." *Communications of the ACM*, Vol. 18, pp. 509-517.

Date, C. J. (1981). *An Introduction to Database Systems*. Reading, Massachusetts. Addison-Wesley.

Franklin, W. M. R. (1984). "Adaptive grids for geometric operations." *Cartographica*, Vol. 21, pp. 160-168.

Faloutsos, C., Sellis, T., and Roussopoulos, N. (1987). Analysis of object oriented spatial access methods. *Proceedings of the SIGMOD Conference*, San Francisco, California, May. pp. 426-439.

Gibson, L. and Lucas, D. (1982). "Vectorization of raster images using hierarchical methods." *Computer Graphics and Image Processing*, Vol. 20, No. 1, pp. 82-89.

Goodchild, M. F. (1989). "Tiling large geographical database". *Proceedings of Design and Implementations of Large Spatial Database*, First Symposium SSD'89, Santa Barbara, California, July 17-18, pp. 138-146.

Greene, D. (1989). "An implementation and performance analysis of spatial data access methods." *Proceedings of the Fifth IEEE International Conference on Data Engineering*, Los Angeles, California, February 6-10. IEEE Computer Society Press, pp. 606-615.

Guenther, O., and Buchmann, A. (1990). "Research issues in spatial database." *SIGMOD Record*, Vol. 19, pp. 61-68.

Guttman, A. (1984). R-trees: "A dynamic index structure for spatial searching." *SIGMOD Record*, VOL. 14, pp. 47-57.

INGRES Manual (1990). "Dynamic Programming for C", Chapter 4, 12 pages.

INGRES Manual (1992). "Space Calculation", Chapter 16, 20 pages.

Morton, G. M (1966). "A computer oriented geodetic data base and a new technique in file sequencing." IBM Ltd., Ottawa, Canada.

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

Nievergelt, J., and Hinrichs, K. H. (1993). *Algorithms and Data Structures*. Prentice Hall, Englewood Cliffs, New Jersey.

Noronha, V. T. (1988). "A survey of hierarchical partitioning methods for vector images." *Proceedings of the Third International Symposium on Spatial Data Handling*, Sydney, Australia, August, pp. 185-200.

Ocean Mapping Group (1990). "Requirements analysis and conceptual design of data cleaning tools for large bathymetric data sets." University of Mew Brunswick.

Ocean Mapping Group (1991). "B-5 Data structure design working document." University of Mew Brunswick, March 6.

Peuquet, D. J. (1984). "A conceptual framework and comparison of spatial data models." *Cartographica*, Vol. 21, pp. 66-113.

Preparata, F. P., and Shamos, M. I. (1985). *Computational Geometry: An Introduction*. Springer-Verlag, New York.

Roussopoulos, N., and Leifker, D. (1985). "Direct spatial search on pictorial databases using packed R-trees." *Proceedings of the SIGMOD Conference*, Austin, Texas, May. pp. 17-31.

Samet, H., and Webber, R. E. (1988). "Hierarchical data structures and algorithms for computer graphics." *IEEE Computer Graphics & Applications*, May. pp. 48-68.

Samet, H. (1990a). *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, Massachusetts.

Samet, H. (1990b). *Applications of Spatial Data Structures*. Addison-Wesley, Reading, Massachusetts.

Tomlinson, R. F. (1976). "Computer handling of geographical data." Paris, The UNESCO Press.

Ware, C., Wells, D., Nickerson, B. G., Lee, Y. C., Derenyi, E., and Vanicek, P. (1990). "Strategic research project progress report." STR0040636 Progress Report, Ocean Mapping Project, University of New Brunswick, 7 pages.

Ware, C., Slipp, L., Wong, K. W., Nickerson, B. G., Wells, D., Lee, Y. C., Dodd, D. and Costello, G. (1992). "A system for cleaning high volume bathymetry." International Hydrographic Review, Accepted June 1992.

Waugh, T. C. and Healey R. G. (1987). "The GEOVIEW design: A relational data base approach to geographical data handling." *International Journal of Geographical Information Systems*, Vol. 1, No. 2, pp. 101-118.

Witten, I. H. and Wyvill, B. (1983). "On the generation and use of space-filling curves." *Software, Practice and Experience*, Vol. 13, No 6.

Wood, D. (1993). *Data Structures, Algorithms, and Performance*. Addison-Wesley Publishing Company, Reading, Massachusetts.

Yang, W. P. (1992). "A new range search algorithm for large point databases." M.Sc.E. thesis, University of New Brunswick.

APPENDIX A

STRUCTURES ACCOMMODATING SEARCH RESULTS

```
typedef struct ProjectStruct {
        char            *name;
        struct {
                char    *title;
                int     number;
                struct {
                        char    *name;
                        double  semiMajorAxis, semiMinorAxis;
                        double  deltaX, deltaY, deltaZ;
                } ellipsoid;
                int     utmZone, timeZone;
                char    *elevationDatumName;
        } config;
        int             mapProjection;
        int             numVessels;
        Vessel          *firstVessel;
        Vessel          *lastVessel;
        int             numSpatialSubsets;
        SpatialSubset   *firstSpatialSubset;
        SpatialSubset   *lastSpatialSubset;
} Project;


typedefstruct VesselStruct {
        char                            *name;
        HDCS_VesselConfig               vesselConfig;
        struct ProjectStruct            *project;       /* ptr to parent */
        struct SpatialSubsetStruct      *spatialSubset;/* ptr to parent */
        int                             numDays;
        Day                             *firstDay;
        Day                             *lastDay;
        struct VesselStruct             *next;
        struct VesselStruct             *previous;
} Vessel;
```

```
typedef struct DayStruct  {
        char                        *name;
        struct VesselStruct         *vessel;        /* ptr to parent */
        int                          numSessions;
        Session                     *firstSession;
        Session                     *lastSession;
        struct DayStruct            *next;
        struct DayStruct            *previous;
} Day;


typedef struct SessionStruct  {
        char                  *name;
        struct DayStruct      *day;                 /* ptr to parent */
        unsigned char          navStatusChanged;
        int                    numNavLines;
        NavLine               *firstNavLine;
        NavLine               *lastNavLine;
        double                 minTimePD,  maxTimePD;
        double                 minLatPD,   maxLatPD;
        double                 minLongPD,  maxLongPD;
        double                 minDepthPD, maxDepthPD;
        int                    numProfiles;
        PDProfile             *firstProfile;
        PDProfile             *lastProfile;
        unsigned char          statusChanged;
        struct SessionStruct  *next;
        struct SessionStruct  *previous;
} Session;
```

```
typedef struct NavLineStruct {
        int                     lineNumber;             /* 1..numLines */
        char                    *sourceFile;
        int                     bgnIndex;
        int                     endIndex;
        double                  minTime, maxTime;       /* in seconds */
        double                  minLat, maxLat;         /* in radians */
        double                  minLong, maxLong;       /* in radians */
        struct SessionStruct    *session;
        unsigned char           statusChanged;
        int                     numNavPositions;
        NavPosition             *firstNavPosition;
        NavPosition             *lastNavPosition;
        struct NavLineStruct    *next;
        struct NavLineStruct    *previous;
} NavLine;


typedef struct NavPosnStruct {
        int                     positionNumber;
        double                  time;
        double                  latitude;               /* in radians */
        double                  longitude;              /* in radians */
        double                  positionalAccuracy;     /* in meters */
        unsigned char           statusChanged;
        unsigned long           status;
        double                  x;                      /* in meters */
        double                  y;                      /* in meters */
        unsigned long           color;
        struct NavLineStruct    *navLine;               /* ptr to parent */
        struct NavPosnStruct    *next;
        struct NavPosnStruct    *previous;
} NavPosition;
```

```
typedef struct PDProfileStruct {
        int                         index;
        double                      time;
        double                      latitude;
        double                      longitude;
        double                      x;
        double                      y;
        unsigned char               statusChanged;
        struct SessionStruct        *session;
        int                         numDepths;
        PDepth                      *firstDepth;
        PDepth                      *lastDepth;
        struct PDProfileStruct      *next;
        struct PDProfileStruct      *previous;
} PDProfile;
```

```
typedef struct PDepthStruct {
        int                         index;
        double                      time;
        double                      latitude;
        double                      longitude;
        double                      depth;
        double                      depthAccuracy;
        unsigned char               statusChanged;
        unsigned long               status;
        unsigned long               color;
        double                      x;
        double                      y;
        int                         stDevLevel;
        struct PDProfileStruct      *profile;
        struct PDepthStruct         *next;
        struct PDepthStruct         *previous;
} PDepth;
```

```
typedef struct SpatialSubsetStruct {
    int                         id;
    int                         cleaned;
    double                      centerLatitude;     /* in radians */
    double                      centerLongitude;    /* in radians */
    double                      widthHeight;        /* in meters */
    double                      tiltAngle;          /* in radians */
    double                      centerX;            /* in meters */
    double                      centerY;            /* in meters */
    double                      cosTiltAngle;
    double                      sinTiltAngle;
    double                      cosNegTiltAngle;
    double                      sinNegTiltAngle;
    double                      latitude[4];        /* in radians */
    double                      longitude[4];       /* in radians */
    double                      x[4];               /* in meters */
    double                      y[4];               /* in meters */
    double                      minX, maxX;         /* in meters */
    double                      minY, maxY;         /* in meters */
    double                      minLat, maxLat, deltaLat;
    double                      minLong, maxLong, deltaLong;
    int                         minLatMS, maxLatMS, incrLatMS;
    int                         minLongMS, maxLongMS, incrLongMS;
    struct ProjectStruct        *project;     /* ptr to parent */
    int                         numVessels;
    Vessel                      *firstVessel;
    Vessel                      *lastVessel;
    struct SpatialSubsetStruct  *next;
    struct SpatialSubsetStruct  *previous;
} SpatialSubset;
```

APPENDIX B

ALGORITHMS FOR SEARCHING MORTON SEQUENCE INDICES IN HDCS

Algorithm 1. **SearchProfileMS**

Functionality: Search a profile spatial index file and report all the profiles which have at least one of their four MBR corner points within the query window.

Input: A *QueryWindow* structure, a pointer to a file containing one line's profile spatial index.

Output: A list of profiles from this line within and intersecting the QW.

1. Initialize a counter array of size equal to the number of profiles in this file;
2. Calculate the Morton codes, MC_{LL}, MC_{UR}, MC_{UL} and MC_{LR}, for the query window's four corner points;
3. Define the first MC, MC_{start} in the Morton sequence of this file which is not smaller than MC_{LL};
4. Define the first MC in the Morton sequence of this file which is not smaller than MC_{UR};
5. $MC_{TP} \leftarrow MC_{start}$;
6. While $MC_{TP} < MC_{UR}$
    7. $MC_{LL} \leftarrow MC_{TP}$;
    8. Decode MC_{TP} to get TP's latitude and longitude
    9. If the test point is in the QW
        10. Counter increases one for profile with MC_{TP};
        11. $MC_{TP} \leftarrow$ next MC in the file;
    12. Else
        13. Call FindEnteringPoint to calculate MC_{EP};
        14. $MC_{TP} \leftarrow$ the MC just larger than MC_{EP};
    15. EndIf
16. EndWhile
17. Sequentially scan the counter array, and create a list for all the profiles having a non-zero counter.

Algorithm 2. **FindEnteringPoint**

Functionality: Find the point on the edge of, or in the query window, where the Morton sequence is entering.
Input: A Morton code of an outside-window point, a query window, and the Morton codes of the window's southeast and north west corners: MC_{LR} and MC_{UL}.
Output: The Morton code of the entering point.

1. Call PointQWRelation to find the relation between the point and the query window;
2. For Area One:
    3. Call GetQWEdgeMC to calculate the MC on the bottom edge;
4. For Area Three:
    5. IF $MC_{Outside}$ > MC_{LR}
        6. If $MC_{Outside}$ > MC_{UL} /* The Morton sequence crosses the right edge. */
            7. Call GetQWEdgeMC to calculate the MC on the right edge: MC_{RE};
            8. Call FallIntoQW to calculate the MC in the QW;
        9. Else
            10. Call GetQWEdgeMC to calculate the MC_{LE} and MC_{RE};
            11. If MC_{LE} > MC_{RE} /* The Morton sequence touches the right edge */
                12. Call FallIntoQW to calculate the MC in the QW;
            13. Else /* The Morton sequence touches the left edge */
                14. MC_{LE} is the MC in the QW;
            15. EndIf
        16. EndIf
    17. Else /* The Morton sequence touches the bottom edge */
        18. Call GetQWEdgeMC to calculate the MC on the bottom edge;
    19. EndIf
20. For Area Four:
    21. Call GetQWEdgeMC to calculate the MC_{LE};
22. For Area Six:
    23. If $MC_{Outside}$ > MC_{UL} /* The Morton sequence crosses the right edge */
        24. Call GetQWEdgeMC to calculate MC on the right edge;
        25. Call FallIntoQW to calculate MC in the QW;
    26. Else
        27. Call GetQWEdgeMC to calculate MC_{LE} and MC_{RE};
        28. If MC_{Left} > MC_{Right} /* The Morton sequence touches the right edge */
            29. Call FallIntoQW to calculate the MC in the QW;
        30. Else /* The Morton sequence touches the left edge */
            31. MC_{Left} is the MC in the QW;
        32. EndIf
    33. EndIf
24. For Area Seven:
    35. If $MC_{Outside}$ > MC_{UL}
        36. If $MC_{Outside}$ > MC_{LR} /* The Morton sequence crosses the top edge */
            37. Call GetQWEdgeMC to calculate MC on the top edge;

38. Call FallIntoQW to calculate MC in the QW;
39. Else
40. Call GetQWEdgeMC to calculate the MC_{BE} and MC_{TE};
41. If MC_{Bottom} > MC_{Top} /* The Morton sequence touches the top edge */
42. Call FallIntoQW to calculate the MC in the QW;
43. Else /* The Morton sequence touches the bottom edge */
44. MC_{Bottom} is the MC in the QW;
45. EndIf
46. EndIf
47. Else /* The Morton sequence touches the left edge */
48. Call GetQWEdgeMC to calculate the MC on the left edge;
49. EndIf
50. For Area Eight:
51. If $MC_{Outside}$, > MC_{LR} /* The Morton sequence crosses the top edge */
52. Call GetQWEdgeMC to calculate the MC on the top edge;
53. Call FallIntoQW to calculate the MC in the QW;
54. Else
55. Call GetQWEdgeMC to calculate the MC_{BE} and MC_{TE};
56. If MC_{Bottom} > MC_{Top} /* The Morton sequence touches the top edge */
57. Call FallIntoQW to calculate the MC in the QW;
58. Else /* The Morton sequence touches the bottom edge */
59. MC_{Bottom} is the MC in the QW;
60. EndIf
61. EndIf

Algorithm 3. **GetQWEdgeMC**

Functionality: Get an on-edge Morton code which is the smallest among all the on-edge Morton code larger than the Morton code outside the QW.

Input: The Morton code of an outside-window point, two corner points describing one edge of the QW.

Output: MC_{Edge}.

1. Set the first and last point to the corners of the query window;
2. While (first <= last)
3. middle = (first + last) / 2;
4. Calculate the Morton code of middle MC_{middle};
5. If $MC_{outside}$, < MC_{middle}
6. $MC_{Edge} \leftarrow$ MC;
7. last $\leftarrow$ mid - 1;
8. Else
9. first $\leftarrow$ mid + 1;
10. EndWhile

Algorithm 4. **FallIntoQW**

Functionality: Calculate a new Morton code from two Morton codes, one of which is outside the QW and the other is on the edge of the window. The new Morton code is larger than the outside one; and the point it represents is within the QW.
Input: Two Morton codes, one outside and the other on the edge of the QW.
Output: A Morton code inside the QW.

1. Compare two input Morton codes, digit by digit;
2. If two Morton codes are different at *i*th digit
    3. Change the *i*th and *(i-1)*th digit of the on-edge Morton code to two and zero respectively, so that it becomes an in-window Morton code.

Algorithm 5. **Encoding**

Functionality: Compute a 64-bit Morton code by interleaving a pair of 32-bit unsigned integer coordinates.
Input: X and Y coordinates.
Output: A Morton code.

1. For i = 1 to 0, loop
    2. mc[i] ← 0;
    3. For j = 0 to 16, loop
        4. bitPair = ((x & 1) << 1) | (y & 1);
        5. Set the mc[i] ← mc[i] | bitPair << (2 * j);
        6. x = x >> 1;
        7. y = y >> 1;
    8. EndLoop
9. EndLoop

Algorithm 6. **Decoding**

Functionality: Decode a 64-bit Morton code into a pair of 32-bit unsigned integer coordinates.
Input: A Morton code.
Output: X and Y coordinates.

```
1. bitShift ← 0;
2. Initialize the coordinates: x ← 0, y ← 0;
3. mcTmp ← mc;
4. for i = 1 to 0, loop
    5. for j = 0 to 16, loop
        6.  yBit ← mcTmp[i] & 1;
        7.  xBit ← ((mcTmp[i] >> 1)  &  1);
        8.  x ← x | (xBit << bitShift);
        9.  y ← y | (yBit << bitShift);
        10. bitShift ← bitShift +1;
        11. mcTmp[i] ← mcTmp[i] >> 2;
    12. EndLoop
13. EndLoop
```

APPENDIX C

AN EXAMPLE OF USING DYNAMIC SQL TO RETRIEVE MORTON CODES FROM AN INGRES TABLE

Listed below are two functions to illustrate the use of dynamic SQL to retrieve a Morton code which is larger than a given Morton code *mc*. The name of the table to search the Morton code from is also given, and the search result is stored in structure *profileSpIndxData* which has items for a Morton code and a profile number.

```
#include <stdio.h>
#include <malloc.h>

/* Declare the SQLCA structure and the SQLDA typedef */
EXEC SQL INCLUDE SQLCA;
EXEC SQL INCLUDE SQLDA;

EXEC SQL DECLARE stmt STATEMENT;          /* Dynamic SQL statement */
EXEC SQL DECLARE csr CURSOR FOR stmt;

# define NUM_INDEX_ELEMS 15
# define SQL_NOTFOUND 100
# define LEN_SQL_CMD 100                   /* Max SQL statement length */

#include "MortonSequenceSpatialIndex.h"
#include "IngresMortonCode.h"

int32
  FindALargerMC( mc, profileSpIndxData, tableName )

  MortonCode                  mc;                   /* A Morton code */
  ProfileSpatialIndexFileData *profileSpIndxData;   /* A structure */
  char                        *tableName;           /* A string */

{
  int       i, j;
  char      mc0Str[15], mc1Str[15];
  double    doubleMC0, doubleMC1;
```

```
int         base_type;

IISQLDA    *sqlda = (IISQLDA *)0;      /*Pointer to the SQL dynamic area */
IISQLVAR   *sqv;

EXEC SQL BEGIN DECLARE SECTION;
  char sqlSelectPro[LEN_SQL_CMD +1];
EXEC SQL END   DECLARE SECTION;

if ( profileSpIndxData == NULL )
{
  return (-1);
}

/* Convert the Morton code into strings */
UnsignInt32ToStr( mc[0], mc0Str );
UnsignInt32ToStr( mc[1], mc1Str );

/* Form a search command */
sprintf( sqlSelectPro, "%s %s", "SELECT * FROM", tableName );
strcat( sqlSelectPro, " WHERE mc0 > " );
strcat( sqlSelectPro, mc0Str );
strcat( sqlSelectPro, " OR  ( mc0 = " );
strcat( sqlSelectPro, mc0Str );
strcat( sqlSelectPro, " AND  mc1 > " );
strcat( sqlSelectPro, mc1Str );
strcat( sqlSelectPro, " )" );

/* Allocate a new SQLDA */
Init_Sqlda( &sqlda, NUM_INDEX_ELEMS );

EXEC SQL PREPARE stmt from :sqlSelectPro;
EXEC SQL DESCRIBE stmt INTO :sqlda;

/* Allocate memory for the variables corresponding to the items in the table */
AllocateProTableVariables( &sqlda );

/* Open the dynamic cursor */
EXEC SQL OPEN csr;
if (sqlca.sqlcode != 0) {
     return( -1 );
}

EXEC SQL FETCH csr USING DESCRIPTOR :sqlda;
if (sqlca.sqlcode == 0)   {
```

```
      for (i = 0;  i < sqlda->sqld;  i++)        {
        sqv = &sqlda ->sqlvar[i];

        /* Find the base->type of the result */
        if ( (base_type = sqv->sqltype)  <  0 )          {
              base_type = - base_type;
        }

        switch ( base_type )              {
          case  IISQ_INT_TYPE:
              profileSpIndxData->ProfileNumber = *(long *)sqv->sqldata;
              break;

          case  IISQ_FLT_TYPE:
              if ( i == 0 )     {
                    profileSpIndxData->MC[0] =
                                    (unsigned long) (*(double *)sqv->sqldata);
              }
              else      {
                    profileSpIndxData->MC[1] =
                                    (unsigned long) (*(double *)sqv->sqldata);
              }
              break;
          }
        }
      }

      rc = 0;
}
else if ( sqlca.sqlcode == 100 )
{
      rc = 100;
}

EXEC SQL CLOSE csr;

return( rc );
```

```
AllocateProTableVariables( sqlda )

  IISQLDA          **sqlda;

{
  int    i;
  IISQLVAR *sqv;
  int    base_type;
  int    res_cur_size;

  struct
  {
    int res_length;     /* Size of mem_data */
    char *res_data;     /*Pointer to allocated result buffer */
  }    res_buf;

  res_buf.res_length = 0;
  res_buf.res_data  = NULL;

  for (res_cur_size = 0, i=0;  i < (*sqlda)->sqld;  i++)
  {
    sqv = &(*sqlda)->sqlvar[i];

    if ((base_type = sqv->sqltype) < 0)
         base_type = -base_type;

    switch(base_type)
    {
       case IISQ_INT_TYPE:
         res_cur_size +=sizeof(long);
         sqv->sqllen = sizeof(long);
          break;

       case IISQ_FLT_TYPE:
         res_cur_size += sizeof(double);
         sqv->sqllen = sizeof(double);
          break;
    }

  }

  if (res_buf.res_length>0 && res_buf.res_length < res_cur_size)
  {
      free(res_buf.res_data);
      res_buf.res_length =0;
```

```
}

if (res_buf.res_length == 0)
{
     res_buf.res_data = Alloc_Mem(res_cur_size,
              "result data storage area");
     res_buf.res_length = res_cur_size;
}

for (res_cur_size=0, i=0;   i < (*sqlda)->sqld; i++)
{
  sqv =&(*sqlda)->sqlvar[i];

  if ((base_type = sqv->sqltype) < 0)
      base_type = -base_type;

  sqv->sqldata = (char *)&res_buf.res_data[res_cur_size];

  res_cur_size += sqv->sqllen;

  if (base_type == IISQ_CHA_TYPE)
  {
      res_cur_size++;
      if (res_cur_size %2)
              res_cur_size ++;
  }

  if (sqv->sqltype < 0)
  {
    sqv->sqlind = (short *)&res_buf.res_data[res_cur_size];
  }
  else
  {
    sqv->sqlind = (short *)0;
  }
}
}
```

APPENDIX D

THE MODIFIED R-TREE DELETION ALGORITHMS

Algorithm 1: **ModifiedRTreeDeletion**

Functionality: This function deletes profile within the range [p_1, p_2] from a R-tree spatial index s.
Input: A pointer to the spatial index s, and a range of profiles to be deleted [p_1, p_2].
Output: A modified R-tree index file.

1. Initialize *AvoidList* to NULL;
2. For each profile i in the profile range [p_1, p_2], loop
   3. If i is on *AvoidList*, delete i from the list and repeat Step 2 for $i+1$;
   4. Find the leaf-node f containing i;
   5. Delete entry i from f;
   6. For each profile j in f, loop
      7. If the j is within the range [p_1, p_2]
         8. Add j into *AvoidList*;
         9. Delete profile j from node f;
      10. EndIf
   11. EndFor
   12. If all the profiles in the leaf-node have been deleted
      13. Add the record to the deleted record linked list in the R-tree file;
      14. Call **DeleteUpwards** to delete f from f's parent and the profiles in the siblings of f;
   15. EndIf
16. EndFor
17. Call **CondenseTree** to propagate the deletion up to the root;
18. Check the root and remove the root if it has only one child;

Algorithm 2. **DeleteUpwards**

Functionality: This function starts from a node f and goes up to f's parent P to delete P's other children.

Input: A pointer to the spatial index s, a pointer to node f, and a pointer to a node g where a deletion stops.

Output: A modified R-tree index file.

1. If the node f is the root, return.
2. $P \leftarrow f \uparrow$ parent.
3. Search and delete f from P;
4. Call **DeleteSubtree**(P) to delete the profiles in P's sub-trees which are on the right of f;
5. If there is no entry left in P
   6. Call **DeleteUpwards**(s, f, g) recursively to delete the parent of P;
7. EndIf

Algorithm 3. **DeleteSubtree**

Functionality: This function deletes the profiles in the sub-trees of a node *P*.
Input: A pointer to the spatial index *s*, a pointer to node *P*, a pointer to a node *g* where a deletion stops, and a pointer to *AvoidList*.
Output: A modified R-tree index file.

1. For each child *C* in the node *P*, loop
    2. If *C* is a leaf-node
        3. If the profile number *i* in an entry *e* of *C* is within the range $[p_1, p_2]$
            4. Add *i* into *AvoidList*;
            5. Delete *e* from *C*;
        6. EndIf
        7. If any entry is deleted from the node and the node is not empty
            8. Return;
        9. Else if a entry is deleted from the node and the node is empty
            10. Add the node into deleted record list in the file;
            11. Delete *C* from *N*;
        12. Else if no entry is deleted
            13. Return;
        14. EndIf
    15. Else /* when *C* is not a leaf-node */
        16. Call **DeleteSubtree** recursively to delete the subtree of *N*;
        17. If there is no entry in *C* of the subtree
            18. Delete *C* from *N;*
        19. EndIf
    20. EndIf
21. EndFor
22. If node *N* is empty
    23. Add the node into deleted record linked list in the file
24. EndIf

APPENDIX E

A DETAILED LIST OF HDCS DATA USED FOR THE EXPERIMENT

ConceptionBay
Matthew number of days = 4

Day / Line	Number of lines / profiles	Size of data
1991311	number of lines = 10	
13:46:06	number of profiles = 1145	size of data = 2765 KB
14:03:47	number of profiles = 1202	size of data = 2903 KB
14:21:30	number of profiles = 1171	size of data = 2823 KB
14:54:31	number of profiles = 1277	size of data = 3077 KB
15:13:55	number of profiles = 1097	size of data = 2648 KB
16:12:31	number of profiles = 1170	size of data = 2823 KB
16:40:50	number of profiles = 1071	size of data = 2592 KB
16:58:51	number of profiles = 1082	size of data = 2618 KB
17:17:11	number of profiles = 1200	size of data = 2896 KB
17:35:29	number of profiles = 1197	size of data = 2898 KB
1991312	number of lines = 15	
13:36:42	number of profiles = 1384	size of data = 3331 KB
13:57:02	number of profiles = 1370	size of data = 3297 KB
14:24:03	number of profiles = 1195	size of data = 2884 KB
14:42:16	number of profiles = 1262	size of data = 3041 KB
15:13:39	number of profiles = 1421	size of data = 3418 KB
15:32:13	number of profiles = 1452	size of data = 3498 KB
15:58:39	number of profiles = 1472	size of data = 3538 KB
16:17:00	number of profiles = 1261	size of data = 3036 KB
16:38:40	number of profiles = 1241	size of data = 2992 KB
16:57:29	number of profiles = 1278	size of data = 3085 KB
17:12:19	number of profiles = 1294	size of data = 3117 KB
17:31:44	number of profiles = 1269	size of data = 3052 KB
17:52:43	number of profiles = 907	size of data = 2198 KB
18:06:58	number of profiles = 789	size of data = 1924 KB
18:37:41	number of profiles = 743	size of data = 1804 KB
1991313	number of lines = 21	
13:13:58	number of profiles = 1449	size of data = 3480 KB
13:44:12	number of profiles = 1710	size of data = 4105 KB
14:12:06	number of profiles = 907	size of data = 2198 KB
14:48:16	number of profiles = 867	size of data = 2101 KB
15:02:26	number of profiles = 963	size of data = 2326 KB

15:17:36	number of profiles = 888	size of data = 2151 KB
15:31:28	number of profiles = 875	size of data = 2117 KB
15:44:55	number of profiles = 1003	size of data = 2419 KB
15:59:49	number of profiles = 961	size of data = 2326 KB
16:14:06	number of profiles = 1036	size of data = 2510 KB
16:27:06	number of profiles = 1027	size of data = 2477 KB
16:40:40	number of profiles = 1056	size of data = 2552 KB
16:55:41	number of profiles = 1002	size of data = 2419 KB
17:09:40	number of profiles = 1098	size of data = 2653 KB
17:25:30	number of profiles = 995	size of data = 2402 KB
17:40:36	number of profiles = 1031	size of data = 2493 KB
17:53:38	number of profiles = 515	size of data = 1272 KB
18:01:52	number of profiles = 506	size of data = 1236 KB
18:10:25	number of profiles = 1032	size of data = 2493 KB
18:23:26	number of profiles = 1050	size of data = 2536 KB
18:36:03	number of profiles = 1071	size of data = 2585 KB
1991314	number of lines = 3	
16:44:37	number of profiles = 1076	size of data = 2594 KB
16:57:19	number of profiles = 1046	size of data = 2529 KB
17:10:17	number of profiles = 1078	size of data = 2602 KB

VITA

Candidate's full name:	Feng Gao
Place and date of birth:	Shanghai, China May 7, 1963
Education:	
9/1980 - 7/1984	Beijing College of Architecture and Civil Engineering, Beijing, China. B.Sc.E (Surveying Engineering)
9/1984 - 7/1987	Tsinghua University, Beijing, China. M.Sc.E (Surveying Engineering)
9/1990 - 12/1993	University of New Brunswick, Fredericton, N.B., Canada

Training:

2/1990	UNESCO sponsored invitational training course on *Remote Sensing and Active Faults for Land Use Management.* Manila, Philippines

Publications:

Gao, F. and Liu, H. (1986). The study and development of a cartographic database management system for large scale maps. *Proceedings of the First National Symposium on Cartography for Large Scale Maps*, Kunming, China, November.

Gao, F. (1987). "Computer aided cartography systems." *Surveying and Mapping*, No. 2, 1987.

Gao, F. and Liu, H. (1987). "The study and development of a cartographic database management system for large scale maps." *Beijing Surveying and Mapping*, No. 2, 1987.

Liu, H. and Gao, F. (1987). The design and implementation of a cartographic database management system. *Proceedings of the Second National Symposium on Computer Aided Cartography*, Wuhan, China, May.

Gao, F. and Guo J. (1988). "A new method for field data acquisition in computer aided cartography." *Bulletin of Surveying and Mapping*, No. 4, August.

Guo, J., Ji, R., Na, X., Gao, F., Wang, G., Ding, H., and Rong, G. (1988). Using an infrared spot position measuring system for dynamic testing of structural models. *Proceedings of the First Symposium on Surveying Instruments*, Beijing, China, November.

Guo, J. and Gao, F. (1988). Communication between SHARP PC-1500 and IBM PC family. *Proceedings of the First Symposium on Surveying Instruments*, Beijing, China, November.

Guo, J., Gao, F., Zhang, J. and Tao, Q. (1988). The research process of a automated cartographic system for large scale maps. *Proceedings of the National Symposium on Applications of Computer Aided Cartography for Engineering Surveying*, Wuhan, December.

Gao, F. (1989). "IBM-PC FORTRAN language's screen manipulation and its applications in menu techniques." *Microcomputer & Applications*, No. 1, March.

Gao, F. and Guo, J. (1989). Using software engineering technology in the development of a computer aided cartographic system. *Proceedings of the Symposium for the 30th Anniversary of the Founding of the Chinese Society of Geodesy, Photogrammetry and Cartography*, Xian, April.

Gao, F. and Guo, J. (1989). Data handling and management in computer aided cartography for large scale maps. *Proceedings of the Symposium for the 30th Anniversary of the Founding of the Chinese Society of Geodesy, Photogrammetry and Cartography*, Xian, April.