# SPATIAL JOIN:
# A STUDY OF COMPLEX SPATIAL OPERATION AND ITS UNDERLYING SPATIAL INDEXING METHODS

by

Hong Fan

B.Sc. University of Science and Technology of China, 1989

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
August 1992

ISBN 0-315-83635-0

# APPROVAL

| | |
|---|---|
| **Name:** | Hong Fan |
| **Degree:** | Master of Science |
| **Title of thesis:** | Spatial Join: A Study of Complex Spatial Operation and its Underlying Spatial Indexing Methods |

**Examining Committee:** Dr. Binay Bhattacharya
Professor, Computing Science
Chair

Dr. Woshun Luk
Professor, Computing Science
Senior Supervisor

Dr. Jiawei Han
Associate Professor, Computing Science
Supervisory Committee Member

Dr. Nick Cercone
Professor, Computing Science
External Examiner

**Date Approved:** August 18, 1992

Title of Thesis/Project/Extended Essay

Spatial Join: A Study of Complex Spatial Operation and its Underlying Spatial

Indexing Methods.

Author:

(signature)

Fan Hong

(name)

August 18, 1992

(date)

# Abstract

In recent literature, there has been extensive research on simple spatial operations such as point location and range query, as well as comparative studies on spatial indexing methods (SIM) for simple objects based on simple spatial operations. The thesis tackles the problem of polygon spatial join, which is one of the most complex spatial operations on complex objects which are simple polygons.

Polygon spatial join can be defined as finding all pairs of polygonal objects that overlap each other over their boundaries from two given polygonal data sets. Spatial join is used extensively in geographical information systems, where geographical data is organized by "layer", and a join of the layers creates synthesized information of the same geographical area. It can also be directly extended to realize polygon overlay, which is also a very important complex operation in GIS.

We solve the problem by extensively utilizing popular SIMs: PM Quadtree and R-tree such that complex objects and object relations can be handled efficiently as well. This is based on the observation that spatial join relies on the object spatial occupancy, and these SIMs decompose space from which the spatial data is drawn in a way that spatial properties of spatial objects can be developed and stored. We design algorithms for spatial join based on PM Quadtree and R-tree as well as algorithms with no spatial index involved for comparisons. We also present Grid Coordinate System (GCS) — a SIM for simple spatial objects which is a kind of Grid File based on the object spatial occupancy instead of on the transformed multidimensional point space. Both GCS and R-tree are shown to be empirically superior to PM Quadtree with respect to the spatial join operation.

Comparative studies of the three SIMs under spatial join are also presented. We make use of ObjectStore which is an object-oriented database system as the storage manager for the spatial data. Empirical results are obtained through extensive experiments on the random polygonal nets. We generate the polygonal net for the studies in such a way that it can be adjusted through parameters regarding size, shape and distribution of the composing polygonal data. The polygonal net is represented by a vector data model designed as a multi-file storage-saving structure enhanced with indexing capability.

# Acknowledgments

I would like to express my deepest appreciation and gratitude to Dr. Woshun Luk, my senior supervisor, for his constant guidance, advice, and patience. He has been available and helpful throughout the preparation of this thesis, and his supervision and support is the key motivating force behind this thesis work.

I am very grateful to Dr. Nick Cercone for being my external examiner, and Dr. Jiawei Han for being my supervisory committee member. They were generous with their time, and both fair and helpful with their comments and suggestions. Thanks are also due to Dr. Binay Bhattacharya for his reading and valuable comments on this thesis.

My special thanks to my friend Don Smith and Graham Finlayson who spent much of their precious time reading my thesis and correcting my grammatical mistakes. I would like to thank Kersti Jaager and other department secretaries for giving various kinds of help during my study at Simon Fraser University.

My thanks go also to many of the graduate students in the School of Computing Science at Simon Fraser University, who encouraged me, and provided me with various kinds of help when they were most needed: Xiaobing Chen, Wenfei Fan, Daryl Harms, Tong Lu, Wei Lu, Mark Maurice, Pat T. Pattabbiraman, Kathy Peters, Frank Tong, Yin Lam Wong, Ju Wu, Kuros Yalpani, Xiaogu Zhou, Wei Zhou.

Finally, this work is dedicated to my parents and Benguang Yao. Their immeasurable love and care have always been a source of confidence for me in my life.

# TABLE OF CONTENTS

# Chapter 1
# Introduction

Non-standard computer applications such as robotics, computer vision, computer aided design, and geographical data processing require special operations that are defined on geometric data. These operations are substantially different from the operations defined on non-spatial data, generally in two aspects: geometric data are n-dimensional objects embedded in space, and are accessed by their extent and positions in space as opposed to access by non-spatial attributes; geometric operations possess complex data structures, and careful design of algorithms and data structures are required in order to perform geometric operations efficiently.

There has been extensive research on the point location and range query in the database literature since they are typical database queries. These queries are basically search-based; disk access times affect their performance. A large number of multidimensional point access methods (**PAM**) and spatial indexing methods (**SIM**) have been proposed, both hash-encoded and tree-structured, to manage the retrieval of simple spatial objects and multidimensional point data in order to process the queries efficiently. These include R-trees [GUTT84], K-D-B-trees [ROB81], $R^+$ trees [SELL87], Cell trees [GUN89], Grid File [NHS84], 2D-Isam [NIEV84] (a two-level tree structure similar to grid files) and so on. Since different types of queries pose distinct requirements, it is very important to study the performance of the spatial access methods under various types of spatial queries. Studies in [Gre89] have shown that among the most popular spatial access methods, R-tree is a good choice for the general query processing such as point and range queries. Studies in [KSSS90] further establish the BUDDY hash tree as the choice for both point access methods and spatial access methods for rectangles.

However, little is known about the set operations such as computing set intersection, set inclusion, set difference or any other possible operations that can be performed on sets of geometric objects, especially when set objects are extended spatial objects such as polygons. Even less is known about the performance of spatial indexing methods under set-oriented spatial queries, and the performance of these set operations under various data circumstances. This is largely due to the complex nature of the operations, hence the complicated behavior of spatial indexing methods involved as well as the variety of the representations of the problem itself.

In this thesis, we will study one of the most important set operations: spatial join in spatial information systems, such as geographical information systems and spatial databases. Since this operation is heavily dependent on the spatial indexing method, comparative studies of popular spatial indexing methods will also be presented. In fact, a new indexing method is presented in this thesis, which is shown to be empirically superior to other well-known spatial indexing methods, such as Quadtree and R-tree with respect to the spatial join operation.

## Section 1.1 Problem and Definitions

In this section, we introduce the set spatial join by first discussing the motivation for the problem. Thereafter the problem is formally defined.

### 1.1.1 Motivations

The most significant application of spatial join can be found in GIS, where geographical data is stored as a series of x,y coordinate pairs representing points, lines and polygons. Map information is organized into sets of *layers* or *themes* of information. A base map can be organized into *layers* such as streams, soils, world cities, crop productivity, and administrative boundaries such as land uses, time zones, trading areas, and political areas. **Figure 1** below depicts an example of a map sheet together with its composing layers. The base map is composed by the join of four layers which are land use, soil types, crop productivity, and roads.

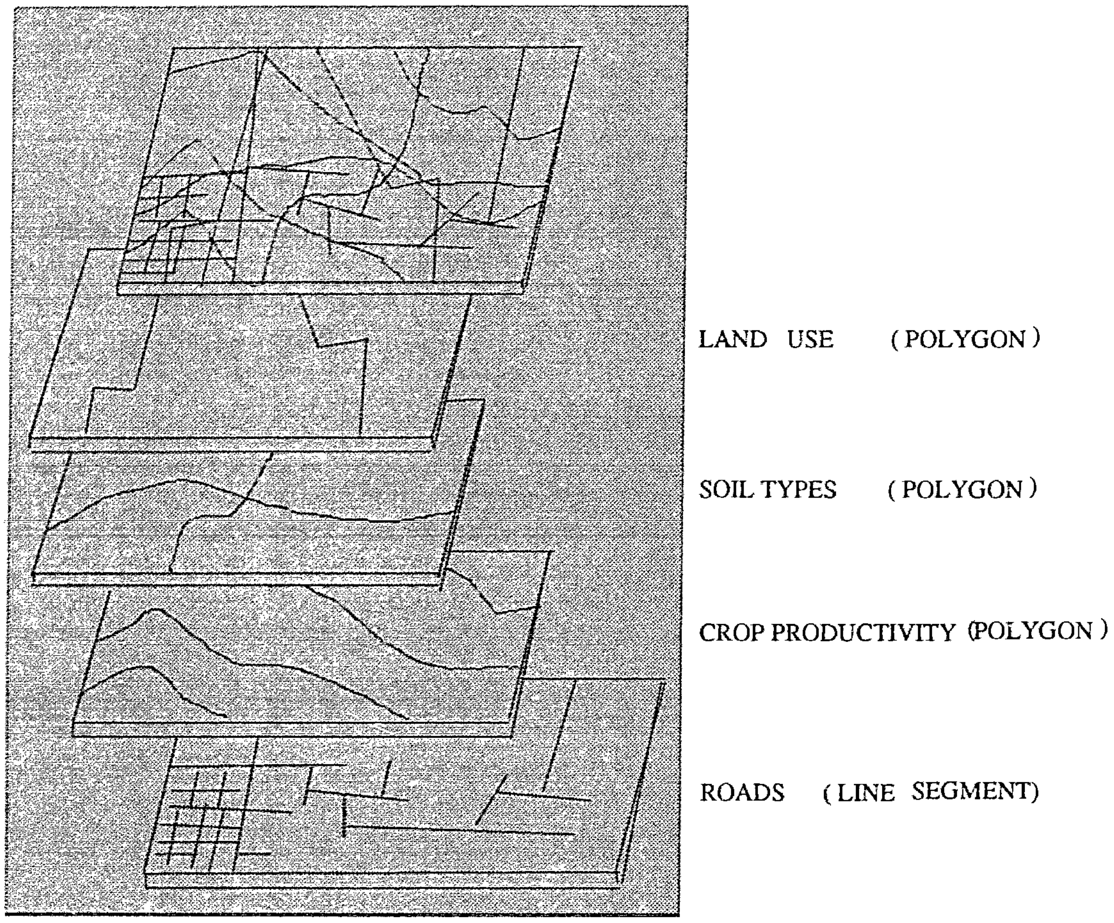

Figure 1 *Layers* in Geographical Information System

Spatial join is a very powerful operator in the sense that it synthesizes information found in multiple representations of the same geographical area, i.e., multiple *layers*, and can therefore answer complex queries. For example, the spatial join of the two *layers* will be able to answer queries about the area of the land having various combinations of characteristics, such as finding the soil types of certain land area, or determining the most productive soil for a particular crop. Joining the *layer* of world cities with the *layer* of time-zones will give cities in the world together with their time zones. Therefore, spatial join can add value to the database by combining information on different *layers*.

Spatial join can be further developed to realize *map overlay* which is a very important operation in geographical information systems. It is different in the sense that *overlay* creates all of the new regions resulted from the overlay out of the result of a spatial join. For example, the spatial join of two overlapping polygons returns both polygons together with a flag indicating they overlap. The result of overlay would be three new polygons — the common intersection, and the two polygons cut by the common intersecting area. But spatial join can not be replaced by *overlay*. When the user is concerned with only the synthesized information of multiple layers, calculating their *overlay* to answer the query would be redundant and extremely inefficient since *overlay* involves extensive computation of join results. It is necessary to separate the two operations in order to enhance the system's overall performance. In VLSI applications, some minimum separation between objects of certain layers is checked. If one data set contains 2–D objects of one layer, and the other data set contains objects of another layer, spatial join will report violations of constraints required for this minimum separation.

Spatial join could also be used in the query processing of spatial databases. The result of a spatial join can be saved as a join index, and used by the query optimizer to speed up the general query processing. Therefore, spatial join plays an essential role in spatial data management systems.

### 1.1.2 Definitions

Set spatial join is a set operation that operates on two or more sets of spatial objects. The resulting set is obtained according to spatial property of geometric data, namely intersection of spatial objects. Concepts involved and formal definitions of two-way spatial joins are defined below.

A *spatial object* is defined with respect to two-dimensional data. *Points, lines* and *polygons* are examples of *spatial objects*. A *simple polygon* is a polygon with non-intersecting edges and without holes. *Simple polygons* are *spatial objects*.

Since a *simple polygon* is the most complicated of these spatial objects defined, and the way to detect overlapping involving *points* and *lines* is different from involving only polygons, we will mainly study polygon spatial join. Our study can be easily modified to include *lines* because they are closely related to polygons. All indexing methods we study can accommodate *lines* as spatial objects, however spatial indexing methods well suited for large numbers of simple polygons may not be a good choice for lines and points.

For polygon spatial join, it is necessary to distinguish the polygon containment or enclosure from polygon overlapping on the boundary. Geometrically, containment or enclosure is detected by applying a *point-in-polygon* algorithm, while overlapping on the boundary is determined by checking boundary intersections. Henceforth in this thesis, we study two-way polygon spatial join emphasizing on the polygon boundary overlapping named *partial* spatial join as opposed to *total* spatial join. Our approach determines polygon overlap by examining boundary intersections; containment or enclosure detection is realized by performing a *point-in-polygon* algorithm on the result of boundary overlapping join.

Now we are ready to formally define the term spatial join as it is applicable to this thesis:

**Definition: Given two sets of spatial objects M and N, the Spatial Join returns all pairs of objects (m, n) such that, m belongs to M, n belongs to N, and m and n overlap spatially on their boundaries.**

"Join" here is used in a manner very similar to the natural join operator of the relational model [CODD70], except that the selection of the tuples is based on geometrical properties. The overlapping of spatial objects corresponds to shared attributes in a corresponding relational join. It can be considered a kind of cross referencing of two data sets, but the result can not be obtained directly from the search of the data.

**Figure 2** below pictures an example of the spatial join. Bold polygons represent one data set, and plain polygons represent another data set. The spatial join of the two data sets returns the set of one pair (2, 5).

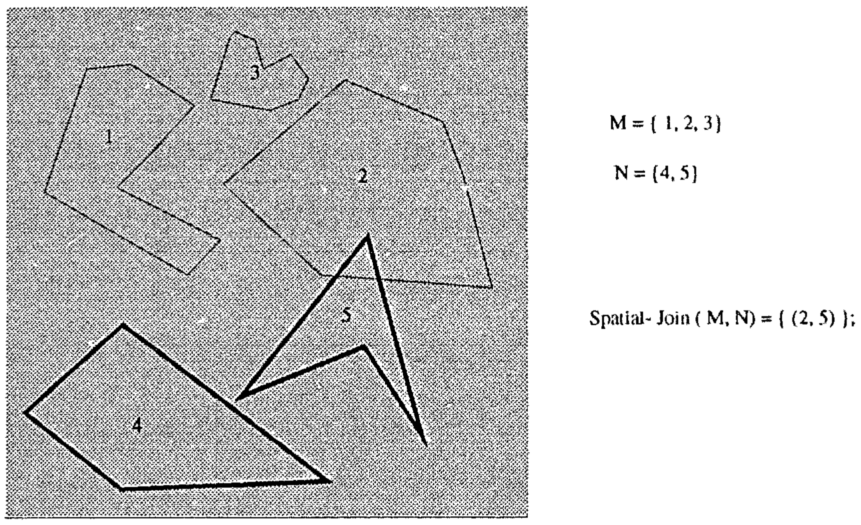

Figure 2 An example of two-way polygon spatial join

*Spatial join* can also be considered as a generalization of well studied queries such as *range Queries*. *Spatial join* is reduced to a *range query* when M has only one object, (i.e., a rectangular box representing a search area or a window), and N has a set of objects. In this case, spatial join finds all objects from N that intersect with the box or the window.

However, for this thesis, both M and N have more than one spatial object, i.e., |M|>1 and |N| >1. Typically, M and N are both map layers composed of a number of simple polygons. Therefore, the spatial join we are dealing with in this thesis is a many-to-many kind of query or a set operator. A set operator can be realized by iteratively applying the corresponding non-set operator on a single object or each pair of objects. The problem with implementations of this kind is one of performance. Algorithms which directly solve set operation will generally yield better performance.

Generally speaking, objects from the same data set can intersect each other not just over their common boundaries. But certain algorithms, including one of them developed in this thesis, require that objects in each set are strictly non-overlapping. The restriction is necessary in order to maintain segment order or to build the specific spatial indexing method as it requires. Besides, the concept of *layer* implies non-overlapping of objects in the *layer*. Therefore we assume that polygons from one data set do not overlap each other except on their common boundaries.

## Section 1.2 Literature Reviews

There have not been many studies of spatial join appearing in the literature. Two of the representative articles among those papers that do appear are the paper on PROBE by Jack A. Orenstein and Frank A. Manola [OM88], and that on PSQL by Nick Roussopoulos and Christos Faloutsos [RF88]. While practical solutions have been developed and implemented in GIS, researchers in computational geometry have been working on finding optimal solutions for basic geometry problems which can be used to solve complex problems like spatial join and overlay. They often require complex data structures and extensive processing of raw data. Popular spatial indexing methods in spatial databases operate efficiently on simple spatial objects. They can be extended to handle more complex objects.

### 1.2.1 Spatial Join in PROBE: An Object-Oriented Image Database System

PROBE [OM88] is an object-oriented image database system which deals with spatial data and data with complex data structures. In PROBE spatial objects are represented by collections of raster regions, i.e., each object is approximated by the union of cells overlapping the object. The representation is therefore conservative, and the precision is limited by the resolution of the grid. The raster approach contrasts with the vector approach where objects are precisely specified by line equations. However, both are fundamental spatial data modelling strategies.

Set operations for raster representations can be implemented simply— the same action repeats for each cell unit. For example, the spatial join can be implemented by applying the logical AND for each cell, and AND is a built-in function available in standard programming languages. Unfortunately this approach incurs large space requirements. This is especially true when grid resolution is high.

PROBE overcomes this drawback by encoding the grid. The encoding is obtained by recursively partitioning the space containing the object until the boundary of the object is obtained or the maximum resolution is reached. Each partition is just a vertical or horizontal split of the space. A vertical split is characterized by one bit from x, and a horizontal split characterized by one bit from y. The sequence of splits creates a sequence of these characteristic bits. Each cell region has a unique sequence of splitting, and interleaving of the characteristic bits will generate a bit string that uniquely identifies this region. The unique bit string is called the z value of the region.

Under this encoding scheme, 2–d objects are transformed into sets of 1–d bit strings. Instead of explicitly listing all the occupying cells, PROBE provides a more compact representation by associating the z values with the object. Spatial join can be performed by searching for any z value in one input that contains a z value from the other input. This is easily done by checking if one z value is a prefix of the other.

The Problem with this approach is that a non-point object has more than one z value associated with it; redundancy is therefore inherent. This is the problem with all of the encoding schemes trying to optimize the handling of raster data. Redundancy promotes computation overhead, yet the result is only an approximate answer due to the conservative representation. Secondly, the algorithm relies on an encoding of the objects, specifically objects have to be decomposed to obtain their z values. Spatial join is a set-to-set operation, and thus the cost of decomposition could be prohibitive. One solution is to maintain both the encoded and non-encoded representations. This however is expensive in terms of space consumed because for each of the objects, two representations are maintained.

### 1.2.2 Spatial Join in PSQL: A Relational Database System Incorporating Spatial Data Processing

PSQL is a relational database system which allows spatial data processing. 2–d objects can be directly manipulated by users through an SQL-like query language. This is realized by specialized spatial operators and functions. Good query performance is achieved by employing a specialized processor and spatial indices such as R-tree and $R^+$tree. Representation of the spatial object in PSQL is vector-based for the computation of spatial operations, though a bit-map is used for display purposes.

The vector representation strategy models a map by explicitly defining its component geometric entities, and sometimes relationships among these entities. Point, line or line segment, and region or area, are the basic elements comprising the vector data model. Although operations based on the vector form can be carried out with exact precision, a great deal of coordinate calculations are required. Especially when dealing with set operations, its raster equivalent is not only computationally simpler, but also offers a variety of analytical options by associating with each cell the attributes of interest. Generally, the computation under the vector representation is characterized by complex data structure, and complicated algorithms. Optimization strategies such as using indices to speed up object retrieval are also employed in an effort to achieve good performance.

In PSQL, spatial join was implemented by a simultaneous search on the two sets of spatial organizations corresponding to the same area. It is based on an iterative search and segment intersection checking. Secondary indices were only used to speed up the object retrieval in order to reduce the disk page accesses, but they do not contribute to the actual computation.

Savings in disk page accesses often result in performance improvement with regard to simple spatial operations which do not require significant computation. However, with complex operations like spatial join and overlay, the inside memory computation cost must be considered. It is worth the effort making an optimal or suboptimal algorithm, or accommodating an optimization strategy in order to cut down the total cost.

A problem with PSQL's implementation is that little effort has been made towards an efficient computation. However, [RF88] is currently improving the segment intersection algorithm using the existing indexing capability.

### 1.2.3 Geographical Information System

Geographical information systems (GIS) allow the manipulation, storage, retrieval, and analysis of geographical data and the display of data in the form of maps [NW79] [SE90]. Different from the conventional database systems [SM89], GIS exhibits a range of requirements and techniques known collectively as geographical data processing [ARON89] [BW90] [OOST90].

As one of the important application requirements, modern GIS has adapted practical implementation solutions of spatial join and overlay according to different data representation forms.

There are two broad approaches used by raster-based software. One is to store the raster representations in a matrix, and then examine the related matrices using the boolean operator AND. Output is produced for each cell when both values are true. Optimization strategies include reorganization of the cells so that cells with common values are grouped together to be manipulated efficiently, and encoding of the grid cells so that, as in PROBE, the representation is more compact and efficient. It should be pointed out that recently, Quadtree encoding [Same84] [Same88] [Same89] has attracted more and more attention. For example, with *region quadtree*, a 2–d raster array can be represented by its region quadtree. As a hierarchical representation, it saves space, and it is possible for set operations performed on region quadtrees to visit less nodes than the sum of total nodes of the input data.

The other approach is to use the computer's graphics system directly by converting the raster image to a screen image, then applying the same logical test AND to two screen images. The pixel will be "on" in the resulting screen if the two corresponding pixels are both "on".

Generally, software based on the vector representation requires a test to find out if any segment from polygon in one data set intersects with any segment from any polygon in the other data set. The situation can be seen in **Figure 3** below, where all the edges in polygon A have to be checked for intersection against each of the edges in polygon B.

One widely-used optimization technique is "buffering". Here complex objects are bounded by simple geometric representation, usually rectangles, and spatial join is performed on these approximations to generate a conservative answer. Only then is segment-by-segment checking carried out. This approach appears to be an efficient strategy. We develop it in this thesis by using more complicated processing techniques on the object rectangle approximations.

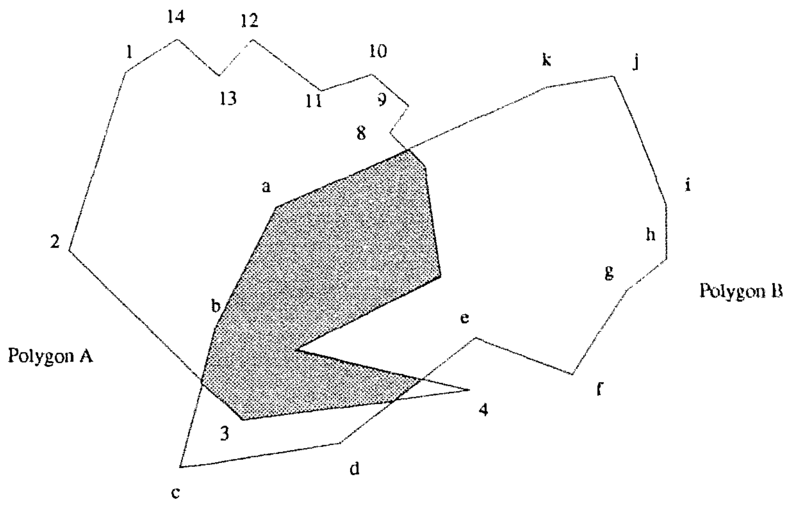

Figure 3 Testing polygon intersection by checking edge-by-edge

### 1.2.4 Spatial Databases and Spatial Indexing Methods

Many access methods have been designed for typical spatial database queries involving point data. The typical queries request all objects that contain a given point (point query) or that overlap a given search space (range query). Solutions to the efficient processing of these queries can be found in [BEN75], [SAME84], [OREN84], [NHS84], [SW88], [Krie90], etc. Since these structures were originally designed to manage point data, i.e., to provide an efficient search among large sets of points, non-point data has to be parameterized and mapped into a high-dimensional point. Segments, for example, can be represented by a point in four dimensional space, while more complex objects like polygons have to be approximated first by simple spatial objects like "box" in order to reduce the dimensionality of its representative point in the mapping space. Queries based on the approximation therefore cannot preserve the proximity.

Orenstein in [OREN90] compares the performance of the object search in the native space and the transformed space based on simple queries involving different numbers of rectangles. It is shown that the cost of maintaining two representations with many-to-many queries is high even when only searching objects. With complex operations that involve the space occupied by the spatial data, for example spatial join, the solutions are not straightforward. Retrieval is based on spatial properties not explicitly stored in the database. It is thereby inherently not appropriate to use point-based multidimensional access methods directly to solve complex spatial operations requiring space occupancy of complex objects.

To accommodate this situation, many multidimensional point access methods were extended to a spatial access method using the techniques of clipping, overlapping regions, and transforming. Performance comparisons of promising ones can be found in the paper [KSSB89]. However, the results are based on rectangles and intervals which are simple spatial objects, and range and point queries. No further research has been carried out on extending the multidimensional point access method to access complex objects like polygons, and to experiment with complex spatial operations.

Recently, spatial indexing methods based on spatial occupancy have demonstrated their efficiency with regard to optimizing range and point queries with various underlying spatial data. Spatial occupancy implies objects' locations in space and spatial relations among them. These popular spatial indexing methods, including R-tree and PM Quadtree, provide efficient retrieval of simple spatial objects like segments and rectangles. Diane Greene has provided the implementation and performance analysis of four popular spatial access methods in [Gre89]. However, the result was also based on rectangles. Access methods for more complex objects, such as the polygon, is more suitable for operations dealing with complex objects, such as polygon spatial join. However, only very few are known. The cell-tree [Gun89] is the most promising candidate, but it is still far from practical due to the high cost of building and maintaining the tree.

A very recent paper [HS92] talks about a qualitative comparison of some of the popular access methods for a large line database. The performance was again measured mainly by point and window queries, and the result only demonstrates their comparability as to when and why their performance differ. This is largely due to the high variation of both object parameters (size, shape, degree of overlapping, distribution), and the index parameters unique to each of the access methods.

As these spatial access methods for simple objects hierarchically represent the space, or partition the space from which the spatial data is drawn into regions according to their spatial occupancy, they not only overcome the drawbacks of those based on the multidimensional point access method, but also present potential for complex objects and complex operations. However, there has been little research on utilizing these spatial indexing methods for complex operations, especially polygon spatial join, and even less is known regarding the performance of these spatial access methods in complex spatial operations.

## Section 1.3 General Idea of Thesis

We were motivated by the potential of popular spatial access methods for simple objects to deal with complex objects, as well as the demonstrated performance of the "buffering" or "filtering" technique.

On one hand, the "buffering" technique provides basically a two-step processing of the data, and the answer from the first step of processing is an approximate one. This conservative result should be optimized based on the observation that candidates are selected out of the simple "boxes" — an extracted locational information from the local data structure of the complex objects. The surrogates specify object extent as well as their general locations in the global space without detail. But, since they exist independently, and representing a flat, disorganized 2–d space, the search of spatial relationships has to be performed on all the surrogates of objects in the space.

Therefore, surrogates should be organized to create more topology such as the relative position of the objects and the distribution of the objects. This global information is essential for the efficient processing of set operations. Lack of this global information results in exhaustive implementation. A good organization of the objects means a good representation of the space as a whole. The "buffering" technique itself basically sets no limit on the way these object approximations should be organized, and the geometric simplicity of these approximations also makes the intended optimization feasible.

On the other hand, popular spatial access methods like PM Quadtree and R-tree, can be extensively utilized to handle complex operations such as spatial join. This is based on the observation that these access methods decompose the space from which the data is drawn, in a way that develops and stores spatial properties, such as intersection of data or components of data. In PM quadtree, space is recursively decomposed, accommodating line segments according to their relationship to the sub-space; while in R-tree, the decomposition is dynamic, driven by the rectangle objects and relative positions of rectangle objects. Since spatial join relies on the object spatial occupancy, we believe that they are better data structures for spatial join than any point-based spatial access method or extended multidimensional point access methods.

The way we deal with complex objects utilizing simple-object-based spatial access methods is to approximate complex objects by simple objects such as rectangles which preserve both the spatial extent and the spatial relations of the complex objects. Join results of complex objects are derived from that of object components according to the topology implied, or from applying further intersection checks on the approximate results. The latter optimizes the "buffering" technique by organizing the surrogates in a more compact way towards the efficient processing of spatial join.

PM Quadtree and R-tree present hierarchical space. We present the Grid Coordinate System (GCS) which exhibits a uniformly divided non-hierarchical space composed of disjoint cell units. Unlike the Grid File, which is applied to the multidimensional data transformed from the complex objects as commonly known, GCS is applied on the original complex objects, and divides complex objects into disjoint cells according to their approximations' spatial occupancy. GCS is a spatial index method for simple spatial objects.

## Section 1.4 Thesis Overview

In this thesis, we utilize and extend the spatial access methods for simple objects such as PM Quadtree and R-tree to realize the complex spatial operation – polygon spatial join. We also develop the GCS as the version of Grid File working on original spatial data for the efficient implementation of spatial join. Not only do we show that spatial access methods for simple spatial data can be extensively utilized to handle complex spatial objects, we also provide extensive comparative studies of these spatial access methods in the context of this particular complex spatial operation. We generate random polygonal nets for the comparative studies. The polygonal net can be adjusted through parameters which modify the size, shape, and distribution of the polygonal data, and is represented by a vector data model which we design as a multi-file storage-saving structure with indexing capability. The empirical result is obtained by using ObjectStore which is an object-oriented database management system as the storage manager.

The thesis is organized as follows: Chapter 2 includes preliminaries including the file structures and experimental setup. Chapter 3 presents two pragmatic solutions of spatial join based on the file structures established in Chapter 2, and without utilizing any spatial access methods. Boundary Join is used as a baseline to compare the index solutions in later chapters. Chapter 4 introduces the basic concepts and structure of PM quadtrees, R-trees, and Grid Coordinate System (GCS). Chapter 5, 6 and 7 present the design and implementation of spatial join based on the three spatial access methods introduced in Chapter 4. In Chapter 8, the performances of the three spatial access methods for spatial join are compared, and empirical results are analyzed. Finally, we conclude in Chapter 9 that PM Quadtree, R-tree and GCS can be extensively utilized to realize spatial join, and that they improve over the Boundary Join which has no indexing involved. Both R-tree and GCS are feasible SIMs for the efficient implementation of spatial join, while for PM Quadtree, spatial join does not bring out the best of it.

# Chapter 2
# Preliminary: Experimental Setup

In this chapter, we will briefly explain the experimental setups, including the implementation environment in which the main empirical results were obtained, as well as how to generate a random net with different data distributions on this platform, and the data model we used as the internal representation of polygonal net.

## Section 2.1 Generating Random Polygonal Net

We ran the performance and comparisons on SPARC stations under UNIX using ObjectStore C++ implementations of all the algorithms. Performance were measured by the total execution time. All of the algorithms use ObjectStore as the storage manager which provides virtual memory management and scalability with large data size. We will first describe how random polygonal nets under different distributions are generated.

### 2.1.1 Randomly Generating Polygonal Net

The spatial extent of the polygonal-net is restricted to a box of size [0, 0] to the bottom left to [1,1] top right, and the origin located at the bottom left corner. A set of straight lines in the box is defined by treating their orientations and the positions of ending points on the four edges as random variables. As depicted in **Figure 1** below, there are six total possible orientations of straight lines, which are SE, SN, SW, EN, EW, WS and ending points on the edges are characterized by (x, 0), (x, 1), (0, y) and (1, y).

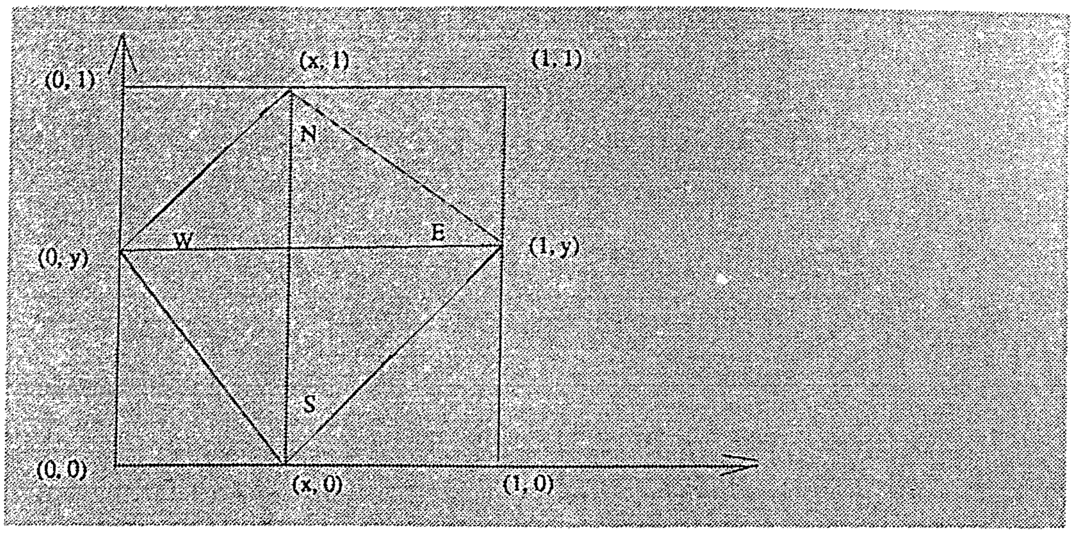

Figure 1 Orientations of Straight Lines And Equations of Ending Points

The orientations are chosen independently, having a common uniform probability distribution. For each orientation, the positions of two ending points on the two square edges follow a uniform or Gaussian distribution in (0, 1]. The number of straight lines thus generated can be decided in advance.

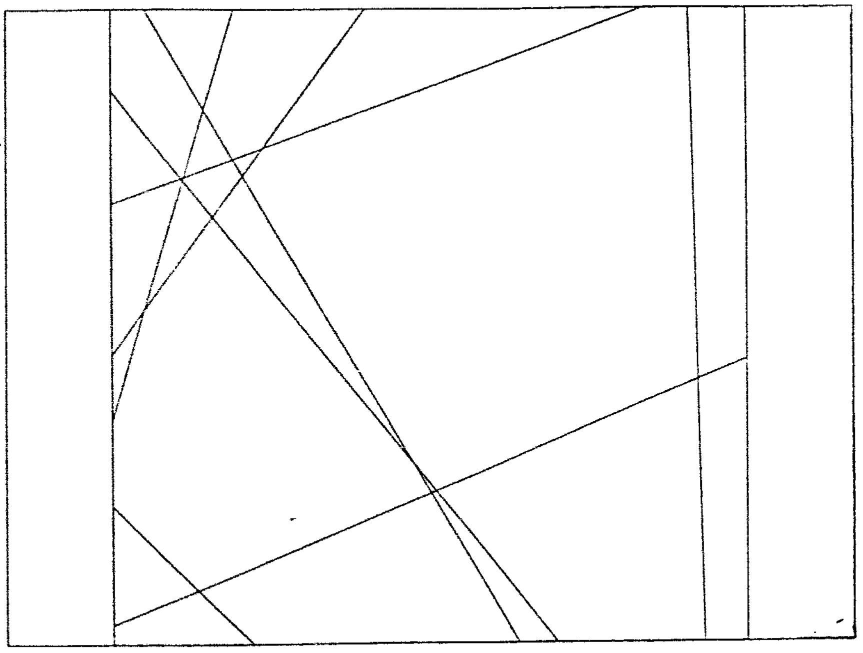

Figure 2 A Polygonal Net With Uniform Distribution Of Ending Points

**Figure 2** above samples a polygonal net with number of segments chosen as 8, and the positions of ending points follow a uniform distribution. As a comparison, **Figure 3** below shows a polygonal set with positions of ending points following a Gaussian distribution N(0.5, 0.303), but with the same number of segment lines.

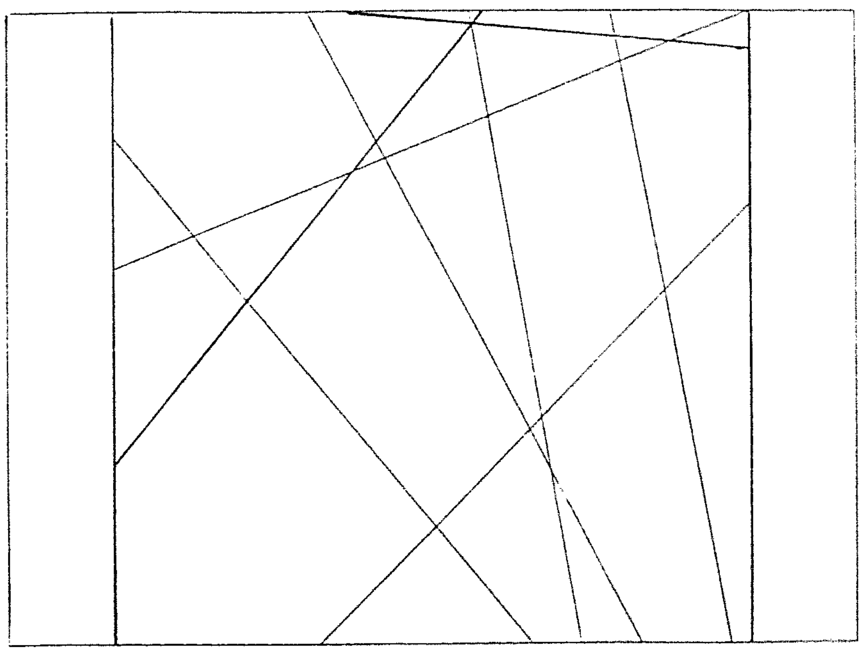

Figure 3 A Polygonal Net With Gussian Distribution N(0.5, 0.303) of Ending Points

Polygons, composed of intersecting points of the generated straight lines, were traced out after the polygonal net. To enhance the line segments to simulate natural boundaries, a third random variation was introduced on each line segment, where m number of points are added to each segment, but connected with a uniform bias towards the original straight line segment. This uniform bias can be controlled within a certain limit called *smoothness factor*, so that it curves to different extent. The *smoothness factor* ranges from 0 to 100 with increasing extent of curveness. For example, a *smoothness factor* of 0 represents straight lines with added points all on the original line segment, while a *smoothness factor* of 100 shows most curved edges composed by the additional points.

Therefore the number of points added, together with the *smoothness factor*, can be adjusted to simulate a very curved boundary or a smooth one, while the number of straight lines will directly affect the total number of polygons in the polygonal-net. Besides, to eliminate those overly small polygons thus generated, a ratio can be set as to what is the preferred relative size of the smallest polygon generated compared to that of the largest polygon generated. It was set to 0.01 for our testing data, i.e., we keep only those polygons having its area size at least 1% of that of the largest polygon generated.

**Figure 4** and **Figure 5** below present polygonal nets after the boundary modification of **Figure 2** and **Figure 3** respectively. The number of added points was chosen to be 7, and the *smoothness factor* as 80. The ratio of area of the smallest polygon to that of the largest polygon is 0.01.

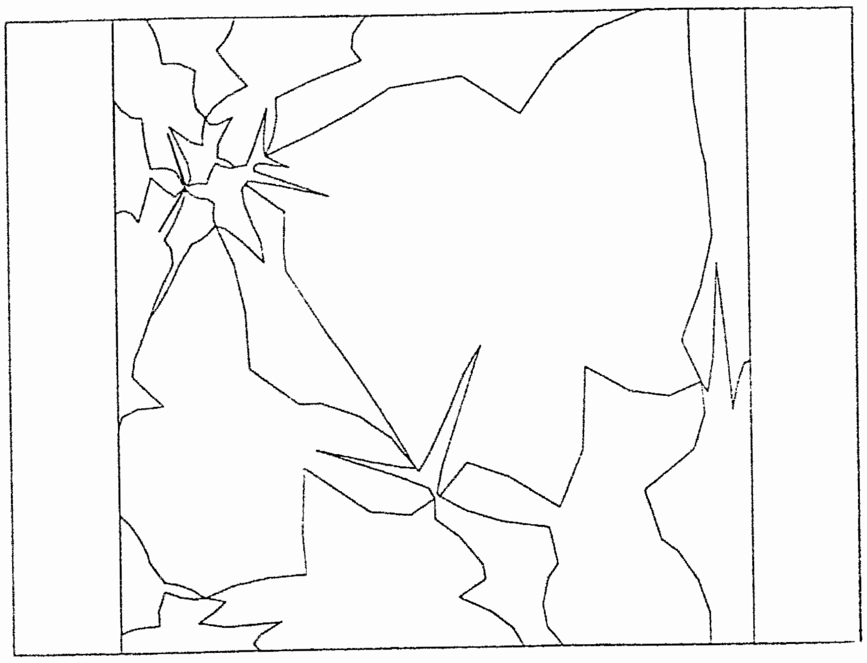

Figure 4 A Modification on **Figure 2** with *smoothness factor*=80 and (area-of-smallest-polygon/area-of-largest-polygon)=0.01

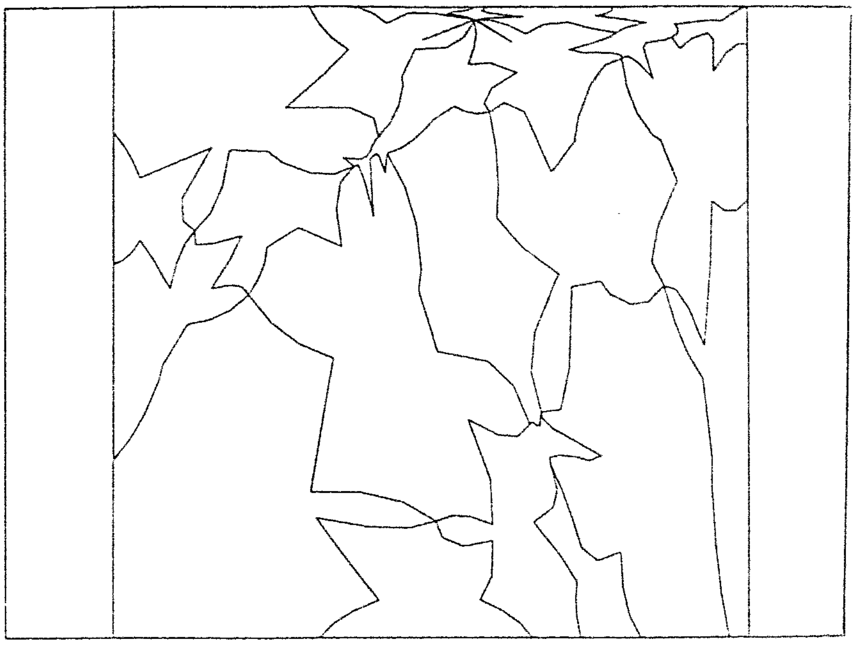

Figure 5 A Modification on **Figure 3** with *smoothness factor*=80 and (area-of-smallest-polygon/area-of-largest-polygon)=0.01

With **Figure 5**, the *smoothness factor* was also lowered down to 20 to make a comparison, and the result is shown in **Figure 6** below.

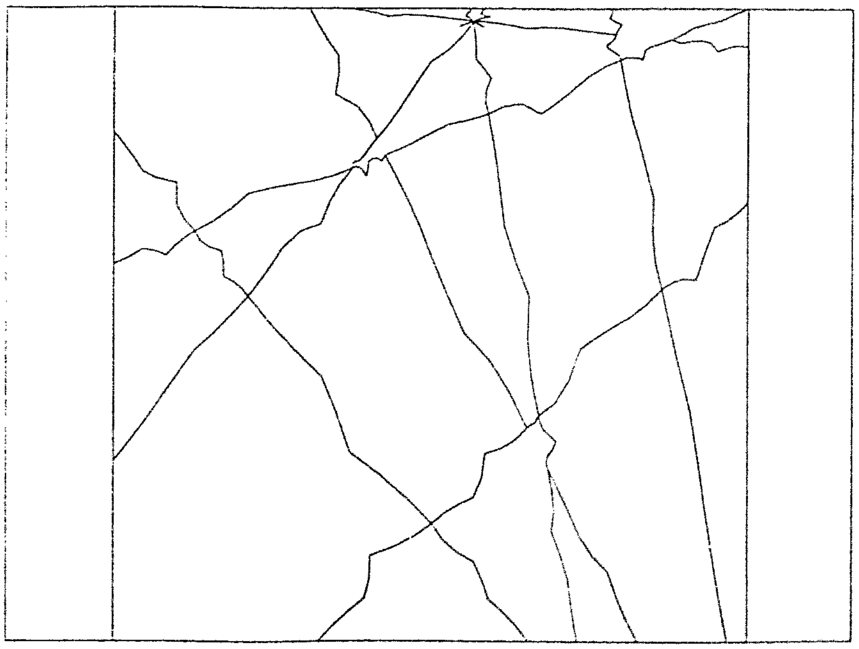

Figure 6 A Modification on **Figure 3** with *smoothness factor*=20 (area-of-smallest-polygon/area-of-largest-polygon)=0.01

### 2.1.2 Setting Up Testing Data Set

There are six sets of test data used for the purposes of the experiments in this thesis. All of them were generated under the same control variables as well as uniform ending point distribution, i.e., a *smoothness factor* 80, 7 additional points for each line segments, and a polygon size ratio 0.01. They vary only by the size of the polygonal net, i.e., the total number of polygons generated. When the ending points distribution was chosen differently, i.e., Gaussian N(0.5, 0.303), we have another 6 sets of data generated under the above circumstances. Both were used for the experiments in this thesis.

To give an idea of the testing data, statistics of six sets of data under normal distribution is listed below. Testing data under Gaussian distribution are slightly different, but not by much.

Data Set 1) $38\times100$ polygons from the two maps making 234 pairs of partial intersections. There are $105\times241$ chains having totally $945\times2169$ segments;

Data Set 2) $100\times165$ polygons from the two maps making 871 pairs of partial intersections. There are $241\times414$ chains having totally $2169\times3726$ segments;

Data Set 3) $165\times269$ polygons from the two maps making 1011 pairs of partial intersections. There are $414\times636$ chains having totally $3726\times5724$ segments;

Data Set 4) $382\times538$ polygons from the two maps making 2156 pairs of partial intersections. There are $932\times1307$ chains having totally $8388\times11763$ segments;

Data Set 5) $538\times761$ polygons from the two maps making 3438 pairs of partial intersections; There are $1307\times1858$ chains having totally $11763\times16722$ segments;

Data Set 6) $916\times1144$ polygons from the two maps making 6872 pairs of partial intersections. There are $2009\times2366$ chains having totally $18081\times21294$ segments;

## Section 2.2 File Structures: A Vector Data Model With Topological Information

The way we organize the polygonal data is a multi-file structure with indexing capability. The main entity comprising this vector data model is the *chain*. Other entities include *vertex*, *node*, *segment* and *polygon*. Definitions of these entities will be given initially, and then based on these definitions, we will explain the file structures.

### 2.2.1 Definitions and Concepts

A *vertex* is described by a ( x, y ) co-ordinate pair. A*segment* is defined as the straight line connecting two *vertices*. A *vertex degree* is defined as number of segments passing through the *vertex*. A *vertex* having a degree of at least 3 is called a *node*, or a *node* is a *vertex* that connects more than 2 segments.

A *chain* is defined as a sequence of line *segments* or *vertices* with no *vertex* in the sequence connecting more than two line *segments* except for the two ending vertices of the chain. In other words, a sequence of *segments* between two *nodes* is a *chain*.

These concepts can be pictured in the **Figure 7** below, in which a *vertex*, a line *segment*, three *nodes* A, B, C, and two complete polygon *chains* AB and AC are displayed, and A is the *starting node*, B and C are the *ending node*.

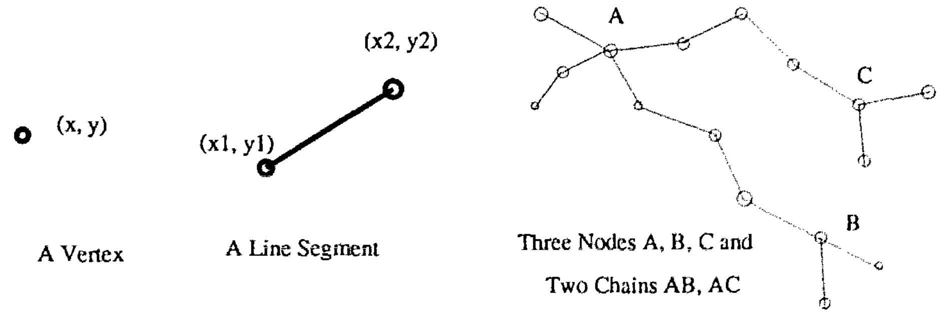

Figure 7 Entities of Vector Data Model

Notice from the **Figure 7** above that, *vertex* A connects 4 segments, so it has *degree* of 4, while B and C both have a *degree* of 3.

A *polygon* is a closed sequence of *nodes* or *chains*. In addition, each *node* is given a unique identification, and so is each *polygon* given an unique polygon identification. Each *chain* can therefore be uniquely described by a node identification pair ($N_i$, $N_j$), and each *polygon* be described by a sequence of node identifications $\{N_1, N_2, N_3, ......, N_k\}$.

### 2.2.2 File Structures

The vector data model has two types of data files: the chain file and the polygon file. Both are composed of index file and data file, so polygon file is composed of a polygon index file and a polygon data file; while a chain file is composed of a chain index file and a chain data file. The organization is shown in **Figure 8** below.

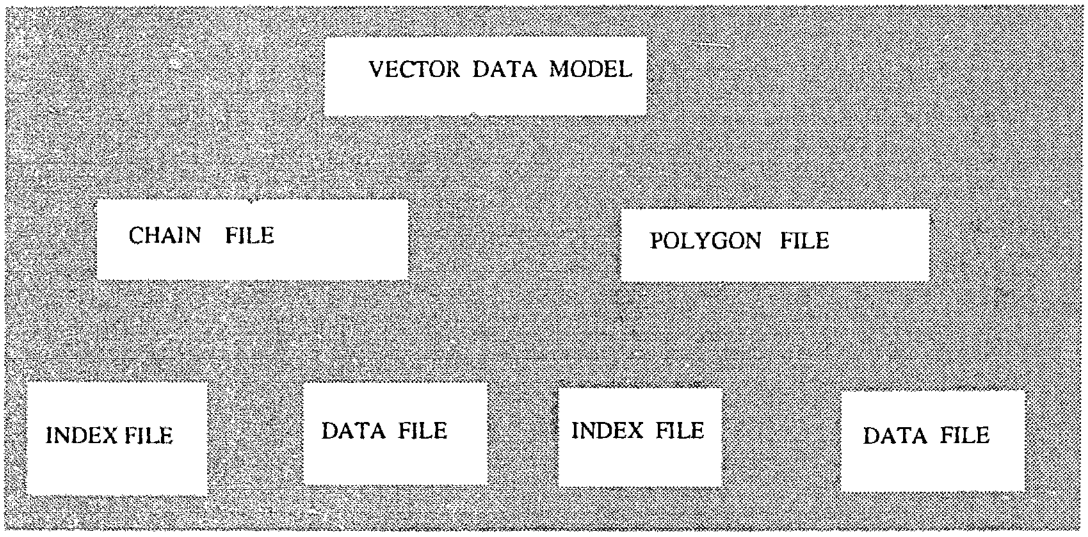

Figure 8 File Organization

Chain files contain all the polygon chains generated without duplication. In order to avoid duplication, each chain is assigned a direction from the smaller node identification to the larger node identification and is stored once and only once by preserving the direction. Since a *chain* is characterized by a node identification pair ($N_i$, $N_j$), all the chains of ($N_i$, $N_j$), where $j < i$, and i ,j < (total number of chains), are stored together in sequence under starting node $N_i$, and each of them can be identified by its ending node identification followed by its segment coordinate sequence. The number of such ($N_i$, $N_j$) pairs with $i < j$, is therefore defined as the *node degree.*, it specifies the number of chains clustered under a specific node which is different for each node.

Chain Index File is a fixed-length file recording all the *nodes* with a *node degree* at least 1. It has three fields specifying respectively the starting node identification, node degree, and first chain address in the relative Chain Data File. In Chain Data File, each *chain* is specified by its ending node id and its segment coordinate sequence, together with its right and left polygon identifications. This is used to derive information on polygon relationships from the processing of the chains. Segment sequence is a fixed-length field decided by the number of additional points added during the modification of the originally generated polygonal net. The field compositions of the chain files are depicted in **Figure 9** below.

An arbitrary chain can therefore be retrieved by looking for the index record first in the chain index file according to its starting node id, and then sequentially search for the ending node id to find its segment coordinate sequence. Since *node degree* is a small number, normally 3 or 4, the search in the data file is very efficient, compared to the situation without an index file, where sequential search is done on the whole chain data file.

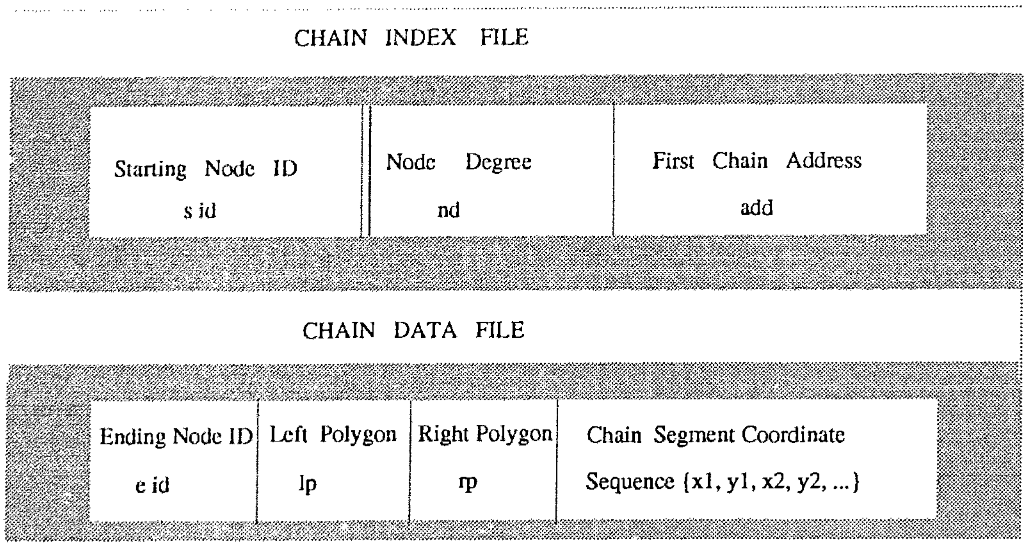

Figure 9 Chain File Data Structure

Polygon file structure is relatively simple. The index file is also a fixed-length file with three fields specifying respectively polygon identification, the number of nodes the polygon has, and the address of the node sequence in the related data file. While in the data file, each polygon is represented by its node sequence {$N_1$, $N_2$, ....., $N_k$} as well as other information associated with polygons if necessary, like the minimum bounding rectangle, or area. Again without an index file, the search of an arbitrary polygon by its polygon identification can only be done sequentially on the whole polygon data file, since each polygon has a variable number of composing nodes. The data structures of both polygon files are shown in **Figure 10** below.

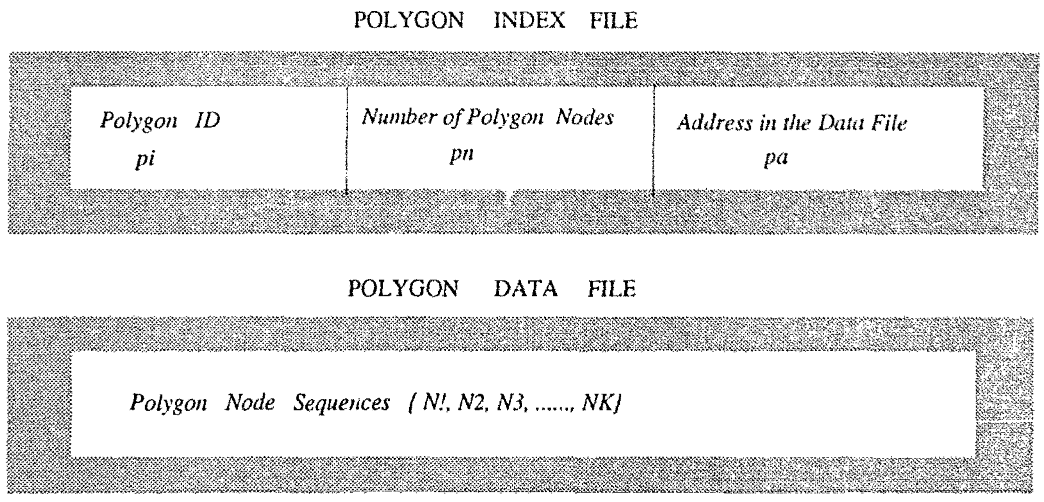

Figure 10 Polygon File Data Structure

**Figure 11** below demonstrates the complete structure of this vector data model.

To get a complete polygon coordinate representation, further search on the chain file has to be done after obtaining the polygon composing node sequence. But the duplicate storage of coordinates is therefore avoided — vertex coordinates are stored only once in the whole file organization. *Polygons*, *chains*, and *segments* are closely linked together by *nodes* to form a complete and non-redundant organization, and searching is improved by the index files.

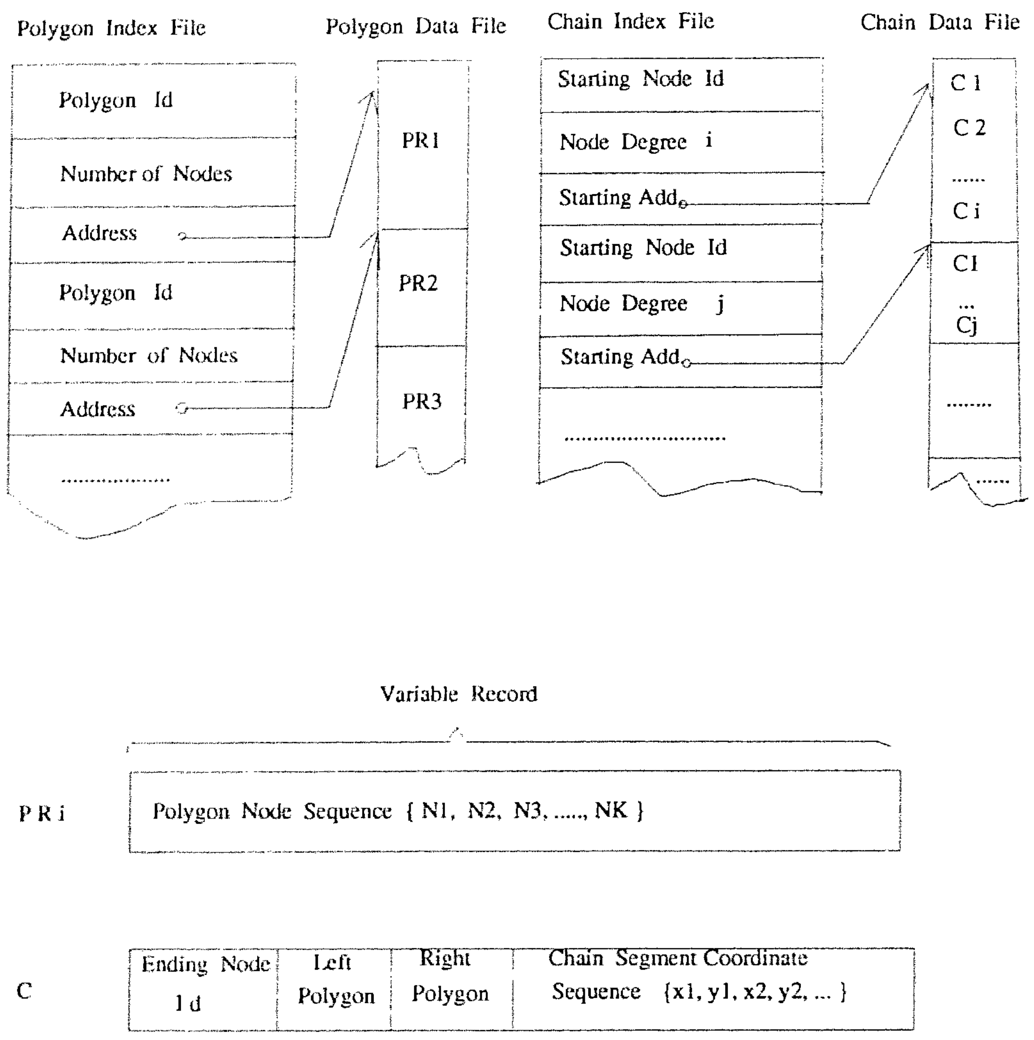

Figure 11 The complete structure of the vectore data model

# Chapter 3
# Spatial Join Without Indexing

Based on the vector data model described in Chapter 2, two solutions without using any indexing technique will be presented in this chapter: Polygon Spatial Join (**PJ**) and Boundary Spatial Join (**BJ**). Both algorithms are examples of realizing the spatial join of complex objects with only the knowledge provided by the vector data model, and without explicitly using any spatial indexing techniques. But they will provide the base line performance for other more sophisticated algorithms developed in the later chapters.

## Section 3.1 PJ: Polygon Spatial Join

Polygon Spatial Join uses a brute force approach. The algorithm iterates through all the polygons in one polygonal net or map 1, doing a pair-wise intersection checking with each polygon of the other polygonal net or map 2. The intersection of two polygons is checked chain-by-chain, and the checking of Chain intersections stops as soon as one intersection is found.

The algorithm can be described below:

```
[algorithm 3.1]

Input:  m polygons and m' segments in map 1, n polygons and n' segments;
          in map2;

Output: Set of pairs ($P_i$, $Q_j$), such that $P_i$ belongs to map1 and $Q_j$ belongs
          to map2; $0 \le i \le m$, $0 \le j \le n$;

Begin

1.  For each polygon $P_i$ from map1; $0 \le i \le m$;
2.  For each polygon $Q_j$ from map2; $0 \le j \le n$;
  21.  For each chain $C_k$ of $P_i$;
```

```
22. For each chain $C'_l$ of $Q_j$ ;
    If $C_k$ intersects $C'_l$, report intersection of $P_i$ and $Q_j$ ;
End.
```

This is a straight forward implementation which has a time complexity of order O(n'×m'). n' and m' are number of polygon segments in the two polygonal nets respectively, and each segment in map 1 has to be checked against each of the segment in map 2.

## Section 3.2 BJ: Boundary Spatial Join

Common boundary chains could be repeatedly tested for intersection during the **PJ**, for the processing is polygon-based. The vector data model provides additional topology for each chain: its left and right polygons. It is therefore possible to process each chain exactly once, and yet obtain the result of polygon intersections. This leads to the design of Boundary Spatial Join or **BJ**.

The algorithm also starts at some point of a boundary line and marches from a segment to an adjacent segment according to local conditions for each polygon in the two polygonal nets or maps. The process for each pair of polygon chains stops as soon as a pair of chain intersections is detected, and the information is recorded. In addition, since to a polygon chain is attached its left and right polygons, more results can be obtained from this topology, i.e., both polygons in one map having one of the intersecting chains as a common boundary, intersect with both polygons in the other map having the other chain as a common boundary. The situation can be shown in **Figure 1** below. The graph shows that the two bold-bounded polygons intersect with regular-bounded polygons respectively, as the result of the intersections of AB and CD.

The information of polygon intersection is kept in an $n \times m$ matrix. It is consulted each time before the pair-wise checking is performed. So only those polygon pairs that are not present in the resulting sets are actually checked, and its result is also recorded in this matrix. This matrix is implemented as a two-dimensional array with polygon identity as the index, so the intersections from the chain can be directly transferred and saved in the array, so is the retrieval of polygon intersections. All these can be done in a constant time.

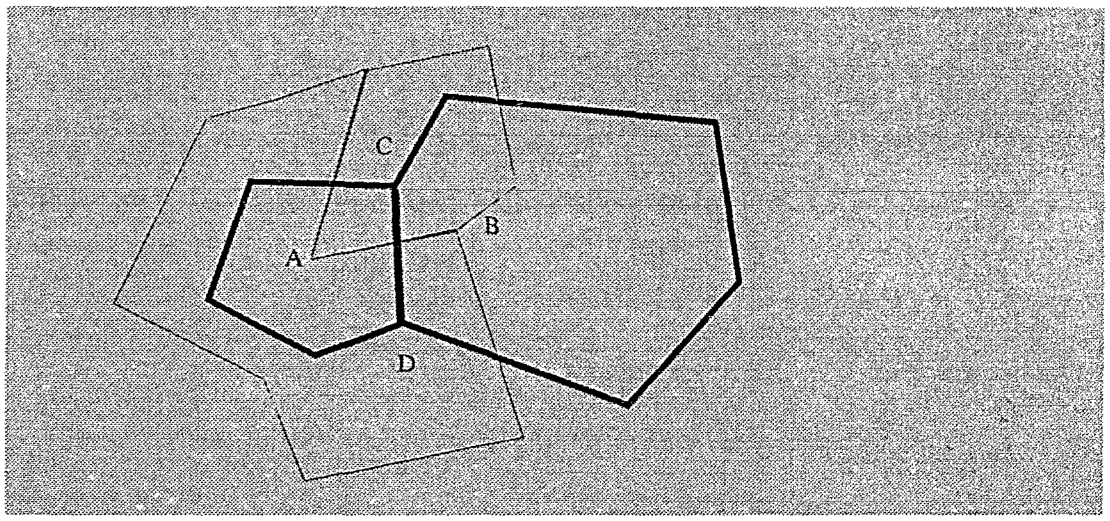

Figure 1 Intersection of Chains AB and CD Implies 4 pairs of polygon intersections

The algorithm can be described below. Time complexity of this algorithm is also of order $O(n' \times m')$ since it iterates through every polygon. By adding the result matrix, more memory is needed, but the performance is expected to be better than that of **PJ** by taking advantage of the topology implied in the data model.

**[algorithm 3.2]**

```
Input:  m polygons and m' segments in map 1, n polygons and n' segments;
        in map2;
Output: Set of pairs $(P_i, Q_j)$, such that $P_i$ belongs to map1 and $Q_j$ belongs
        to map2; $0 \le i \le m$, $0 \le j \le n$;
Begin

1. Initialize the result matrix $M_{n \times m}$ to bit 0;
2. For each polygon $P_i$ from map1; $0 \le i \le m$ ;
3. For each polygon $Q_j$ from map2; $0 \le j \le n$ ;
   If M [$P_i$ $Q_j$ ] is not set to 1 ;
   31. For each chain $C_k$ of $P_i$, if $C_k$belongs to $P_{i'}$ as well ;
   32. For each chain $C'_l$ of $Q_j$, if $C'_l$belongs to $Q_{j'}$ as well ;
       If $C_k$ intersects $C'_l$, set M [$P_i$ $Q_j$ ], M [$P_i$ $Q_{j'}$ ], M [$P_{i'}$ $Q_j$ ],
       M [$P_{i'}$ $Q_{j'}$ ] to bit 1;
4. For each M [i j ] which is set to bit 1, report intersection of polygon i and j.

End.
```

## Section 3.3 Empirical Results and Conclusion

Both PJ and BJ were implemented and tested over the six sets of data under normal distribution described in chapter 2. The result is tabled in the **Table 1** below.

| *Elapse Time (sec.)* | *DataSet 1* | *DataSet 2* | *DataSet 3* | *DataSet 4* | *DataSet 5* | *DataSet 6* |
|---|---|---|---|---|---|---|
| **PJ** | 555.59 | 1230.60 | 1667.04 | 2916.04 | 6017.55 | 8427.19 |
| **BJ** | 353.66 | 1003.41 | 1257.55 | 2026.73 | 4381.03 | 6532.26 |

Table 1 Performance Comparison of Polygon Join and Boundary Join

Although more memory is needed by BJ, its performance is indeed better than that of PJ. The type of matrix element was declared as *char* occupying one byte, and none of the maps has more than 1200 polygons. This adds up to at most

1MB, which is reasonable compared with the 64MB main memory of *phoenix* on which the algorithms were run.

However, there is a large amount of computation overhead with this method due to its lack of space organization. **Figure 2** below illustrates the situation in which polygon A from the other map has to be checked with all the polygons in area C of the map having polygons from c1 to c8, which are in fact far apart from polygon A.

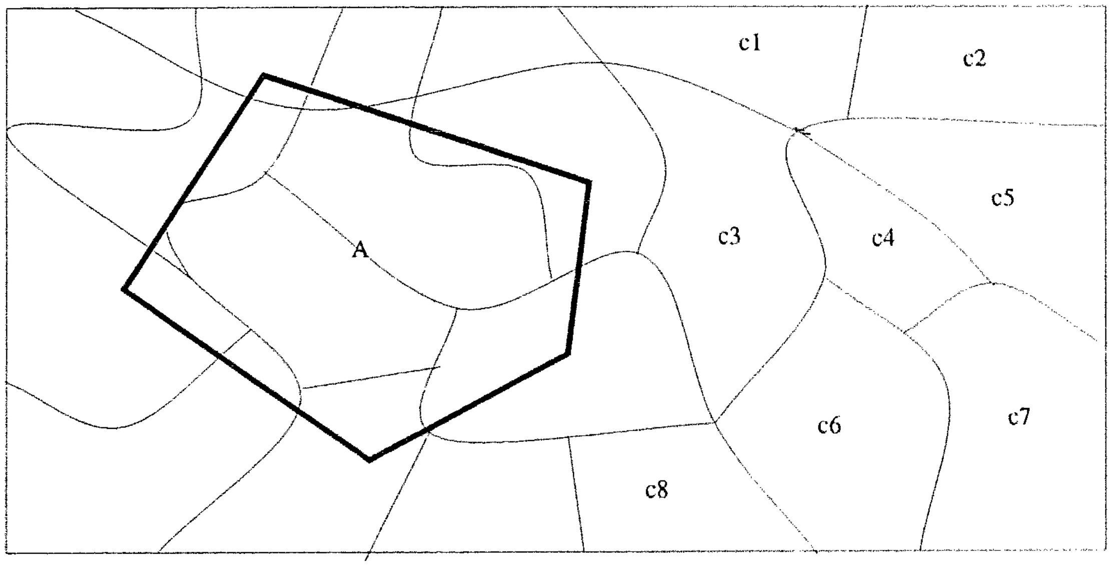

Figure 2 Checking Redundancy

The polygon address space in this case, that is the way polygons were accessed, has nothing to do with polygon locality. Polygons are merely iterated individually. Although polygon locality is reflected by polygon coordinate list, the coordinates are only used when actual intersection checking is performed. Deliberate utilization of the locality will make a better polygon address space.

In the following chapters, we will introduce different spatial indexing techniques in order to realize spatial join efficiently. Since *Boundary Spatial Join* has better performance over *Polygon Join*, we will use **BJ** as the base line performance for the complex spatial join algorithms developed in the following chapters. However, to be clearly distinguished, we will use **Boundary-Join** instead of **BJ** in the later context.

# Chapter 4
# PM Quad-tree, R-tree and GCS

The spatial indexing method we chose first is PM Quad-tree family. It is a compact hierarchical representation of polygonal maps based on recursive data partitioning, and PM stands for "polygonal map". R-tree is another hierarchical data representation, while Grid Coordinate System uses uniform grid. Related concepts and terms about these SIMs will be explained in this chapter to make the algorithms developed on them in the later chapters easy to be understood.

## Section 4.1 PM Quad-tree Family

PM Quad-tree family [Same85] [Nels86] represents an improvement over *edge quadtree* [Shne81] [Warn69]. They both focus on a representation that specifies the boundaries of areas, but PM Quad-tree is an exact representation of collections of polygons, not an approximation one, like in *edge quadtree*, a vertex is represented by a pixel. So PM Quad-tree can apply directly on our vector data model.

The basic entities of PM Quad-tree are vertices and edges, since no isolated vertex exists in our case, a complete PM Quadtree is constructed by inserting all the polygon edges into it. The construction requires non-intersection of existing edges themselves. Edges are inserted into a PM Quad-tree by searching for the position they are to occupy. This is done by traversing the tree in preorder and clipping each edge against the block. Segment of an edge resulting from the clipping of the edge on the border of the block is termed as *q-edge*. The clipping stops when conditions on a number of *q-edge*s in each block holds, otherwise, the block is successively decomposed into four equal quandrants and clipping is applied on each of them. This is called leaf splitting.

Different stopping conditions comprise different decomposition criterions, which make the variants of PM Quad-tree: PM1, PM2, and PM3.

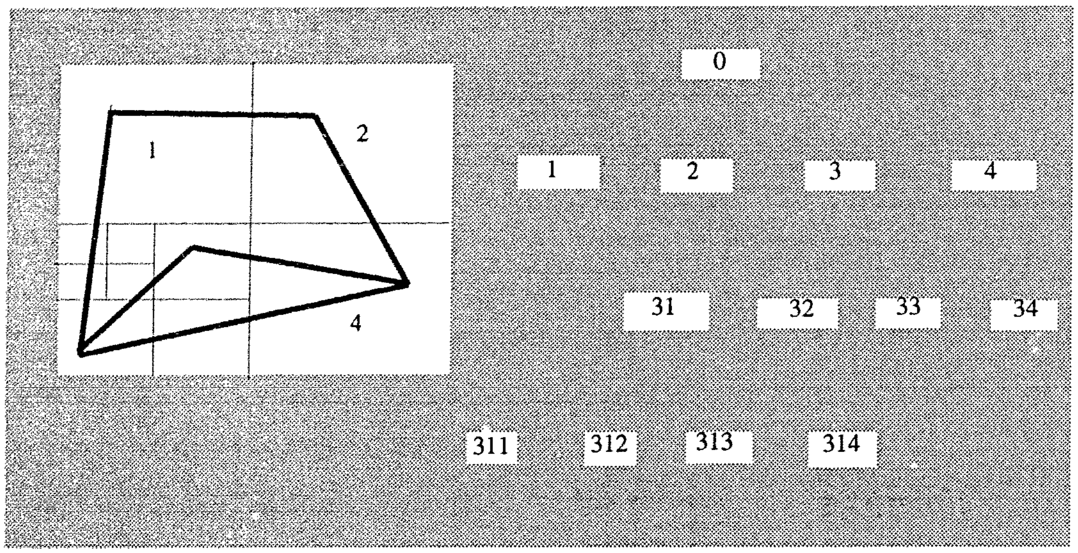

Figure 1 An Example of PM1 Quad-tree

**Figure 1** above is an example of PM1 Quad-tree. Each decomposition block is represented by a node in the tree. There are two types of nodes: leaf node (*white* node), and nonleaf node ( *grey* node). Non-leaf nodes contain pointers to the four sons corresponding to the direction NW, NE, SW, SE, while leaf nodes contain collections of *q-edge*s called *dictionary* associated with the leaf node. Both types of node also contain information about the block they represent, i.e., the size and center of the square. This is for the clipping and further splitting.

In our experiments, *q-edge* is merely a pointer to the edge it belongs. Clipping is performed as a checking, no actual *q-edge* is obtained. *Dictionary* contains these *q-edge* pointers linked together by OS-list provided by ObjectStore implementation. This saves time and reduces redundant storage.

Polygon chain is used as the basic unit when constructing the tree. Each edge however is associated with its belonging chain id, so that the information extracted from *dictionary* checking can be directed to the polygon information based on our vector data model.

## Section 4.2 R-tree

R-tree is another type of popularly used hierarchical spatial indexing method derived from B-tree dealing with rectangular data. R-tree is a multi-level tree structure designed to handle n-dimensional objects originally proposed in [GUTM84].

Although R-tree is derived from B-tree, unlike B-tree, the search of a specific rectangle or the search of set of rectangles in the tree may often require several nodes to be visited at each level before ascertaining the rectangles to be visited at the next level. This is because the intermediary nodes on a given level can overlap, therefore their rectangles do not represent disjoint regions. The more serious the overlapping is, the more nodes have to be visited, the larger the search space.

Besides the root node, R-tree has leaf node and non leaf node. A non-leaf node contains entries of the form (*Child, Rect*), where *Child* is the address of a child node, and *Rect* is the minimum bounding rectangle of all rectangles which are entries in its child node. A leaf node contains entries of (*Object-id, Rect*), where *Object-id* refers to certain object , and *Rect* could be the object or object's MBR. All leaves of R-tree appear on the same level.

An example of R-tree is pictured in **Figure 2** below.

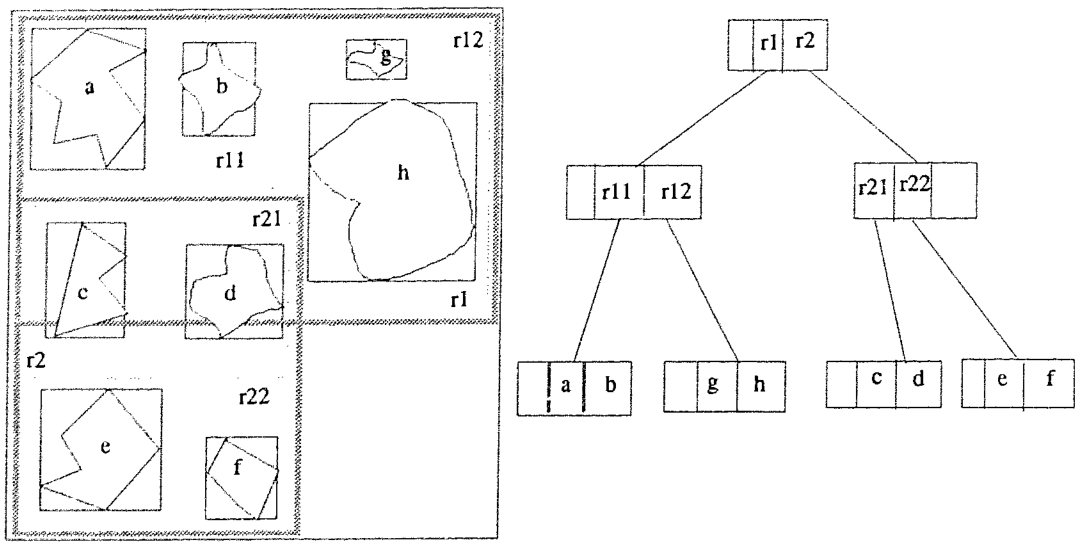

Figure 2 A demonstration of R-tree structure

All the non-leaf nodes are assigned a minimum and maximum number of entries. The maximum entries allowed is defined as *node size*. If *node size* is M, each non-leaf node should have at least [M/2] entries. If m = [M/2], then the order of the tree is defined as (M, m).

## Section 4.3 GCS: Grid Coordinate System

GCS is orthogonal grids. Under this representation, a two dimensional space is covered by a flat grid with equidistant on both X and Y axis. When superimposed on a polygonal net, objects which could be polygons, chains, or segments, are then divided into cell groups according to their cell occupancy. Grid cells are numbered in a way by row and column and objects belong to the same cell are stored in the cell index array by the object indices.

*Grid resolution* is defined as G×G, where G is the number of grid cells in one row or column, and 1/G is defined as *grid size*. Number of objects belonging to a cell is defined as *object grid volume*, so there are *polygon grid volume, chain grid volume*, and *segment grid volume*, depending on the object chosen. Pair of (*cell* *index, object index*) is also defined so that the total number of (*cell index, object index*) pairs can be used to measure the amount of computation with regarding to different *grid resolution* and different data size.

A different *grid resolution* generates a different *grid volume* for same data set. The smaller the *grid volume*, the less computation will be performed in each grid. Resolution affects on the volume can be pictured in the **Figure 3** and **Figure 4** below. When grid resolution is chosen 2×2 in **Figure 3**, the average *segment grid volume* is 8/4=2, while in **Figure 4** when *grid resolution* is 4×4, the average *segment grid volume* turns into 14/16=0.875.

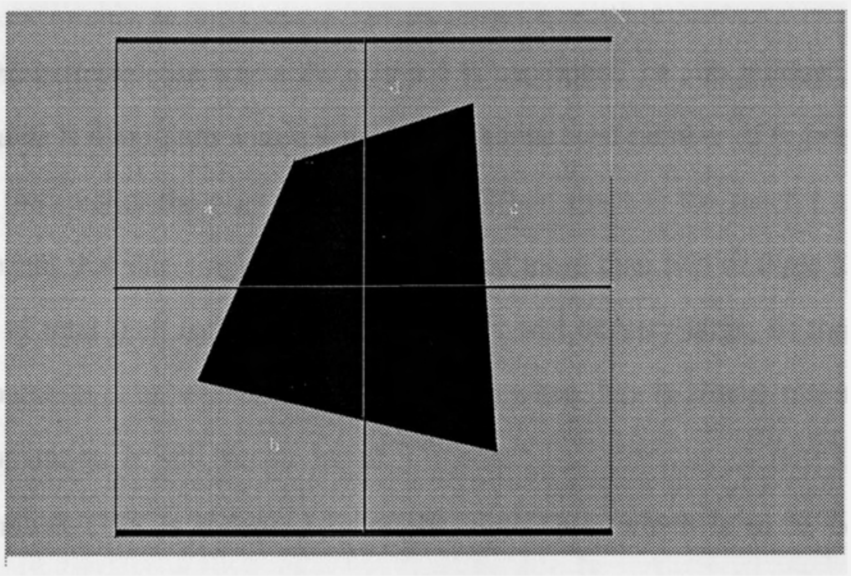

Figure 3 grid resolution = 2×2, edge grid volume = 2

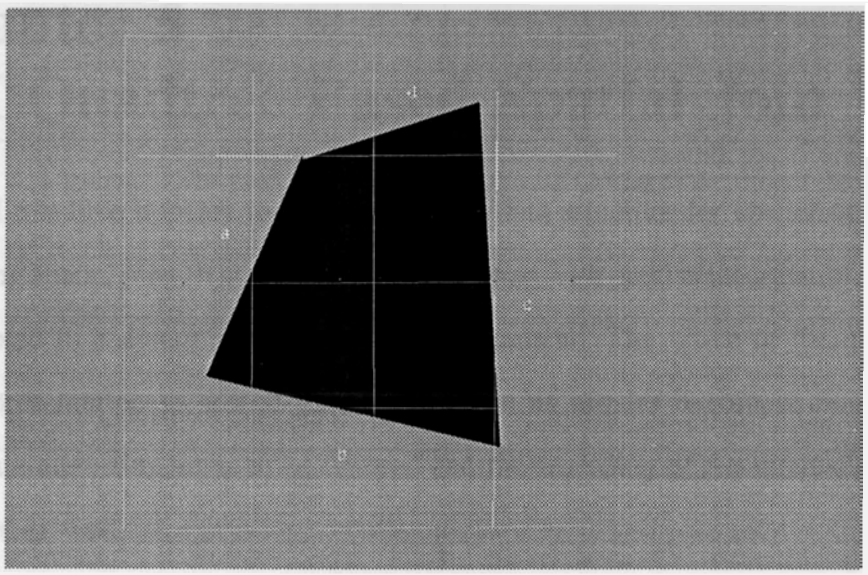

Figure 4 grid resolution = 4×4, edge grid volume = 0.875

In the extreme case where the *grid size* is maximum, i.e., no griding at all, the *grid volume* is maximum which is equivalent to the total number of segments. But not that the smaller the *grid size*, the better. When space is too much fragmented to the extent that one single segment is clipped more than two or three times, the number of total (*cell index, object index*) pairs will be very large, so the amount of computation is unbearable. This is the case when data is almost rasterized but the computation is still vector based.

Although GCS is independent of the input data, which is to be converted according to its own data volume and data distribution, performance of any operation based on this schema relies on choosing a appropriate *grid size* according to the volume and distribution of the input data. It is therefore necessary to study the statistical aspects of the input data in order to achieve good performance.

# Chapter 5
# PM Quadtree Based Spatial Join

PM quadtree appears to be an attractive data structure for spatial operations including spatial join. It stores the polygonal map with precise information and can be adapted to a dynamically changing environment. The virtue of the quadtree-like representation to spatial join operation is its regular decomposition, which makes uninteresting areas to be ignored and the searching of the interesting areas efficient as well.

Although there has been some research and empirical results on the performance of some spatial operations like point location under quadtree representations, we are concerned about the performance of different spatial join algorithms under PM quadtree representation and the performance of certain spatial join algorithms under different PM quadtree variants resulting from different decomposition criterions. Our goal is to find an efficient method to perform spatial join which would result from combining the best algorithm with the optimal PM quadtree variant.

## Section 5.1 PM Quadtree Based Top-Down Traversal Algorithms

We are presenting three algorithms to perform spatial join. They all feature top-down tree traversal no matter which PM quadtree variant is used. But different strategies are used with respect to what object is used during the traversal and whether further leaf splitting is performed.

### 5.1.1 PMSJ: Parallel Traversal With Splitting Join

This is an algorithm performed on two Quadtrees at the same time. First, Quadtrees for each of the two polygonal maps are constructed. Then the algorithm traverses the two Quadtrees in parallel. Only corresponding quadtree nodes at the same level are compared. When one tree is a leaf and the other tree is not, the leaf is split into a node with four sons, each of which is leaf node. The procedure is then applied recursively to the corresponding sons. When both Quadtrees are leaf nodes, the dictionary of one of the Quadtrees is checked against that of the other for possible intersections of the segments, and should any intersection occur, the intersection of the corresponding polygons is recorded.

The algorithm is presented as below:

```
[algorithm 5.1]
Input: m polygons and m' segments in map 1, n polygons and n' segments
    in map2;
Output: Set of pairs (Pi, Qj), such that Pi belongs to map1 and Qj belongs
        to map2; $0 \le i \le m$, $0 \le j \le n$;
Begin
    1. Construct Quadtree Q1 for map1;
    2. Construct Quadtree Q2 for map2;
    3. Starting from root of Q1 and root of Q2, compare corresponding nodes;
        if both are grey, go to the next level, check corresponding nodes;
        else
        if both are leaf, perform dictionary to dictionary check and report
        intersections, return;
        else
      if one is grey, one is leaf,
          split the leaf, generating four new leaves, and comparing
          leaves with the corresponding sons of the grey node.
End.
```

The fact that only leaves at the same decomposing level are compared and their corresponding dictionaries are checked for the possible intersections, makes the traversal very expensive. This is because both tree nodes are further decomposed down to the level of whichever is deeper when the corresponding nodes are not at the same leaf level, and each time with each node the splitting is done by decomposing the present square into four equal quadrants, passing the dictionary to each of them and clipping all the line segments in the dictionary against them. The Quadtrees from the two different maps eventually turn into exactly the same and the maximum decomposition schema and levels.

### 5.1.2 PMNSJ: Parallel Traversal Non-Splitting Join

**PMNSJ** is designed to eliminate the over-splitting of **PMSJ** and makes use of the detailed representation provided by the decomposition.

When corresponding quadtree nodes at the same level are compared with each other, no splitting is done if one tree is a leaf and the other tree is not. Instead, it continues traversing down the other quadtree to its leaf level, and then compares all the dictionaries along the traversal of the tree to that of the leaf node from the other quadtree; i.e., the leaf dictionary of one of the Quadtrees is compared with all the dictionaries of the subtree with the corresponding node as the root. Although there is more dictionary checking due to the fact that traversal of any one of the two quadtree stops as soon as its leave is reached, there is no further splitting of the quadtree, which involves much more computation than that of the dictionary checking.

The algorithm is presented below as well:

**[algorithm 5.2]**

**Input**: *m polygons and m' segments in map 1, n polygons and n' segments in map2;*

**Output**: *Set of pairs (Pi, Qj), such that Pi belongs to map1 and Qj belongs to map2;* $0 \le i \le m$, $0 \le j \le n$;

```
Begin
    1. Construct Quadtree Q1 for map1;
    2. Construct Quadtree Q2 for map2;
    3. Starting from root of Q1 and root of Q2, compare corresponding nodes;
       if both are grey, go to the next level, checking corresponding nodes;
       else
       if both are leaf, perform dictionary to dictionary check and report
       intersections, return;
       else
       if one is grey, one is leaf,
         perform dictionary to tree checking;
           continue traversing down the quadtree with grey node, until leaves
           are reached. Whenever a leaf is reached, perform the dictionary to
           dictionary checking and report intersections, return.
End.
```

Dictionary to tree checking is done by a recursive procedure *DictionarToTreeCheck(Dictionary, Quadtree).* It calls itself until the leaf of the *Quadtree* is reached. So the intersections are checked between the Dictionary and all the dictionaries of the *Quadtree.* Back to the **PMNSJ**, it implies that whenever the corresponding tree nodes are not at the same level, i.e., one has reached its leave, but the other has not, the dictionary of the leaf node from one of the Quadtrees is checked against all the dictionaries under the corresponding grey node from the other quadtree. The subtree under this grey node is traversed and all the dictionaries are visited.

Below is the pseudo code for the *DictionarToTreeCheck(Dictionary, Quadtree).*

```
Procedure DictionarToTreeCheck(Dictionar1, Quadtree2);
Begin
```

```
If Is_Empty(Dictionary1) Then
        Return
If Is_Leaf(Quadtree2) Then
{
        DictionaryToDictionaryCheck(Dictionary1,Quadtree2->Dictionary );
        Return;
};
For (j=o;j<NumberOfSons;j++)
        DictionaryToTreeCheck(Dictionary1, Quadtree2->Son[j]);
End.
```

Procedure *DictionaryToDictionaryCheck(Dictionary1, dDictionary2)* sequentially traverses two lists and checks for the edge intersection. Since the number of edges in each dictionary is small, a complex data structure is not necessary in this case.

### 5.1.3 PMIJ: PM Index Join

This is a method which uses quadtree as an index to search those edges that are most likely intersecting the present edge.

First, a quadtree of the one of the two maps is constructed. Second, an attempt is made for each chain from the other map to insert into the quadtree already built, but no actual insertion is carried out when leaf level is reached, rather the intersections against all the edges inside the dictionary are checked.

The algorithm is presented as below:

**[algorithm 5.3]**

```
Input: m polygons and m' segments in map 1, n polygons and n' segments
    in map2;
Output: Set of pairs (Pi, Qj), such that Pi belongs to map1 and Qj belongs
    to map2; $0 \le i \le m$, $0 \le j \le n$;
Begin
```

```
1. Construct Quadtree Q1 for map1;
2. For each chain in map2, starting for the root node of Q1, examine the
   chain segment against the node square;
      If the node is grey, clip the chain segment against the node square;
         if clipped, go to the next level of the grey node;
         if not clipped, return;
      if the node is leaf, perform dictionary checking between the chain
      segment and the dictionary of the leaf, report intersections;
      return;
End.
```

Step 2 is implemented by a recursive procedure *QuadtreeInsertCheck (AChain, Quadtree)*, in which the chain from one map to be checked with the present quadtree of the other map, is clipped against the blocks of the Quadtrees starting from the root, and only blocks that are clipped by the chain are further traversed until their leaves are reached. Then intersections are checked between the chain and the dictionaries of the leaves. By clipping the chain along the existing quadtree, not only the areas that have no intersection with the chain are avoided, but also the clip is conducted in the existing quadtree schema which leads the direct mapping of the possible intersecting leaves to the chain. Furthermore, there is again no additional splitting during the traversal.

The code for **QuadtreeInsertCheck** is presented below.

```
Procedure QuadtreeInsertCheck(Chain, Quadtree2)
Begin
ClipLine(Chain, Quadtree->Block, result-list);
If Is_Empty(result-list) Then
        Return;
If Is_Leaf(Quadtree2) Then
{
        DictionaryToDictionaryCheck(result-list, Quadtree2->Dictionary);
```

```
            o
        Return;
    };
    For (j=o;j<NumberOfSons;j++)
        QuadtreeInsertCheck(result-list, Quadtree2->Son[j]);
    End.
```

### 5.1.4 Empirical Results

Each of the three algorithms **PMSJ, PMNSJ, PMIJ** described thus far were implemented based on PM1 Quadtree for comparison purposes. The same six sets of test data in chapter 3 were used here as well. Execution times were measured in seconds. **Figure 1** below pictures the performance curves of each of the algorithms.

Notice that the second algorithm **PMNSJ** *(Parallel Traversal Without Splitting)* improves over the first algorithm **PMSJ** *(Parallel Traversal With Splitting)* as we expected because further splitting is avoided, and replaced by dictionary checking, which are less expensive than clipping and splitting.

The result also shows that **PMIJ** outperforms both of the parallel traversal algorithms **PMSJ** ,and **PMNSJ**. By clipping the chains from one of the maps along the existing quadtree from the other, those dictionaries that the chain is to be inserted are the most likely intersecting ones with respect to the specific chain, and not the whole dictionary as is the case in the second algorithms. Although we don't save much by not constructing the second quadtree because the chain is to be clipped along the existing quadtree anyway, more accurate relative objects are obtained by this clipping, and the redundancy is further reduced.

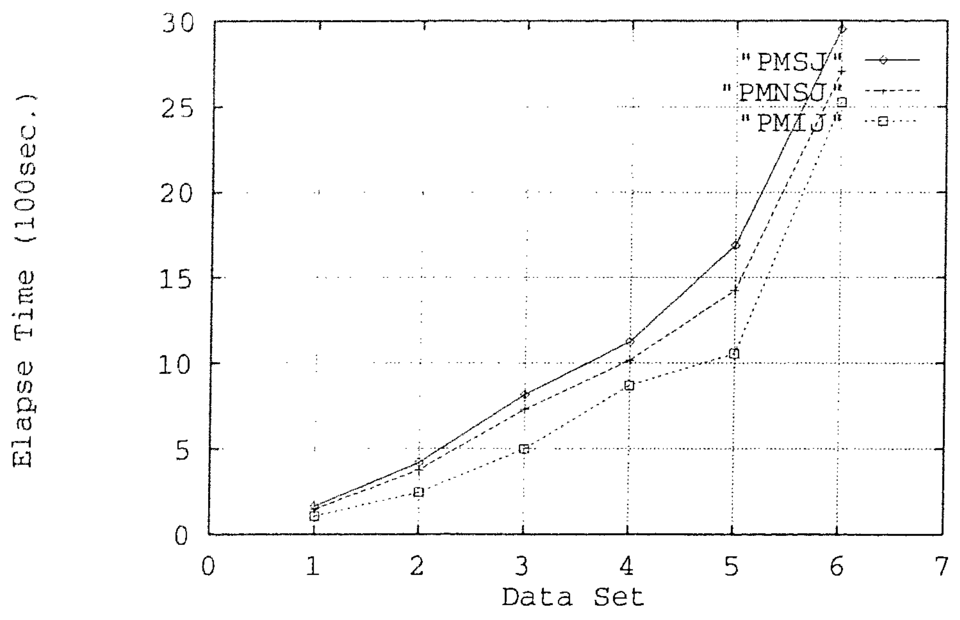

Figure 1 Performance of **PMSJ**, **PMNSJ**, and **PMIJ**

Another important fact is that the performance of all three algorithms drops considerably as the number of polygons and edges are increased. It is due to the fact that the quadtree representation occupies large amount of spaces. It provides detail spatial information of a map down to lccations of segments of each edge. When edges are heavily fragmented, it is repetitively stored in the quadtree much more than once, since each leaf allows only one segment unless more than one segments originate from the same point with PM1 Quadtree. But the extent of the fragmentation can be reduced by adapting a less restrictive decomposition criterion. This results in the experiments with different quadtree variants.

## Section 5.2 Comparison Of the Algorithms on Different Quadtree Variants

There are three variants of PM Quadtree developed by **Samet and Webber** under different decomposition criterions . The PM2 and PM3 quadtree are obtained by successively weakening the definition of what constitutes a valid leaf node, resulting in PM Quadtrees with less depth and less leaves. Although with PM1 quadtree, more detailed information could be obtained, chain segments are overly fragmented such that redundant intersection checking can not be ignored in the performance analysis as far as spatial join are concerned.

In this section, we discuss the way that different decomposition criterions affect the performance of the spatial join, and conclude with one PM Quadtree as the choice for spatial join. Empirical results will also be presented to verify our analysis.

### 5.2.1 Analysis of PM Quadtrees in Spatial Join

Among the three variants PM1, PM2, and PM3, the PM3 quadtree has the least requirement for segments in one block. It only limits the number of vertices in each block area without the requirement to the segment across the block, while PM1 Quadtree allows only one segment across the block, and PM2 Quadtree allows more than one, but they have to meet at a common vertex exterior to the block.

All of the three variants of the PM Quadtree present exact map data. What would be the effect if different variants of the Quadtrees are used in our spatial join algorithms?

In spatial join, we are concerned about how much benefit can be obtained from this decomposition schema, such as how fast the interested area can be reached and how accurate the information is, in the sense that those chain segments appeared in the reached block area, are closely located to the searching chain segment. They are the segments that most likely intersect this segment than that appeared in any other block areas which can not be reached by this segment.

By this observation, quadtree is actually used as a space decomposition method in our algorithms. The space is decomposed regularly into four equal quandrants upon the arrival of the new chain segment and the violation of the criterions. Clipping and node splitting make the chain segment appear in more than one node, thus introduce redundancy. The space should be decomposed to such an extent that the map chain segments are divided into groups by their spatial locations under the quadtree decomposition schema, and there shouldn't be too much redundancy with each chain segment resulting from node splitting to meet the criterions.

Under quadtree decomposition schema, the more node splitting that happens, the deeper the tree is, and hence the more fragmented the chain segment would be. Since PM3 Quadtree has the least requirement for the ending block, it results in least number of node splitting, but yet it decomposes the space to a quite fine extent for spatial join, with each block containing maximum one vertex. A study by *Samet and Webber* shows that from PM1 to PM3 quadtree, there are up to 19% reduction in depth and leaves with a cityline map. According to our analysis with the spatial join, PM3 quadtree would have the best performance with any of our algorithms discussed in the last section.

In general, redundancy can hardly be avoided due to the fact that it is very difficult to have a variable space division of the space to include the polygonal chains into non-intersecting groups, unless each chain segment is made one group unit, in which case it is reduced to the brute-force performance. The way quadtree divides the space is not variable, but the extent the space is decomposed can be controlled, so is the redundancy resulted from the decomposition, but very limited.

### 5.2.2 Empirical Results

To verify our conclusions, the three algorithms were implemented on PM2 and PM3 Quad-trees as well. The same six data sets were used and the results were grouped by algorithms, i.e., we compare the performance of the same algorithm on different PM Quadtree variants.

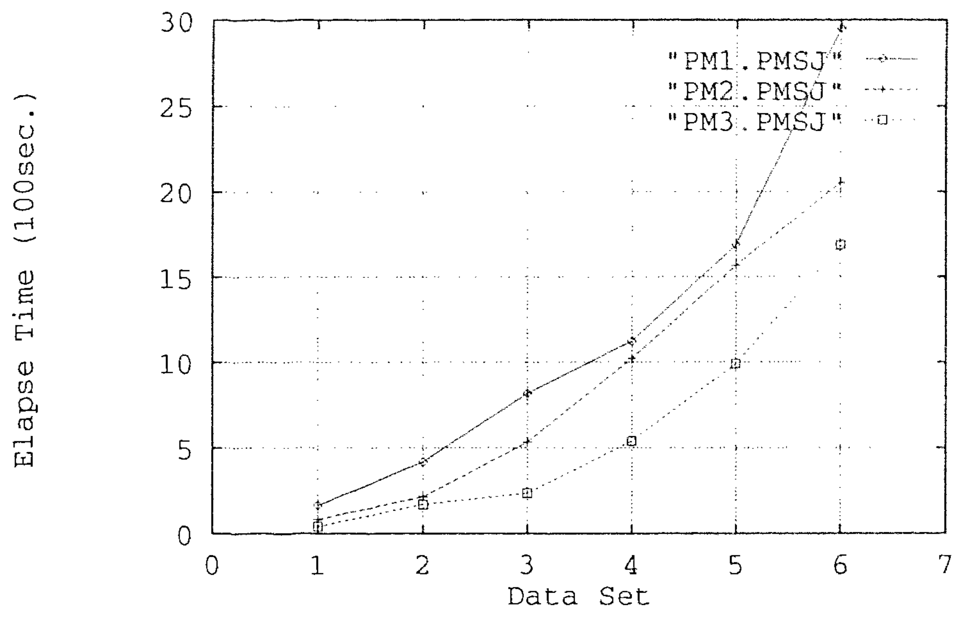

Figure 2 Performance of **PMSJ** on PM1, PM2, and PM3 respectively

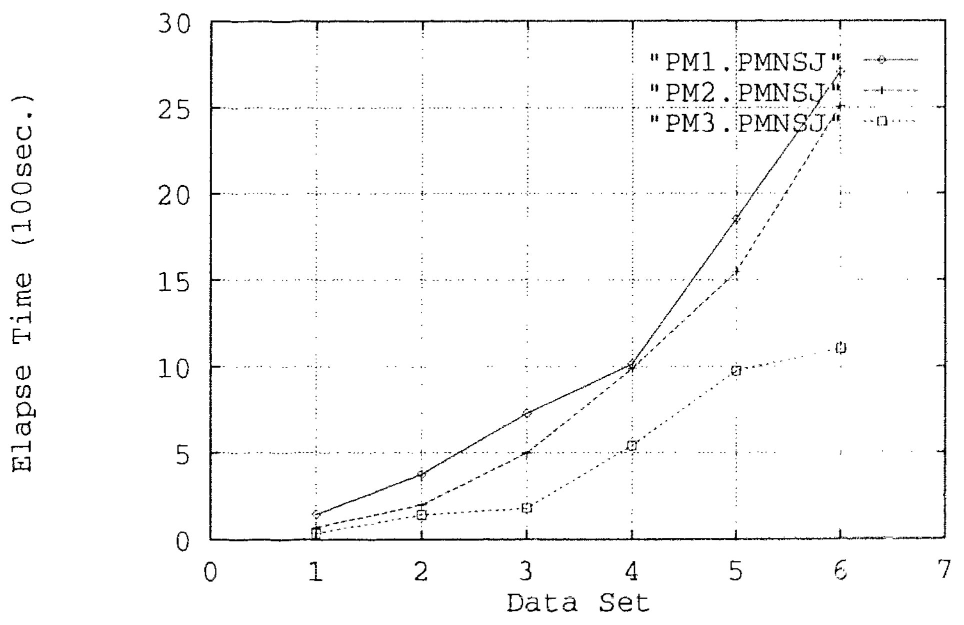

Figure 3 Performance of **PMNSJ** on PM1, PM2, and PM3 respectively

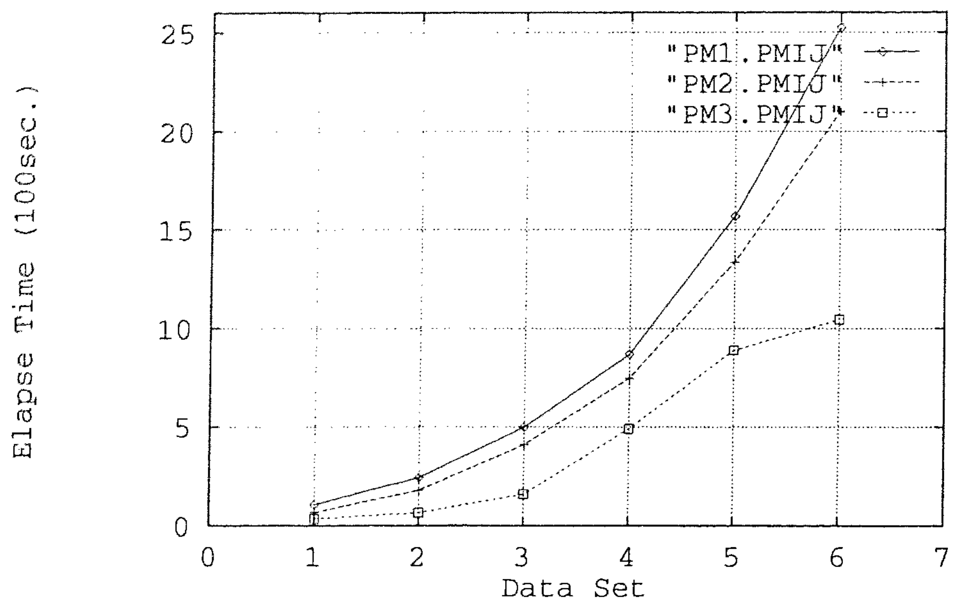

Figure 4 Performance of **PMIJ** on PM1, PM2, and PM3 respectively

**Figure 2** above shows the performance curves of **PMSJ** on PM1, PM2, and PM3 respectively. **Figure 3** shows that of the **PMNSJ**, and **Figure 4** presents results of the **PMIJ**.

Overall, PM3 quadtree has the best performance with any of the three algorithms, which verifies our conclusion in the above section. Furthermore, when tabling the PM3 Quadtree performance data of the three algorithms in **Table 1** below, it is also the **PMIJ** that showed its best performance.

Although space requirements are reduced with both PM2 and PM3 Quadtrees, which improves the performance with large data sets, the overall performance is still not satisfying. We look at ways to further improve it in the following section.

| **PM3** *Elapse Time (sec.)* | *DataSet1* | *DataSet2* | *DataSet3* | *DataSet4* | *DataSet5* | *DataSet6* |
|---|---|---|---|---|---|---|
| **PMSJ** | 41.08 | 169.37 | 236.421 | 539.2 | 989.34 | 1687.41 |
| **PMNSJ** | 36.80 | 143.08 | 181.42 | 542.15 | 976.50 | 1103.42 |
| **PMIJ** | 35.04 | 65.55 | 159.76 | 489.73 | 887.43 | 1045.33 |

Table 1 Performance of **PMSJ**, **PMNSJ**, and **PMIJ** on PM3 Quadtree

## Section 5.3 PM3IJ: A PM3 Quadtree Based Spatial Join Algorithm

So far we have concluded, that **PMIJ** *(Quadtree-based index join)* implemented on PM3 Quadtree generates a best result among the three algorithms implemented on different PM Quadtree variants. In this section, we present a practical solution based on **PMIJ** of its PM3 Quadtree variant, by deferring the *dictionary* checking, in order to reduce the amount of computation resulting from the redundant storage.

### 5.3.1 PM3IJ: Two-Step Processing of Spatial Join

Careful study of the algorithm **PMIJ** shows that, even with PM3 Quadtree in which chain edges are least fragmented, each edge may appear in more than one leaf node, therefore more than one *dictionary* contains the same chain segment. Since actual intersecting points are of no interest for spatial join, the *dictionary* checking is inevitably redundant in this way. Although the storage of redundant information can not be avoided, the redundant checking of the intersections could be replaced by a two-step process, which will separate the process of grouping the map chains from the process of the actual checking.

An extra data structure is needed to keep track of the possible intersecting chain segments during the traversal of the quadtree for each chain segment. The repetitive information will be filtered out by this structure, and each possible candidate will be recorded only once. Actual intersection checking would not take place until the end of the chain traversal, and then the structure is cleared and reused for the next chain traversal.

The data structure we used is linked list. Again a complex data structure is not necessary, since there are generally a small number of segments in each list, and each segment in the list has to be traversed. If a polygon chain has *m* segments, then there are *m* such linked lists recording a very fine collection of candidates. These m lists are organized in chain segment order, so corresponding checking can be performed between each segment and its candidate lists. Intersections are then collected from this checking as usual.

The algorithm is presented below:

**[algorithm 5.4]**

```
Input: m polygons and m' segments in map 1, n polygons and n' segments
       in map2;
Output: Set of pairs (Pi, Qj), such that Pi belongs to map1 and Qj belongs
       to map2; $0 \leq i \leq m$, $0 \leq j \leq n$;
Begin
    1. Construct PM3 Quadtree Q1 for map1;
    2. For each chain $C_i$ in map2, starting from the root node of Q1;
            2a. For each segment $s_j$ of $C_i$, initialize list $L_j$
                If it's grey, clip $s_j$ against the node square;
                    If clipped, go to the next level of the grey node;
                    If not clipped, return;
                If it's leaf, for all the segments in the leaf dictionary,
                    If it's already in $L_j$, return;
                    If not in the list, add $s_j$ to $L_j$; return;
            2b. For each $s_j$ of $C_i$, and its corresponding list $L_j$
```

```
                Apply intersection checking. and report polygon intersections;
                Clear the list $L_j$;
End.
```

Step 2a is also implemented by a recursive procedure **QuadtreeInsertCheck**, in which the *dictionary* checking is replaced by a procedure *AddCandidateToList ( result_list, Candidates_list )*. As a comparison, its pseudo-code is also presented below.

```
Procedure QuadtreeInsertCheck(Chain, Quadtree2, Candidates_list)
Begin
ClipLine(Chain, Quadtree->Block, result-list);
If Is_Empty(result-list) Then
        Return;
If Is_Leaf(Quadtree2) Then
{
        AddCandidatesToList(result-list, Candidates_list);
        Return;
};
For  (j=o;j<NumberOfSons;j++)
        QuadtreeInsertCheck(result-list, Quadtree2->Son[j]);
End.
```

In the implementation of **PM3IJ**, memory for the additional lists is dynamically allocated and deallocated. Besides, the number of segments of each chain is limited, so is the number of lists needed. The following sub-section presents our empirical result and the concluding remarks for this chapter as well.

### 5.3.2 Empirical Results and Concluding Remarks

When applied to the same sets of testing map data, the optimized solution showed its improvement over the non-optimized. It is compared with all the three algorithms based on PM3 quadtree. The result is pictured in the **Figure 5** below.

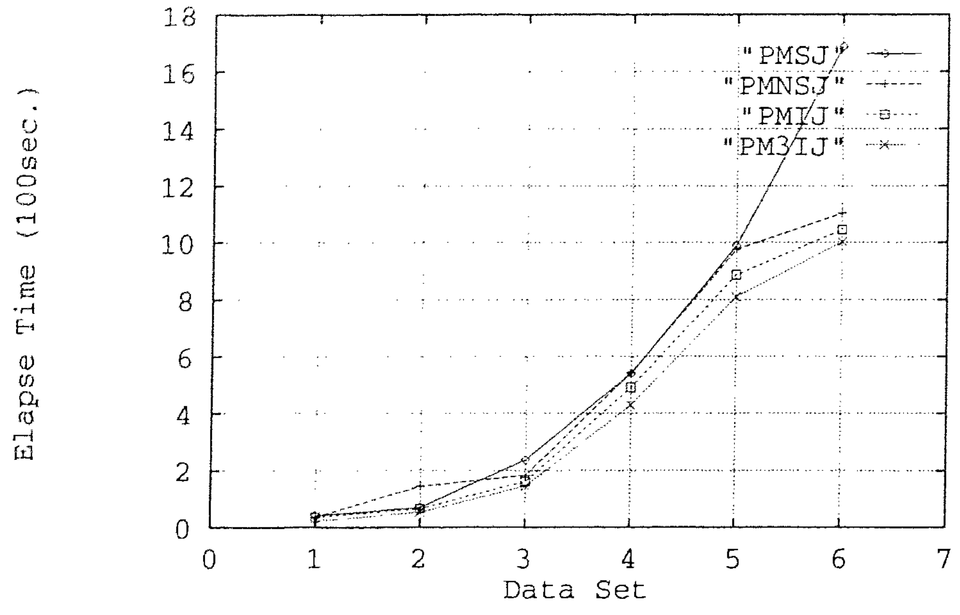

Figure 5 Comparing **PM3IJ** with **PMSJ**, **PMNSJ** and **PMIJ** of PM3 variant

A satisfying performance is obtained when smaller data sets are applied. e.g., with a 38 × 100 data set, the joining pairs can be computed in less than 30 sec. Because it doesn't consume less space, actually more by using additional data structure, the performance is not improved much when large data sets are applied.

The conclusion we can draw from the above experiments is that, quadtree is a fine, dynamic spatial data structure. Polygons, lines, and points can be inserted into and deleted from the tree dynamically without having to rebuild the tree. As it is a hierarchial data structure, the search of the object can take advantage of the tree structure.

It is also an expensive data structure because the computation needed to build up the tree takes approximately 60% of the total constructing time. e.g. with data set 38 × 100, the time spent to build the first quadtree is 34.8813 seconds, and the clip-line function takes 25.5577 secs, which is 73.2% of the total constructing time. The average time percentage of the clipping over the constructing time with 5 of the polygonal nets is 66.1955% (see **Table 2** below).

| *Elapse Time (sec.)* | *Net 1* | *Net 2* | *Net 3* | *Net 4* | *Net 5* |
|---|---|---|---|---|---|
| **building** | 34.8813 | 86.2634 | 232.42 | 449.238 | 636.274 |
| **clipping** | 25.5577 | 52.1516 | 154.1457 | 307.263 | 397.8768 |
| **clipping/ building $\times$100%** | 73.2704 | 60.4562 | 66.3220 | 68.3965 | 62.5323 |

Table 2 Clipping occupies over 60% of the total construction time

On the other hand, when we use it as a space decomposition method for the spatial join, the space is decomposed into such a fine extent that the edge is subdivided and their location recorded. The decomposition is overhead for the spatial join, hence the space requirement is too high, which results in unsatisfactory performance with large data sets.

For the later comparison with other spatial join algoritms based on different spatial indexing methods, we will use **PM3IJ**, and name it **PM-Join** as a notation of **PM3IJ**.

# Chapter 6
# R-tree Based Spatial Join

As with Quadtree, R-tree can also be used as a space decomposition method to perform spatial join as well to incorporate topology into the processing of object intersecting detection. But it is different from Quadtree in the way the spatial object is represented and the way the space is decomposed and organized. These are the factors that affects our solution for the spatial join operation utilizing R-tree indexing structure.

## Section 6.1 Choose Processing Unit of the R-tree Representation

Possible processing units are polygons, chains and segments (refer to **Figure 1**), all of them can be represented by R-tree data structure, i.e., representing them as 2–d minimum bounding rectangles, which results in polygon R-tree, chain R-tree or segment R-tree. This is different from PM Quadtree's segment and vertex based only representation for 2–d objects. By choosing different objects as the processing unit for the spatial join, we will have different R-tree nodes and different R-tree representations for the same polygonal net, as well as different outcomes from the tree processing.

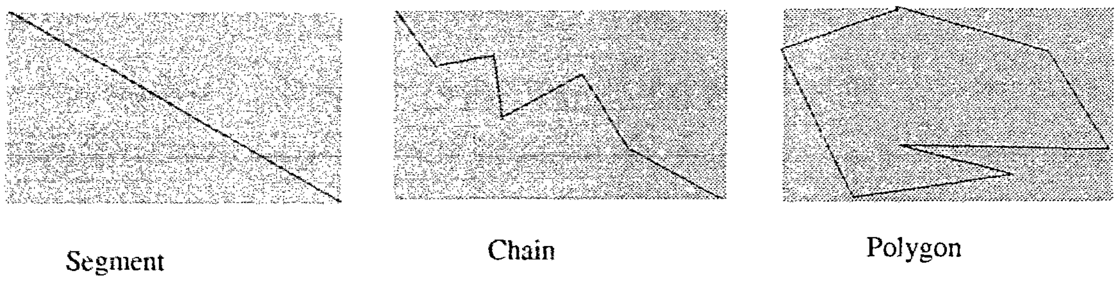

Figure 1 MBRs for different processing unit

When chain is used as the processing unit, for the map that is to be presented, there are obviously more MBRs than there are with polygons as the processing unit, which will result in a larger R-tree, as it is assumed that polygon is composed of chains. Under the assumption, we generate polygonal net with the number of chains in the net approximately three times more than that of polygons, but only *1/n* of that of segments, where remember *n* is chosen to simulate the natural boundary and $n \geq 0$.

This is observed from only the representation point of view. As far as spatial join is concerned, the objective is polygon oriented, i.e., pairs of polygons that intersect. By choosing polygon chain as the unit, the polygon intersections will have to be recollected after the R-tree processing.

There are advantages with Grid Coordinate System when polygon chain is chosen as the processing unit. We present the advantages in the next chapter. However, to directly generate polygon candidate pairs, no advantages can be taken here by choosing either segment or chain as the processing unit. The result is a larger R-tree and computation overhead. Therefore, polygons should be the processing unit, the result from the R-tree processing could be collected and made use of directly in the later processing stage.

## Section 6.2 Two-Step Processing of Spatial Join

Our approach to the spatial join involving R-tree as part of the data representation features two-step processing: the preprocessing step as a filter to find the possible intersecting sets, i.e., where their MBA's overlap; and the polygon-by-polygon intersection checking as the second step. It is during the first step that R-tree is utilized as a spatial data structure to provide additional topology on polygons' locations and relative positions. We will present two different algorithms of using R-tree to fulfill the first step.

In this section, the general strategy of spatial join processing is presented, followed by the description and analysis of two algorithms.

## 6.2.1 General Outline of the Processing

Under our strategy, map data is composed of two levels as one integrity:

Map Data Structure = vector Data Structure + Global Topology; Or

Map Data = Vector Data + R-tree Representation;

Hence, each polygon is represented both as coordinate list in vector form and as MBRs of different levels in the R-tree, while the R-tree itself represents the whole map the polygon belongs to. Vector data structure stores precise information of each polygon in the map, while global topology stores polygon approximations and reconstructs them into a tree according to their space occupancy. The spatial relationship of the polygons is therefore constructed through their approximations, and provides additional topology for the spatial join operation. The approximations are boxes such that each edge is parallel to one axis of the two dimensional space.

The whole processing can be stated as follows:

[Step 1] *Find out all the pairs of polygons which could be overlapping potentially according to their approximation constructs from the two maps. This includes a design of two algorithms involving R-tree: Index Join and Parallel Join.*

[Step 2] *Find out the exact pairs of the overlapping objects by checking all the pairschain-by-chain from the result of the first step, and calculate the intersection information like the boundary or the area of the resulting pairs if needed.*

Let $n_1$, $n_2$ be the number of polygons in each polygonal net respectively, and $h_1$, $h_2$ the maximum heights of their corresponding R-trees. If M and m are defined as the maximum and minimum entries allowed per node respectively, then $h_1$ and $h_2$ can be calculated as $[\log_m(n_1)]$ and $[\log_m(n_2)]$.

If $n_1'$ and $n_2'$ stand for the maximum number of non-leaf nodes in the R-trees of the two polygonal net respectively, then *Index Join* has the worst case complexity $O[n_1 \times (n_2+n_2')]$ and best case complexity $O[n_1 \times h_2]$; while *Parallel Join* has the worst case complexity $O[(n_1+n_1') \times (n_2+n_2')]$ and best case complexity $O[Max(h_1 \times h_2)]$. Maximum total number of nodes in a R-tree with N rectangles can be calculated as $[N / m] + [N / m^2] + \ldots + 1$.

The analysis above shows that number of polygons and maximum entries allowed per node are critical factors of the algorithms' performance in the two extreme cases. Maximum entries allowed per node is defined as *node size* in our context. So the analysis leads to a choice of large *node size* in the two cases. However, in the average case, the way the *node size* affects the performance is not so straight forward, and the overlapping extent between sibling rectangles at each level is neither non-overlapping as in the best case, nor all overlapping as in the worst case. We will further discuss them by conducting experiments on *node size* in the later sections.

*Index Join* based spatial join is named *R-tree Index Spatial Join* or **RIJ**; while *parallel join* based spatial join is named *R-tree Parallel Comparison Spatial Join* or **RPJ** in our following context.

### 6.2.2 RIJ: R-tree Index Spatial Join

R-tree is used as an index in this algorithm. One of the maps having larger number of polygons is represented as a R-tree; Each of the polygon MBRs from the smaller polygonal net, is used as a known space to traverse the R-tree, starting from the root, to find out all the polygons that could overlap it.

Suppose each node of the R-tree has maximum M entries per node, each entry will be checked for the overlapping possibility because the enclosing rectangles from each entry of the nodes on the same level are non-disjoint. If the entry is intersecting, all the entries in the child node pointed by this entry will be checked until the leaf level is reached. By checking the intersection with all the entries in the leaf node, polygon identities can be obtained and resulting pairs are formed. If at any level, the entry is not intersecting, the whole subtree under this entry will not be examined and all the polygons at the leaf level will be exempted from the resulting set.

The algorithm is presented below:

```
[algorithm 6.1]
Input: m polygons and m' segments in map 1, n polygons and n' segments
          in map2;
Output:Set of pairs (Pi, Qj), such that Pi belongs to map1 and Qj belongs
          to map2; $0 \leq i \leq m$, $0 \leq j \leq n$;
Begin
    1. Construct R-tree r2 for the map2 that has larger number of polygons;
    2. For each polygon $P_i$ in map1;
            a. Compute its MBR $R_i$ by scanning through the vertices of $P_i$ ;
            b. Starting from the root, for each node n of r2 at the same level ;
              for each entry e of the node n with covering rectangle $R_e$;
                if $R_i$ intersects $R_e$;
                    if n is leaf node, record pair ($P_i$, $P_e$), return;
                    if n is not leaf node, go to child node of entry e;
    3. For each pair ($P_i$, $P_e$) from the result above;
            For each chain $C_i$ of $P_i$ ;
              For each chain $C_e$ of $P_e$ ;
                if $C_i$ intersects $C_e$, report polygon intersection, return;
End.
```

Although a polygon identity can be associated with only one of the leaf nodes, its MBR may be contained in the covering rectangles of many nodes, therefore, all the nodes at the same level have to be checked. So only in the best case where the searching MBR intersects with only one node at each level, the $O(log_m n_2)$ performance can be achieved for each MBR.

**Figure 2** below demonstrates the best situation at one level, in which the searching rectangle A intersects only enclosing rectangle 7. The enclosing rectangles on each other levels are disjoint as well. All the searching rectangles are intersecting with only one of the enclosing rectangles on these levels. Since the searching polygonal net has $n_1$ polygons, and exactly one node at each level need to be visited, a total of $O( n_1 \times log_m n_2)$ time is needed at the first step, i.e., *the index join.*

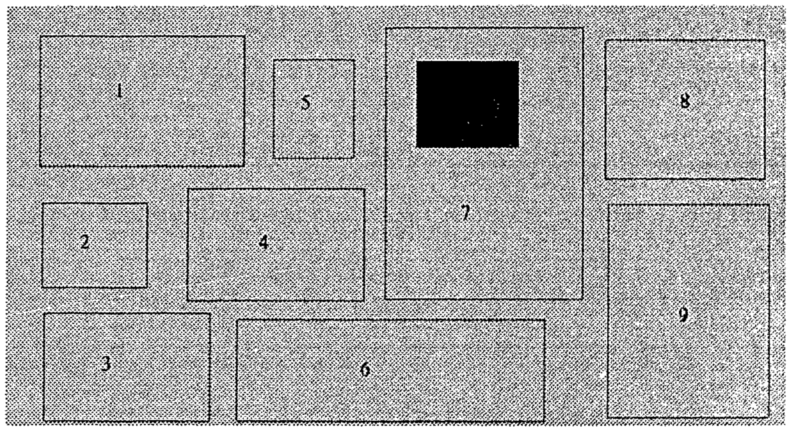

Figure 2 Enclosing rectangles are disjoint and searching MBR intersects only one enclosing rectangle

In the worst case, all the non-leaf nodes have to be visited before ascertaining the final intersecting objects, and the time needed for each processing at the first stage is proportional to the total number of nodes including leaf-nodes in the tree. The search space in this case is *number-of searching-rectangles* × *number-of-total non-leaf-nodes*. Since the leaf-node contains the minimum bounding rectangles of all the polygons in map 2, more than $O(n_1 \times n_2)$ time is needed, where $n_2$ is the number of polygons in the second polygonal net.

This situation can be demonstrated in **Figure 3** below, in which the searching rectangle A is included in all of the enclosing rectangles, so all of them have to be visited on the next level by A. At each level, the sibling enclosing rectangles are overlapping each other severely so that every searching rectangle intersects with every enclosing rectangles on any one of these levels.

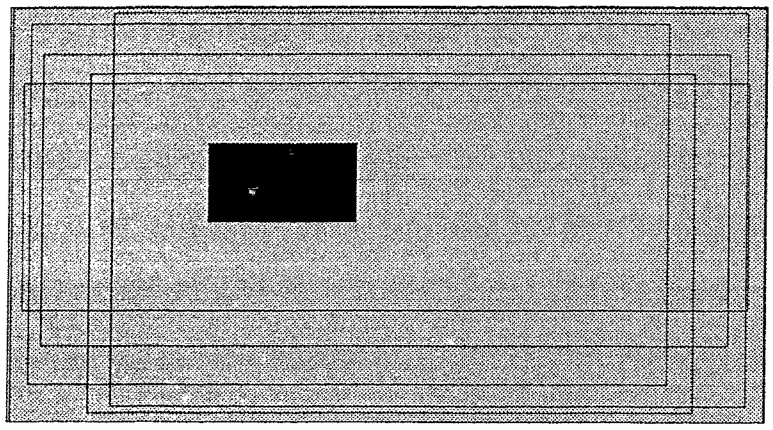

Figure 3 Enclosing rectangles overlap each other and searching MBR intersects all of them

But in the average case, only a limited number of nodes will be visited and compared, because the worst case happens only when the size of the searching MBR is comparable with the space of the intersecting polygonal net, or all the MBRs comprising the R-tree are of similar size and are comparable with the size of the whole map. The former leads to a search of most of the nodes at each level, while the later results in a severely overlapped R-tree no matter what splitting algorithm is adapted when building the tree. Both situations will lead to a search of the majority of the nodes.

In the general case, the map has polygons in uniformed scale. Polygons composing the map are mutually non-overlapping. They could, however, share common boundaries. Each polygon exists as a component of the map being not comparable with the whole map in size. Therefore a large portion of the map will be discarded for each searching MBR during the traversal, and the final performance could be greatly upgraded by the added index join.

### 6.2.3 RPJ: R-tree Parallel Comparison Spatial Join

The *Index Join* does not take into consideration the fact that spatial join is a set operation. From space decomposition point of view, R-tree is nothing more than a hierarchical space occupancy approximation of the polygonal net. The root specifies the bounding space occupied by the map, and each level specifies in detail how the space from the above level is occupied in the form of a set of overlapping rectangles. Down to the leaf level, the space occupancy approximation of each polygon is stored. Therefore, R-trees from two maps representing two space occupancy schemes, can be compared at corresponding approximation level to detect possible set-to-set intersecting pairs.

This leads to the design of the second algorithm for the first step. With this algorithm, the R-trees of both maps are constructed first. Then starting from the root, two trees are compared in parallel manner at each corresponding level. For each node of one tree at certain level, it is compared with all the nodes at the corresponding level from the other tree to decide which node is of interest. By corresponding level, it is the same level counting from the root, not in the topological sense as is the case with Quadtree.

Starting from the root, the rectangle collection representing the space occupancy of the map at each level is compared with that of the other map. This is done by comparing any two of the covering rectangles from both collections. The result of the comparison is recorded in two local stacks which are passed down to the next level for further checking. These local stacks are acting as an approximate snapshots of the sub-space intersections at different levels, and the deeper the level is, the smaller the processing object is, and the more accurate the snapshot is. At the leaf level, the possible intersections of polygon objects are obtained.

Generally the two trees are not of same height. Whichever reaches the leaf level first, the result will be recorded in its local stack and updated while the other tree is traversing down to reach the leaf level.

The algorithm is presented below formally:

```
[algorithm 6.2]
Input: m polygons and m' segments in map 1, n polygons and n' segments
        in map2;
Output: Set of pairs (Pi, Qj), such that Pi belongs to map1 and Qj belongs
        to map2; $0 \leq i \leq m$, $0 \leq j \leq n$;
Begin
  1. Construct R-trees r1 and r2 for map1 and map2 respectively;
  2. Initialize two stacks stack1 and stack2;
       a. Push root node address of r1 into stack1;
       b. Push root node address of r2 into stack2;
  3. While stack1 is not empty and stack2 is not empty;
       a. Initialize two local stacks L_stack1 and L_stack2;
       b. Get node $N_i$ from stack1 and node $N_j$ from stack2;
       c. For each entry $n_i$ in N1 and each entry $n_j$ in N2;
          If the covering rectangles of $n_i$ and $n_j$ intersects;
          Case:
            1). both $n_i$ and $n_j$ are leaves;
                record pair $(P_i , P_j)$;
            2). $n_i$ is leaf and $n_j$ is not leaf;
                push address of $n_i$ into L_stack1;
                push child address of $n_j$ into L_stack2;
            3). $n_j$ is leaf and $n_i$ is not leaf;
                push address of $n_j$ into L_stack1
                push child node address of $n_i$into L_stack2;
            4). both $n_i$ and $n_j$ are not leaves;
                push child node address of $n_i$ into L_stack1
                push child node address of $n_j$ into L_stack2;
        d. Go to step 3 passing down the two local stack;
    4. For each pair ($P_i$, $P_j$) from the result above;
        For each chain $C_i$ of $P_i$ ;
          For each chain $C_j$ of $P_j$ ;
            if $C_i$ intersects $C_j$, report polygon intersection, return;
End.
```

Analysis of this algorithm shows that the configuration of the R-tree itself has significant effect on its final performance. The best performance could be achieved when there are no overlapping of the subspaces partitioned at each level for both R-trees, and each sub-space from one tree on each level overlaps no more than one sub-space from the other tree at the same level. In this very special case, a time proportional to the maximum height of the two trees can be obtained which is extremely fast. In the average case, we can eliminate sets of polygons at each level although more than one node has to be visited and because it operates a set of polygons once at a time, a better performance can be expected over the *index join* based **RIJ**.

However, the worst case complexity is worse because not only does each leaf node of one tree have to be compared with each of the other tree, but each interior node of one tree has to be compared with that of the other as well. If there are $n_1$' non-leaf nodes in r1 and $n_2$' non-leaf nodes in r2, the time needed in this case will be proportional to the total number of nodes including the leaf nodes from both trees, which is $O[(n_1+n_1')\times(n_2+n_2')]$. This is worse than $O[n_1 \times n_2]$. Fortunately this is an extreme case in which all the objects in the map is taking up the entire map space, i.e., overlapping each other on the entire map.

Notable is the fact that the object overlap at each sibling level affects the space partitioning at each level, and therefore affects the search complexity.

### 6.2.4 Experimental Results

Both RIJ and **RPJ** were implemented and tested over the same 6 sets of polygonal data used in chapter 5. Since *Index Join* and *Parallel Join* are the major difference of the two algorithms used as their first step respectively, we tested the performance separately, so that we could compare the *Index Join* and *Parallel Join* as well.

Firstly, the graphed results in **Figure 4** below reflects the improvement of the *parallel join* over the *index join*. They are the first steps of the algorithms **RIJ** and **RPJ** respectively.

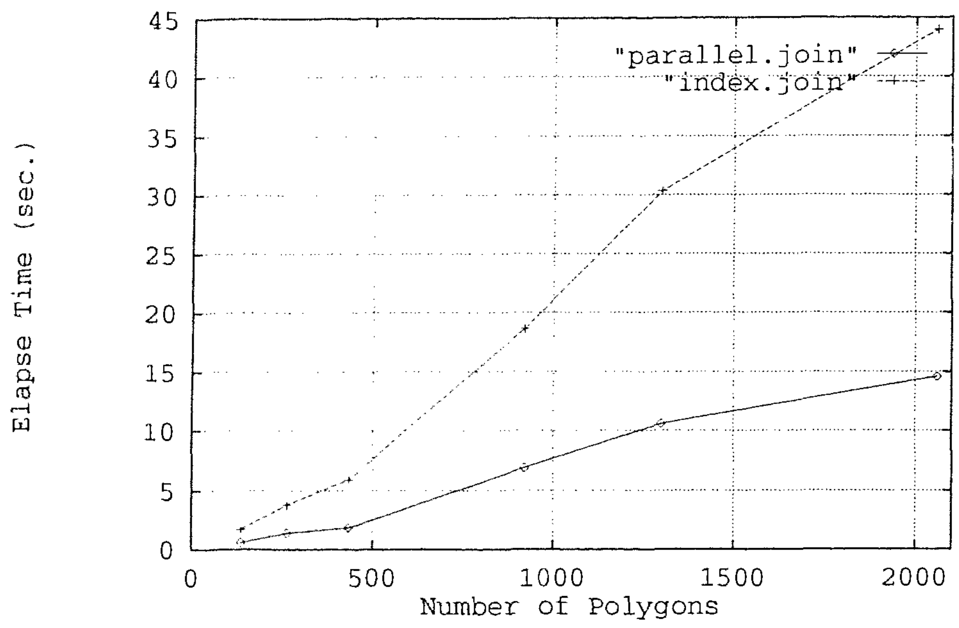

Figure 4 Comparison of **RIJ** and **RPJ** at their first step

On average, there is a 66% decrease in the time needed to perform the *parallel join* over the *index join*. As an example, with data set 6 having a size of 916×1144, *parallel join* takes only 14.54 secs which is 33.06% of the 43.99 secs by the *index join*. It is shown from the figure that the *parallel join* takes approximately only one third of the time needed by the index join, a significant improvement, which verifies our analysis of the two joins in the above sections.

Since R-tree is not involved in the second step, which generates the final intersecting pairs based on the approximate result from the first step, the time needed at this stage is not different for the two methods. When considered as a whole, **Table 1** below shows the total time needed for **RIJ** and **RPJ** respectively over the six sets of data.

| *Elapse Time (sec.)* | *DateSet1* | *DataSet2* | *DaatSet3* | *DataSet4* | *DataSet5* | *DataSet6* |
|---|---|---|---|---|---|---|
| **RIJ** | 38.3617 | 65.9131 | 118.135 | 263.813 | 454.038 | 1205.16 |
| **RPJ** | 36.5430 | 62.5122 | 113.9839 | 245.5521 | 415.738 | 1060.004 |
| **% improved** | 4.7401 | 5.1597 | 3.5139 | 6.9219 | 8.4354 | 12.045 |

Table 1 Comparison of **RIJ** and **RPJ** totally

The improvement of **RPJ** over **RIJ** is not however overwhelming, as shown above, because the final intersection checking is still the most time consuming operation of all. In our measurement, the final intersection checking takes about 95% of the total spatial join time. We use R-tree to perform rectangle join first, in the light of reducing the redundant checking involved in this operation. As an extra data structure to assist the computation, R-tree should never take a large part of the whole operation, which best serves as a topological data structure to filter out those impossible pairs. This can not be achieved by the vector data model without utilizing indexing structure because of its lack of global topology.

On the whole, both **RPJ** and **RIJ** should dramatically improve over the **Boundary-Join**, which uses only local data structure. This will be further discussed in Chapter 8, in which we will use **RPJ** as the representative spatial join algorithm on R-tree, and we name it **R-Join** to distinguish it from other join algorithms.

## Section 6.3 Measurement on Node Size

The experiment above was conducted under a certain *node size* 15, i.e., a maximum 15 entries allowed for each node. According to the analysis we had in section 6.2.2, *node size* affects the performance critically in both best and worst situations. To estimate the effect that different node sizes generate in the average case with the algorithms, we will have several experiments followed by discussions in this section.

### 6.3.1 Experiments

In order to study the behavior under various node sizes, a wide range of node sizes is tested: the sizes include 3, 5, 10, 15, 20, 30, 40, 50, and 60. The algorithm we chose to experiment on is **RIJ**. Performance data were obtained from different stages of the algorithm: the construction stage, the *index join* stage, and final stage, in order to see the effects from various aspects. But the same 6 sets of testing data were used. With each data set, the measurement is always taken by choosing the larger map as the R-tree index and the smaller map as the searching objects.

**Node Size Affects R-tree Construction** The time needed to build the tree is measured first under various node size chosen. **Figure 5** below graphs the testing results.

The graph reflects firstly, that the time needed to build the tree is proportional to the number of objects in the data set obviously, with the curve of data set1 at the bottom and the curve of data set6 above that of all the rest of the data sets.

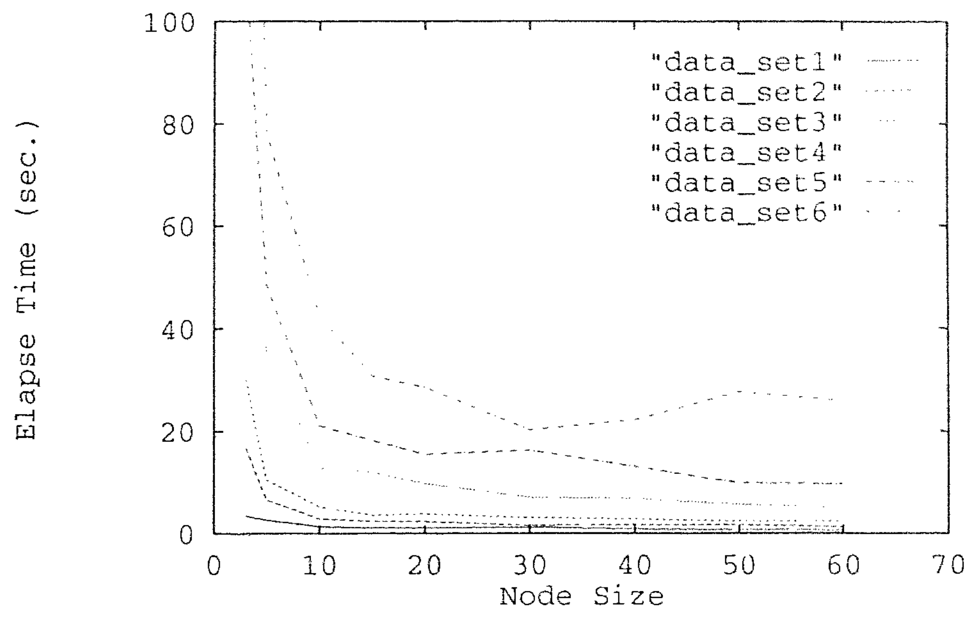

Figure 5 Node Size affects the tree construction time

Secondly, for each data set, the time needed to build the tree tends to decrease as the node size grows larger. For example, for the fourth data set in the table having map set size 382 × 538, when maximum only 3 entries are allowed for each node, 65.73 secs which is more than one minute is necessary to build the tree for 538 objects; while when the maximum entries is chosen as 50, only 5.62 secs is necessary. The larger the node size, the less time spent.

Thirdly, the decrease curve over the chosen 6 node size is not an even one. Prominent drops occur when the node size is less than 10. The time remains low when more than 10 entries are allowed, and change is not obvious thereafter.

**Node Size Affects Performance of Index Join** It can be concluded that having a large node size, in our case more than 10 entries per node, will generate a tree quickly, and therefore a shallow one. However, the choice also depends on the performance of *index join*.

When *index join* is performed on the trees varying on the node size, **Figure 6** graph below reflects that the *index join* on the shallow trees can be performed indeed faster than on the deep trees with node size less than 10. For example, with data set 4, 5.83 secs is needed to do an *index join* on tree with maximum 30 entries per node, while 15.48 secs is needed with maximum only 3 entries per node.

However, when the node size is chosen more than 20, the difference tends to diminish as the node size grows larger. Therefore, with a data set having under 1000 objects, the node size chosen should be least 10, and within the range of 10 to 60, join can be performed with no major time difference.

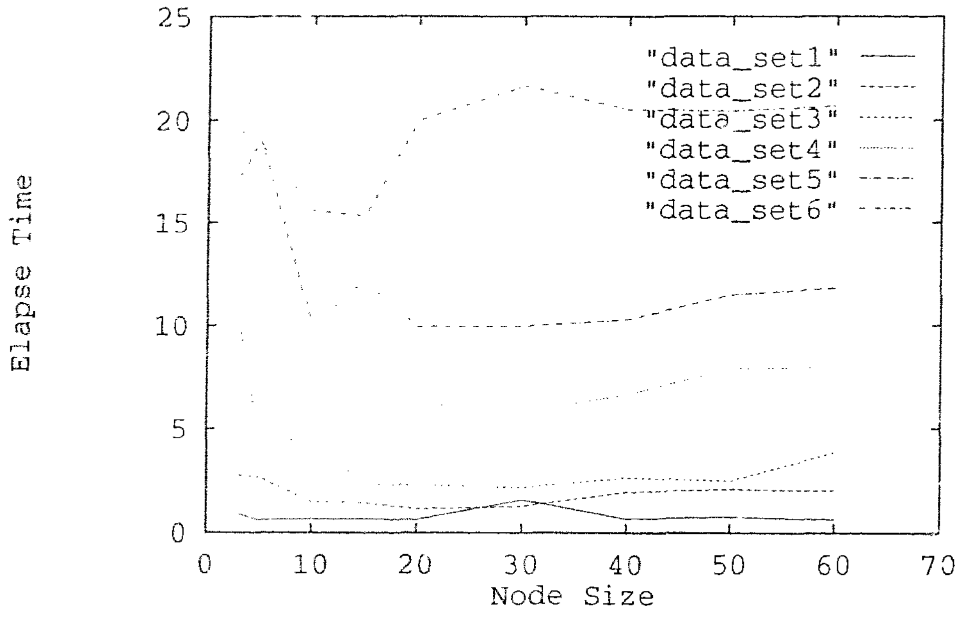

Figure 6 Node Size affects the performance of *Index Join*

Notice that data set6 behaves slightly differently. Although the join time decreases as the node size increases from 3 to 20, further increase of the node size makes the join time go up. The join time needed remains higher after 30 than that of node size under 20 but above 10.

**Node Size Affects the Performance of RIJ** When the two steps are considered together, i.e., measuring the time needed from the beginning of building the tree to the end of performing the final join, **Figure 7** below graphically presents the result obtained.

**Figure 7** shows that each data set has its best performance under node size 20 or 30. Performance doesn't change much thereafter. So generally, a node size of more than 20 will generate a fairly good result, considering also the map size when the final choice is made. With maps having more than 1000 objects, the testing range of *node size* should be expended, and the best node size could vary as well.

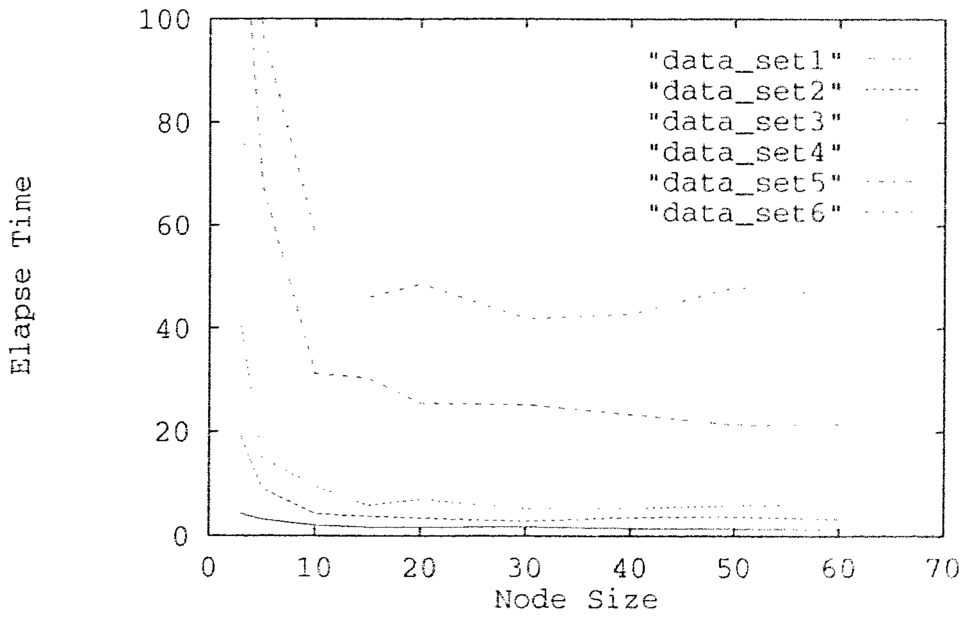

Figure 7 Node Size affects the performance of RIJ

### 6.3.2 Discussions

*Node size* is an optimization parameter for our methods. A large node size means that the tree is shallow; a small node size means that the tree is deep with respect to the same map data. If overlapping at each level is not serious, searching sets of objects will go quickly on a shallow tree, and building the tree needs less time.

Considering all the aspects, *node size* should be chosen experimentally in the average overlapping situation. It is not simply that the larger the node size, the better the performance. Our experiments on data sets under Gaussian distribution shows similar results. Both data sets show obviously better performance with *node size* of more than 10, and the performance afterwards varies little, but does not keep dropping down in certain range.

# Chapter 7
# Grid Coordinate System (GCS) and Spatial Join

In our approach of spatial join processing, the main task of a spatial data structure is to represent space in a way that would assist the object overlapping detection. Both PM Quadtree and R-tree feature hierarchical representation, input data dependent, and complicated computation. In this chapter, we will introduce for the spatial join a Gird Coordinate System, which is independent of the objects that populate the space and not requiring a substantial amount of space partition calculation to achieve the same purpose.

## Section 7.1 CSJ: Cell Spatial Join

In this section, we present a spatial join algorithm **CSJ** based on the Grid Coordinate System (GCS). General strategy and its explanation is provided first, followed by the detailed description of the algorithm and the analysis of it.

### 7.1.1 General Processing Strategy of CSJ

Grid Coordinate System provides a way of uniformly partitioning the space. Spatial objects in the space are therefore divided into groups according to their position in the space. The grouping of objects reduces computation by performing the operation on the objects belonging to the same group.

The strategy of spatial join processing is also a two-step processing: griding step and checking step. Griding step serves as a filter as PM Quadtree and R-tree, to provide possible objectives; only objects in the same grid cell are checked for intersection. GCS combines "buffering technique" to convert objects in the griding step by object approximation in stead of object itself .

The general processing can be stated as following:

[Step 1] *Superimpose a Grid Coordinate System on one polygonal net, convert it into grid representation by computing cell occupancy of objects according to their approximations; This called Net-Conversion*

[Step 2] *For each object in second polygonal net, compute its cell occupancy, and check for intersections with all the objects in the computed cells.*

The processing can take advantage of PM Quadtree's clipping and R-tree's object bounding rectangle representation. Objects are reallocated by the superimposing, and the calculation is not so expensive because of both the object approximation, and non-hierarchical representation. The computational cost in the second step is directly related to the number of objects in each cell, or the number of (*cell index, object index*) pair (refer to Chapter 4 for the concept), and the object chosen as the processing unit.

Object approximation depends on the object chosen which could be polygon, chain or segment. Cell occupancy varies with the object approximation, and so does the algorithm. Deciding an object approximation and calculating the cell occupancy play an important role in the algorithm performance. We will discuss them first before we proceed to the detailed algorithm.

### 7.1.2 Object Approximation and Net-Conversion

When superimposed by the flat grid, objects in the polygonal net are converted to their grid representation by clipping the objects on the grid according to their grid cell occupancy. This is called *Net-Conversion.*

Unlike in PM Quadtree where each segment clips on squares successively, the clipping in GCS is done by object approximation to reduce the net conversion time.

A minimum bounding rectangle with its sides parallel to the axis is again chosen as the approximation form. It is not only simple to find the bounding rectangle of the object, but also very convenient to calculate the cell occupancy. Bounding rectangles can be found by traversing the object once, and cell occupancy can be computed without having to calculate the segment intersections and test the ranges of the intersecting points as we do with PM Quadtree.

**Calculating Cell Occupancy of Object Approximation** **Figure 1** below illustrates how grid cells are calculated according to the object bounding rectangle's occupancy.

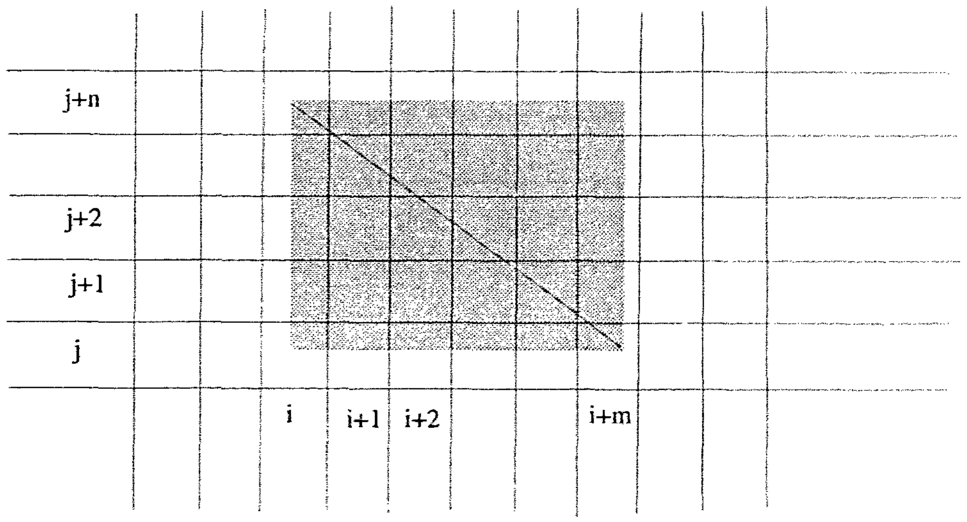

Figure 1 Calculating Grid Cell Index By the Object Bounding Rectangle

In GCS, each grid cell has a unique number decided by its row and column number. Suppose the uniform grid is composed of G×G cells, a cell dwells in i row and j column has its cell number calculated as:

$$N_{ij} = F(i, j) = j + G \times i, \; 0 \leq i, j \leq G-1;$$

For any arbitrary object $O_i$, suppose its bounding rectangle is $R_i$, which can be represented by a quad tuple ($X_{min}$, $Y_{min}$, $X_{max}$ , $Y_{max}$), and $0 \leq X, Y \leq 1$. its occupying cell indices can be computed by the cross product of the occupying row numbers and column numbers.

Suppose the occupying row numbers are i, i+1, i+2, ......, i+m and column numbers are j, j+1, j+2, ......, j+n, since the occupying row numbers or column numbers are always consecutive, these two sequences can be calculated as:

$i = [X_{min} \div (1 / G)] = [X_{min} \times G]$ ,

$i+m = [X_{max} \div (1 / G)] = [X_{max} \times G]$ ; and

$j = [Y_{min} \div (1 / G)] = [Y_{min} \times G]$ ,

$j+n = [Y_{max} \div (1 / G)] = [Y_{max} \times G]$ ;

Therefore the cell index set $\Gamma$ can be computed as:

$\Gamma = \{N_{p,q} | i \le p \le i+m, j \le q \le j+n\} = \{F(p,q) | i \le p \le i+m, j \le q \le j+n\}$ ;

where $\{(p, q) | i \le p \le (i+m); j \le q \le (j+n)\} = (i, i+1, \ldots\ldots, i+m) \times (j, j+1, \ldots\ldots, j+n)$,

Although the object bounding rectangle reduces the amount of computation, it inevitably introduces the inaccuracy. The precise calculation of the object cell occupancy will result in less number of cell indices, i.e. less redundancy, and therefore will result in less amount of further computation. But it is cost to clip the object on the grid to obtain exact cell occupancy.

As we use griding only as a filter step for the actual intersection checking, it should take least time to serve its role of generating possible candidates. Reasonable amount of redundancy is tolerable as the cost of reducing tedious intersection checking. By choosing an appropriate approximation object and processing unit, the redundancy can be reduced, so is the amount of processing to achieve a good performance.

**Segment approximation and Chain-based Processing** By choosing polygon as the approximation unit, we have the advantage of smallest number of processing units compared with using segment or chain as the approximation unit, but also have the disadvantage of computation overhead and needs of calculating the bounding rectangle.

The computation overhead is not so severe with R-tree based spatial join when we chose polygon as the processing unit. This is because R-tree generates better candidate sets, i.e., a more accurate candidate set, while with *Net-Conversion*, the coarse calculation of the cell occupancy results in a much larger candidate set. For example in the **Figure 2** below, polygon $\beta$ occupies 15 cells while its bounding rectangle occupies 30 cells, which is 100% redundancy.

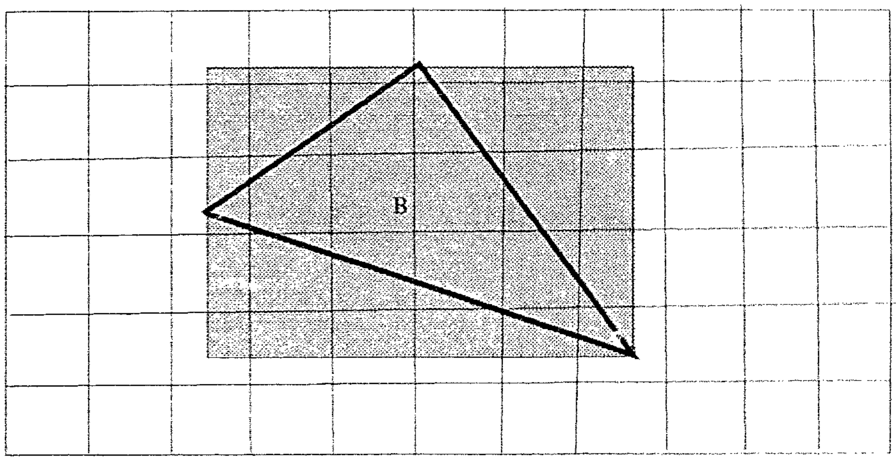

Figure 2 polygon $\beta$ occupies 15 cells, its bounding rectangle occupies 30

Considering the further checking based on this intermediate result, this polygon has to be checked with all the polygons in the false cells, and common boundary chains are processed twice causing more redundancy. The overhead can not be ignored.

Computation overhead caused by the conversion inaccuracy exists as well with chain or segment being the processing unit. With segment, we have the benefit of obtaining the bounding rectangle directly from the coordinates of the ending points, but the number of (*cell index, segment index*) pairs would be greatly increased. This is based on the definition that a polygon is a closed sequence of chains, and each chain is composed of sequence segments. Therefore in general, number of segments is much larger than number of polygons. As the objective is polygon intersection detection, there is inevitable computation overhead.

Generally, the number of chains is comparable with the number of polygons. As the processing unit, it is better than polygon with respect to the repetitive checking of polygon common boundary. Not only is the common boundary checked only once, but repetitive checking of false cells brought by the inaccurate occupancy of the boundary is eliminated as well. It is also better than using segment as the processing unit, because the number of (*cell index, chain index*) pair is much less, so is the total amount of computation, and more topology is attached. Therefore, chain is chosen as the processing unit in our algorithm. The cell occupancy of chain is calculated segment by segment, so to avoid calculating bounding rectangles first. Chain index is preserved in the cell index array after the computation by segment, and repetitive chain index is not stored. Thus we achieve small number of (*grid index, chain index*) pairs to reduce the computation overhead. Furthermore, each chain is processed once and only once. The polygon information attached to the chain makes it efficient to collect polygon intersections out of the chain's.

### 7.1.3 CSJ: Cell Spatial Join

The spatial join starts with applying *Net-Conversion* algorithm on each polygon chain of one of the two maps. During the conversion, the bounding rectangle of each segment of the chain is superimposed on the grid, and indices of the intersecting grid cells are computed. Pairs of (*cell index, chain index*) are recorded in the index array in which chain indices are linked together as a list and referenced by *cell index* which is the array script. No repetitive chain index is stored in the list so to avoid redundant computation. The data structure can be pictured in **Figure 3** below.

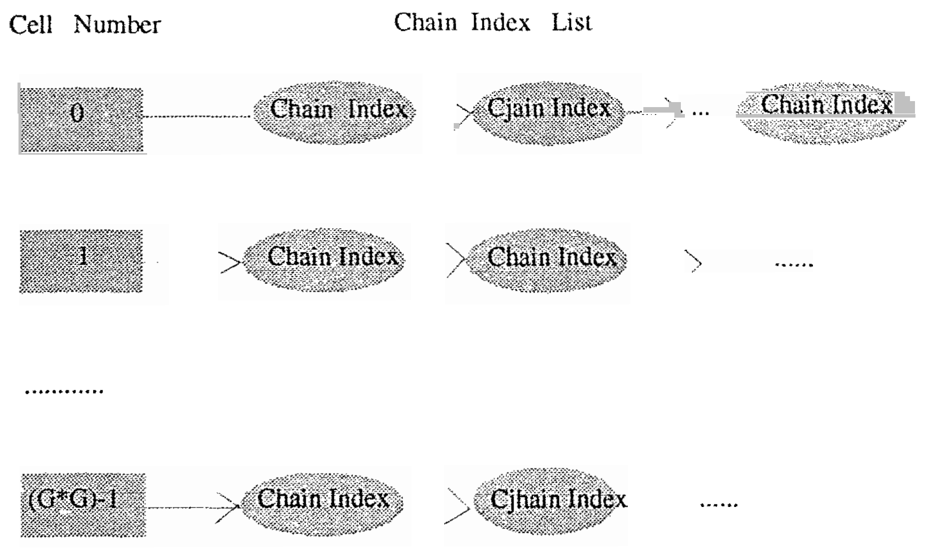

Figure 3 Data structure of Cell Index Array

After the net conversion on the first map, chains of the map are divided into groups according to their occupancy on the 2–d space, and stored in the grid index array. Applying the *net-conversion* on each polygon chain of the second map generates its occupying grid cell indices stored in a cell list as well. Referenced by the indices computed, those chain lists in the index array from the first map are possible candidates for actual intersection checking. Chain by chain intersection checking is therefore performed between this chain and each of the chains in the lists, and once there is an intersection, the intersections of left and right polygons of the two chains are recorded in a matrix recording the polygon intersections.

The algorithm is presented below:

```
[algorithm 7.1]
Input: m polygons and m' segments in map 1, n polygons and n' segments
        in map2;
Output: Set of pairs (Pi, Qj), such that Pi belongs to map1 and Qj belongs
        to map2; $0 \le i \le m$, $0 \le j \le n$;
Begin
1. Apply Net-Conversion algo:ithm on the first polygonal net Map 1;
    1a. Initialize cell index array A;
    1b. For each polygon chain $C_{1i}$ of Map 1;
      For each segment $S_{1j}$ of $C_{1i}$ ;
        a) Get its enclosing rectangle $SR_{1j}$;
        b) Compute indices x of grid cells intersecting with $SR_{1j}$ ;
        c) Add $C_{1i}$ to the lists pointed by A[x], if $C_{1i}$ is not in the lists;
2. Initialize result matrix M, set M [i, j] to bit 0;
3. For each polygon chain $C_{2i}$ in Map2 belonging to polygon $P_{2l}$ and $P_{2r}$ ;
      3a. Clear cell list CList;
      3b. For each segment $S_{2j}$ of $C_{2i}$ ,
        a) Find its enclosing rectangle $SR_{2j}$ ;
        b) Compute indices y of grid cells intersecting with $SR_{2j}$ ;
        c). Record indices y in CList, if y is not in CList;
      3c. For each chain $C_{1k}$ in the list pointed by A[y];
        If $C_{1k}$ and $C_{2i}$ intersects, and $C_{1k}$ belongs to polygon $P_{1l}$ and $P_{1r}$
            Set M [$P_{1l}$, $P_{2l}$], M [$P_{1l}$, $P_{2r}$], M [$P_{1r}$, $P_{2l}$], M [$P_{1r}$, $P_{2r}$] to bit 1;
4. For each M [i, j] which is set to bit 1, report intersection of polygon i and j.
End.
```

We will analysis the algorithm generally in the following sub section.

## 7.1.4 Analysis of the CSJ

Let $E(C_1)$ be the expected number of chains of map 1 passing through one grid cell, and $E(C_2)$ be the expected number of chains of map2 passing through one grid cell. Given a grid having $G \times G$ number of composing cells, the computational cost for this algorithm is $O[G \times E(C_1) \times E(C_2)]$. $E(C_1)$ and $E(C_2)$ are the actual expected *chain grid volum.*

The expected number of chains passing through one grid cell is directly proportional to the expected number of segments passing through each cell, or expected *segment grid volume*, according to the algorithm. Let $E(S_1)$, $E(S_2)$ be the expected *segment grid volume* in map1 and map2 respectively. $E(S)$ is composed of expected number of segments completely contained in the cell $E_1(S)$ and expected number of segments crossing the cells $E_2(S)$.

So we have

$E(S_1) = E_1(S_1) + E_2(S_1)$ ; $E(S_2) = E_1(S_2) + E_2(S_2)$;

These expected values are closely related to the underlying data distribution, or in other words, the performance of the algorithm is input data dependent. Without the statistical knowledge of the polygonal net, calculating the expected value about segment is very difficult. With our randomly generated net, although the points were controlled under certain distribution, the polygonal net resulted from the line intersections present an unknown distribution.

However, whatever the data distribution is, the *grid resolution* affects the performance significantly along with the number of segments and chains in the net. Generally, more number of segments or chains results in larger $E(S)$ with a given *grid resolution*. With specific data set, as the *grid resolution* increases, i.e., $1/G \rightarrow 0$, $E(S)$ decreases, so is the amount of computations. This explains why the cost can be cut down greatly after applying uniform griding schema. But G can not be increased forever.

When *grid resolution* grows very large such that the *grid size* is too small compared with the average *segment length*. Number of segments intersecting the grid cells will be very large, i.e.. $E_2(S) \rightarrow$ infinite, when $1/G \rightarrow 0$; In this case, *Net-Conversion* will cost more than the intersection checking in each cell, even though $E_1(S)$ is very small, and amount of computation units increases unbearably with the high resolutions, which results in too many *(cell index, chain index)* pairs.

Therefore, *grid resolution* should be bounded by the average *segment length* to obtain a reasonable *segment grid volume*.

Since for those segments belonging to same polygon chain, only one chain index is recorded, *chain volume* is generally smaller than *segment grid volume*, this is what we do to reduce the computation units of further checking. But the inaccuracy of segment cell occupancy introduces redundancy, hence makes the *chain grid volume* larger than it actually is.

There is no neat mathematical solution. But it can be concluded from the above analysis, that firstly, a *grid size* that is comparable with average *segment length* should yield a good enough performance, i.e., the average *segment length* should be about the size of the grid, or larger but not too much which will result in heavily fragmented segments, high *chain grid volume*, and too many (*cell index, chain index*) pairs. Furthermore, with the comparable sizes, the best performance should be able to obtained under one certain grid resolution which generates a low *chain volume* for certain data set.

Section 2 presents results from empirical tests that verifies the above analysis.

## Section 7.2 Empirical Analysis

Experiments were conducted to find out how *grid resolution, chain grid volume, (cell index, chain index)* pair are related to each other, and how they relate to the algorithm performance. We used same six sets of polygonal data from Chapter 5 and 6. Data with Gaussian distribution of the ending points was also tested for certain performance data.

In this section, we first list the statistics of the testing polygonal data in order to be used in the later comparison. Then we present the study of optimal *grid resolution* for each of the data set. Related facts under the optimal *grid resolution* are listed to verify our analysis in the above section.

### 7.2.1 Statistics of Different GCSs and Testing Data

In the preliminary Chapter, we have stated that the polygonal net is generated in a 1×1 unit. So the GCS is also within the unit. If *grid resolution* is set as G×G, G≥1, then the *grid size* can be computed as 1/G, which is less or equal to 1. Since segments come from the intersecting of straight lines within the unit, we also have 0>*segment length*<1.

| | *Number of Polygons* | *Number of Edges* | *Average Segment length* |
|---|---|---|---|
| **DataSet 1** | 38 × 100 | 945 × 2169 | 0.145 |
| **DataSet 2** | 100 × 165 | 2169 × 3726 | 0.096 |
| **DataSet 3** | 165 × 269 | 3726 × 5724 | 0.078 |
| **DataSet 4** | 382 × 538 | 8388 × 11763 | 0.057 |
| **DataSet 5** | 538 × 761 | 11763 × 16722 | 0.048 |
| **DataSet 6** | 916 × 1144 | 19458 × 24615 | 0.036 |

Table 1 Statistics of each testing data set

Each set of data is composed of two po'vgonal nets, and each net has its own statistics, therefore functions differently on the same GCS. The overall performance of **CSJ** is decided by the average statistics of the two nets. **Table 1** above presents the statistics of the 6 data sets, and *segment length* is based on the average result.

To be comparable with the average *segment length*, the resolution of the GCS we have chosen varies from 21×21 up to 91×91. The statistics of these GCSs is listed in **Table 2**. Notise that the *grid size* ranges from 0.011 to 0.0476. With DataSet 1, having the maximum average *segment length*, it indicates a range of 3 to 10 times of *segment length* to the *grid size*, while with DataSet6, it indicates a range of 1 to 3 times.

| *Grid Res-olu -tion* | *20* | *30* | *40* | *50* | *60* | *70* | *80* | *90* |
|---|---|---|---|---|---|---|---|---|
| *Grid Size* | 0.0476 | 0.0323 | 0.0244 | 0.0196 | 0.0164 | 0.0141 | 0.0124 | 0.0110 |

Table 2 Statistics of different GCSs

### 7.2.2 Performance of CSJ Under Different Grid Resolution

**Figure 4** below presents the performance curve of each data set on the 8 GCSs. The result shows that each set of data has its best performance under certain *grid resolution* which we call *optimal resolution*. For example, data set 1 has its *optimal resolution* 40×40, while data set 6 achieves its best performance with *optimal resolution* 70×70. For whichever data set, the performance declines on both side of the *optimal resolution*, and the difference presented is not minor. For example, with data set 6, nearly 520 seconds is needed to do the **CSJ** when the resolution is 90×90, while only 273.681 seconds is needed if we perform **CSJ** under the *optimal resolution* of data set 6. This verifies our conclusion, that time needed to perform the join decreases as the *grid resolution* increases from 1, but further increase of resolution causes the drop of the performance. The best performance is achieved under one certain resolution. In the next sub section, we will study the related facts under *optimal grid resolution* in order to be able to decide it beforehand.

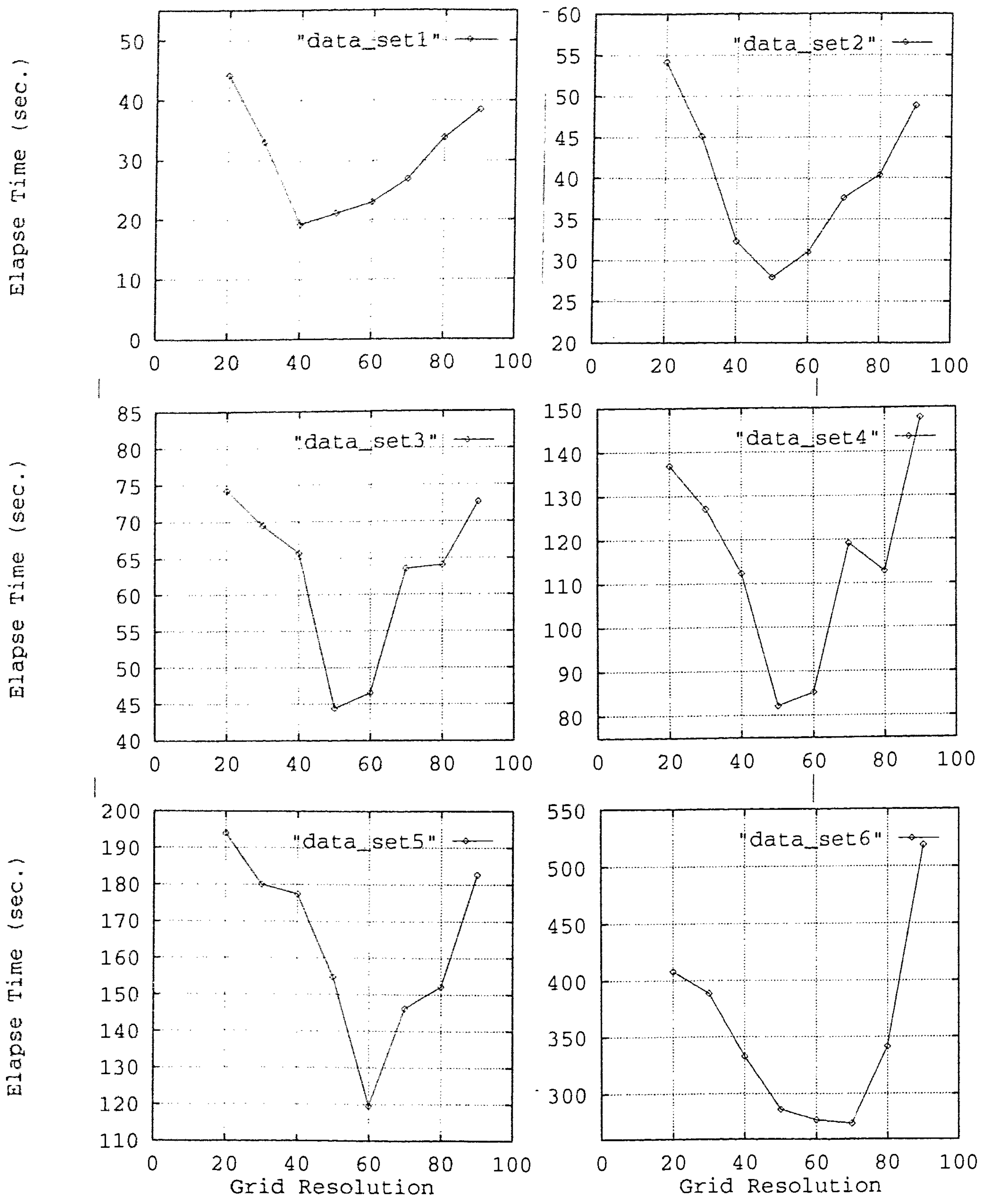

Figure 4 Performance of **CSJ** under different *grid resolution*

### 7.2.3 Chain Grid Volume and Number of (cell index, chain index) Pairs Under Optimal Resolution

To find out what is related to the *optimal resolution*, for each data set, we also tested their average *chain grid volume* and number of (*cell index, chain index*) pairs along with the total grid number under each grid resolution. The results are listed in the **Table 3** to **Table 8** below. Each figure tables the data of one data set.

| | *Grid Resolution−1* | *Number of Grid Cell* | *Number of Pair (cell index, chain index)* | *Average chain grid volum* |
|---|---|---|---|---|
| **DataSet 1** | 20 | 441 | 808.5 | 1.83 |
| | 30 | 961 | 1215 | 1.26 |
| | 40* | 1681* | 1723* | 1.03* |
| | 50 | 2601 | 2271 | 0.87 |
| | 60 | 3721 | 2850 | 0.77 |
| | 70 | 5041 | 3546 | 0.70 |
| | 80 | 6561 | 4245 | 0.64 |
| | 90 | 8281 | 5021 | 0.61 |

Table 3 Statistics under various *grid resolution* for data set 1

| | *Grid Resolution−1* | *Number of Grid Cell* | *Number of Pair (cell index, chain index)* | *Average chain grid volum* |
|---|---|---|---|---|
| **DataSet 2** | 20 | 441 | 1131 | 2.56 |
| | 30 | 961 | 1629 | 1.70 |
| | 40 | 1681 | 2189 | 1.30 |
| | 50* | 2601* | 2810* | 1.08* |
| | 60 | 3721 | 3400 | 0.91 |
| | 70 | 5041 | 4168 | 0.83 |
| | 80 | 6561 | 4854 | 0.74 |
| | 90 | 8281 | 5682 | 0.69 |

Table 4 Statistics under various *grid resolution* for data set 2

|  | *Grid resolution−1* | *Number of Grid Cell* | *Number of Pair (cell index, chain index)* | *Average chain grid volum* |
|---|---|---|---|---|
| **DataSet 3** | 20 | 441 | 1570 | 3.56 |
|  | 30 | 961 | 2218 | 2.31 |
|  | 40 | 1681 | 2888 | 1.72 |
|  | 50* | 2601* | 3650* | 1.40* |
|  | 60 | 3721 | 4446 | 1.20 |
|  | 70 | 5041 | 5325 | 1.06 |
|  | 80 | 6561 | 6200 | 0.95 |
|  | 90 | 8281 | 7271 | 0.87 |

Table 5 Statistics under various *grid resolution* for data set 3

|  | *Grid Resolution−1* | *Number of Grid Cell* | *Number of Pair (cell index, chain index)* | *Average chain grid volum* |
|---|---|---|---|---|
| **DataSet 4** | 20 | 441 | 2494 | 5.66 |
|  | 30 | 961 | 3306 | 3.44 |
|  | 40 | 1681 | 4122 | 2.45 |
|  | 50* | 2601* | 5055* | 1.94* |
|  | 60 | 3721 | 6023 | 1.62 |
|  | 70 | 5041 | 7031 | 1.39 |
|  | 80 | 6561 | 8034 | 1.22 |
|  | 90 | 8281 | 9174 | 1.11 |

Table 6 Statistics under various *grid resolution* for data set 4

| | *Grid Resolution−1* | *Number of Grid Cell* | *Number of pair (cell index, chain index)* | *Average chain grid volum* |
|---|---|---|---|---|
| **DataSet 5** | 20 | 441 | 3227 | 7.32 |
| | 30 | 961 | 4184 | 4.35 |
| | 40 | 1681 | 5157 | 3.07 |
| | 50 | 2601 | 6287 | 2.42 |
| | 60* | 3721* | 7394* | 1.99* |
| | 70 | 5041 | 8614 | 1.71 |
| | 80 | 6561 | 9776 | 1.49 |
| | 90 | 8281 | 11060 | 1.34 |

Table 7 Statistics under various *grid resolution* for data set 5

| | *Grid Resolution−1* | *Number of Grid Cell* | *Number of Pair (cell index, chain index)* | *Average chain grid volum* |
|---|---|---|---|---|
| **DataSet 6** | 20 | 441 | 4459 | 10.11 |
| | 30 | 961 | 5608 | 5.84 |
| | 40 | 1681 | 6769 | 4.03 |
| | 50 | 2601 | 8069 | 3.01 |
| | 60 | 3721 | 9279 | 2.49 |
| | 70* | 5041* | 10132* | 2.01* |
| | 80 | 6561 | 12115 | 1.85 |
| | 90 | 8281 | 13532 | 1.63 |

Table 8 Statistics under various *grid resolution* for data set 6

Those lines marked with * are data under *optimal resolution.* The data shows that the *optimal resolution* grows from $41 \times 41$ up to $71 \times 71$ along with the growth of the data set, and the average *chain grid volume* remains at the rate of 1 or 2 , i.e. the number of ( *cell index, chain index* ) pairs is approximately 1 to 2 times of the total grid cell number under the *optimal grid resolution.*

**Table 9** below also presented the comparison between the average *segment length* and the *grid size* when the best performance is achieved.

| | *DataSet 1* | *DataSet 2* | *DataSet 3* | *DataSet 4* | *DataSet 5* | *DataSet 6* |
|---|---|---|---|---|---|---|
| *grid size* | 0.024 | 0.02 | 0.02 | 0.02 | 0.017 | 0.014 |
| **Average** *segment length* | 0.145 | 0.096 | 0.078 | 0.0565 | 0.0475 | 0.036 |
| **Ratio** | 6.05 | 4.8 | 3.9 | 2.85 | 2.79 | 2.57 |

Table 9 Comparison of *grid size* and average *segment length* under *optimal grid resolution*

The data in the **Table 9** shows that as the data set grows larger, *grid size* needs to be decreased to achieve the best performance, i.e., *grid resolution* needs to be increased to adapt to the high data density, and the ratio of the average segment length to the grid size ranges from 2.57 to 6.05, which reflects the comparable sizes of the two, i.e., the *segment length* should not be overly large compared with the *grid size* when the best performance is to be achieved.

# Chapter 8
# Comparison of Spatial Joins and Their Underlying Spatial Indexing Methods (SIMs)

In this chapter, we will use **Boundary-Join** as the base line performance to demonstrate the point of introducing SIM for the design and implementation of spatial join. Next, we compare these index-dependent algorithms among themselves, in the mean time, present comparative studies of the underlying SIMs. We will show how GCS functions with respect to spatial join operation.

## Section 8.1 Optimize Spatial Join by PM Quadtree, R-tree and Grid Coordinate System (GCS)

Spatial join deals with complex objects, in our case, 2–d simple polygons. Object representation and object accessing space affect the way of the spatial join processing. In Chapter 5,6, and 7, we have introduced additional spatial data structures to assist the processing, especially in the aspect of object accessing space. It is aimed at making use of the object spatial occupancy in the object space, and saving the high cost of actual join by processing the object approximation to generate candidate set for further processing, or applying recursive data partitioning such that objects are accessed in a more efficient way for the spatial join.

In this section, we will demonstrate the efficiency of our three spatial join processing strategies represented by algorithms **PM-Join, R-Join, and Grid-Join** as explained from the previous Chapters by comparing each of them with the non-optimized method: **Boundary-Join**. The comparison will be based on both analysis and empirical results.

## 8.1.1 Optimize Object Accessing Space by PM Quadtree

For PM Quadtree, space is regularly and recursively decomposed into quadrants until a very fine separation of segments is obtained. It yields an exact representation of collection of segments, not an approximation. Each segment has its specific position in the reorganized tree space and any segment intersecting or neighboring will be led to this segment by the tree within limited number of steps and with certain amount of calculations. The exhaustive search is therefore avoided.

PM Quadtree is an example of reorganizing the segments by regularly dividing the object space and repositioning segments according to its relationship to the square units of the decomposed space.

Empirical results was obtained on the performance of both **Boundary-Join** and **PM-Join** over 2 groups of data. Each group has six sets of polygonal data and similar data sizes for each data set, but with different distributions. The first group of data was generated by having a uniform distribution of the ending points composing the polygonal lines; while the second group having a Caussion distribution. Results are shown in **Figure 1** and **Figure 2** respectively.

In both figures, **PM-Join** keeps its performance fairly low under 1000secs, while **Boundary-Join** grows up vastly along with the growth of data size. However, performance of **PM-Join** on normally generated data appeared a little worse than that on uniformly generated data with respect to similar but large data set, while with **Boundary-Join** they vary little. For example, with data set 6, 1644.43 secs is needed on data size $916 \times 1044$ in group 1, and 1980.54 secs needed on size $907 \times 1123$ in group 2. This is because PM Quadtree is sensitive to input data, and different data distribution produces different decomposition schema, hence different quadtrees. Data resulted from Gaussian distribution has a more uneven distribution of polygons (refer to **Figure 2,3** in Chapter 2), which results in more unbalanced quadtree. **Boundary-Join** is sensitive only to data size, not to data distribution.

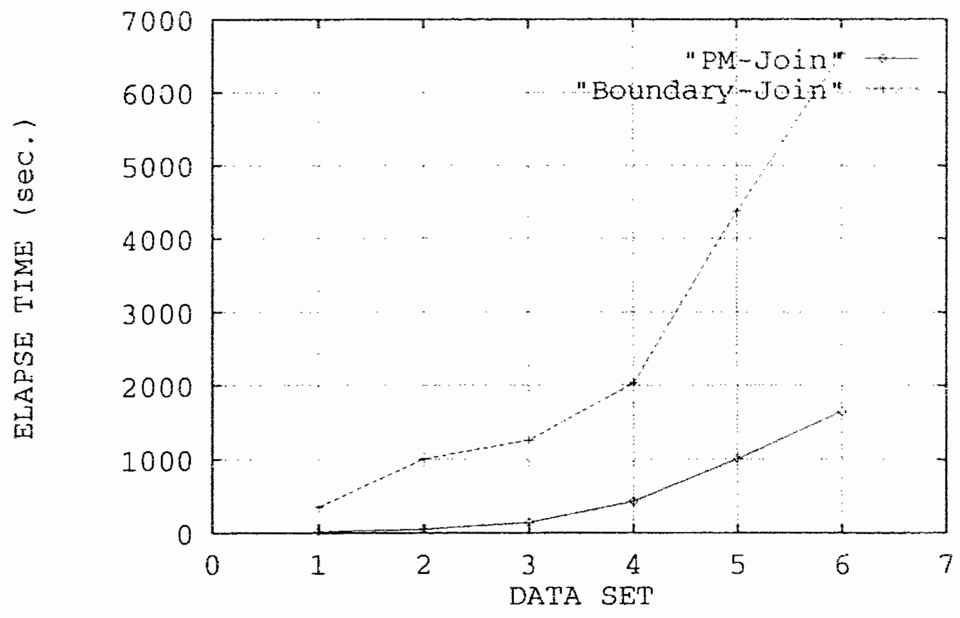

Figure 1 **PM-Join** improves over **Boundary-Join** on uniformly generated data sets

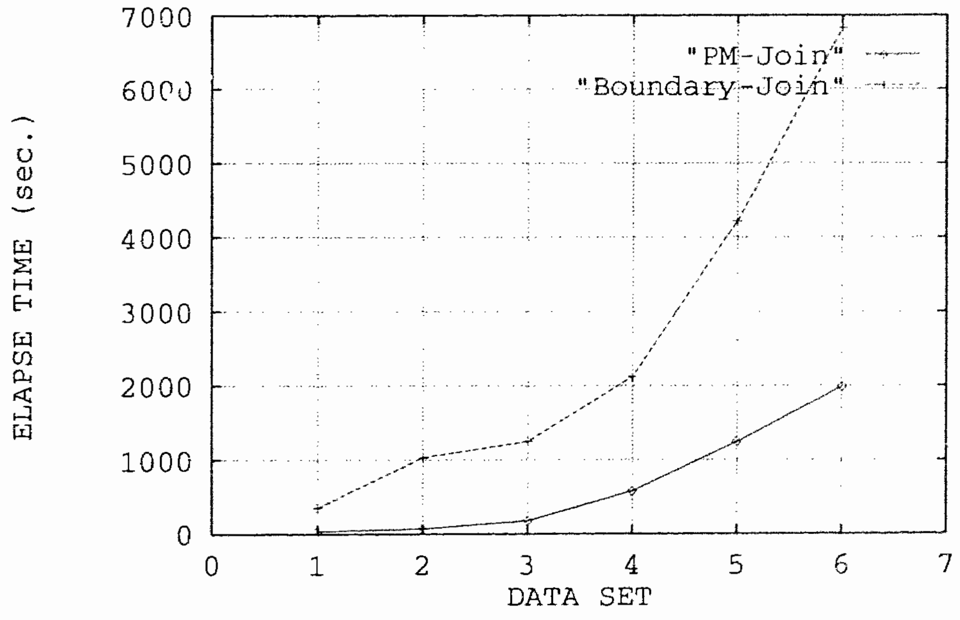

Figure 2 **PM-Join** improves over **Boundary-Join** on normally generated data sets

However, as the result of exact representation, the computation involved in **PM-Join** is quite expensive. Square clipping and node splitting are repeatedly performed on every insertion and traversal of a segment. As an example, **Table 1** below presents the time needed to construct PM3 Quadtree for one polygonal net, the total time needed to perform **PM-Join** , and the percentage of the former over the latter with respect to the first group of testing data.

| ***Elapse Time (Secs)*** | *DataSet 1* | *DataSet 2* | *DataSet 3* | *DataSet 4* | *DataSet 5* | *DataSet 6* |
|---|---|---|---|---|---|---|
| **building PM3 Quadtree** | 13.7806 | 28.1150 | 64.4320 | 177.1361 | 413.85 | 778.60 |
| **PM3IJ** | 35.0408 | 65.5513 | 159.762 | 489.732 | 1045.33 | 1752.43 |
| % | 37.10 | 42.89 | 40.33 | 36.17 | 39.59 | 44.43 |

Table 1 Constructing a PM3 Quadtree takes around 40% of the total spatial join time

About 40% of the time is spent on reorganizing the net into a tree. As intersection checking is intermingled with the construction, this is acceptable especially after considering the improvement it produces over the **Boundary-Join**. Introducing PM Quadtree has improved the performance of spatial join.

### 8.1.2 Optimize Object Accessing Space by R-tree

R-tree generates irregular grouping out of object approximations. Unlike PM Quadtree, the space at each level is not regularly decomposed and sub-spaces are overlapped. Polygon approximation MBR (Minimum Bounding Rectangle) instead of polygon itself is processed at the first stage. MBRs can be obtained by traversing each polygon once.

MBR serves as a surrogate with not precise but enough occupancy information of its belonging polygon. These surrogates are organized hierarchically and rectangle brings the simplicity of R-tree computation. Intersecting and neighboring polygon MBRs can be found efficiently by going through each level of the R-tree to check the rectangle intersections.

**Figure 3** and **Figure 4** below reflect the dramatic effect by introducing R-tree at the first stage on two group of data under different distribution.

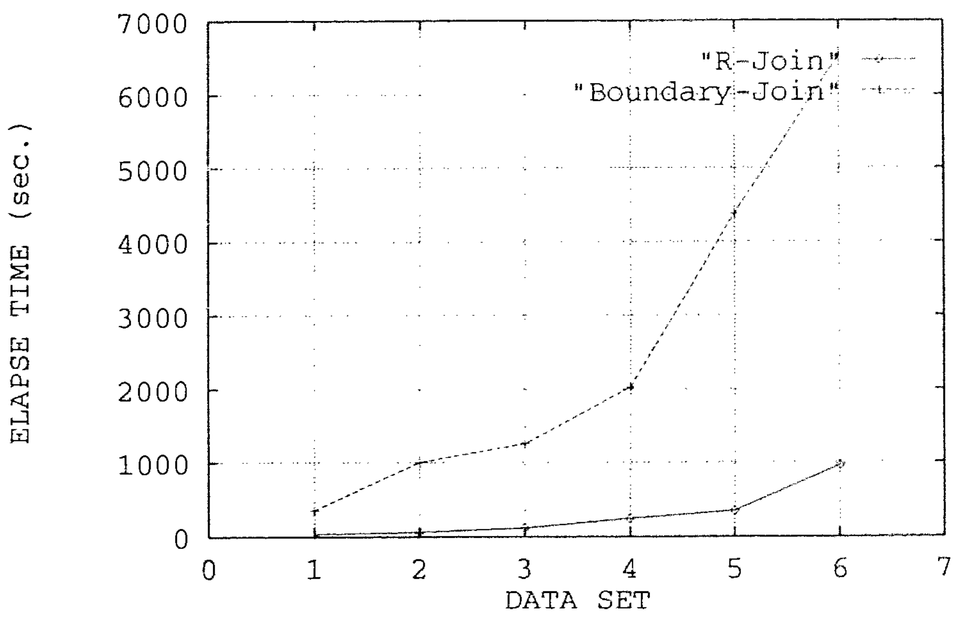

Figure 3 **R-Join** improves over **Boundary-Join** on uniformly generated data sets

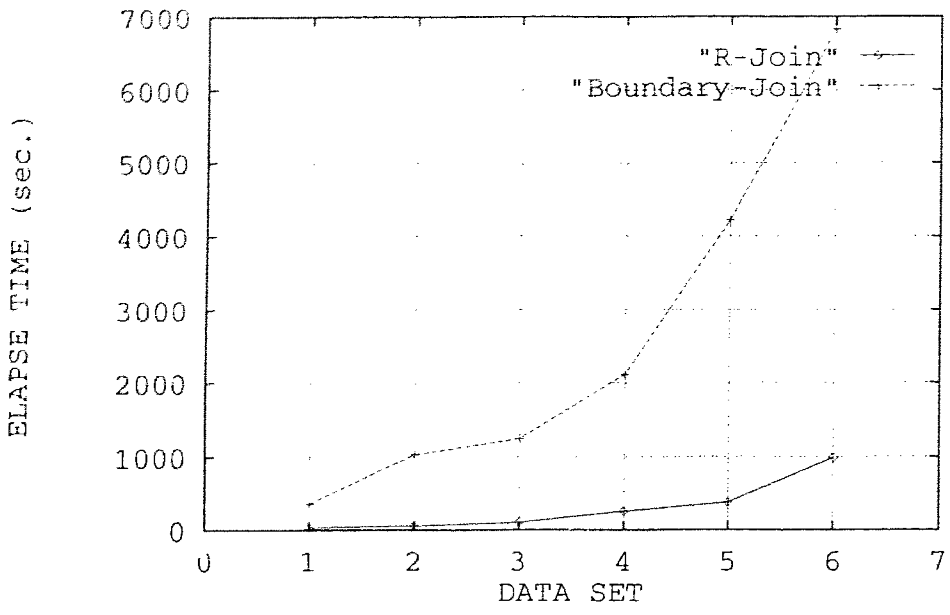

Figure 4 **R-Join** improves over **Boundary-Join** on normally generated data sets

As with PM Quadtree, curves representing the **Boundary-Join** keep growing up along with data size in both figures, while curves representing **R-Join**, remains very low as the data size grows. The contrast is sharper with larger data set. The performance of **R-Join** doesn't show much difference with different data distributions. This is due to its dynamic nature of the decomposition. By accommodating objects dynamically, data distribution does not have much impact on the performance; while the size of the single objects, and the order objects are inserted into the tree, affect the overlapping extent of sibling rectangles of R-tree, therefore affect the performance mainly.

Most significantly, R-tree brings over 90% improvement over the **Boundary-Join**. **Table 2** below presents the time needed to generate the candidate sets at the first R-tree stage, the total time needed to perform *boundary join* on the candidate sets, and the ratio of the two based on the test on first data group.

| *Elapse Time (sec)* | *DataSet 1* | *DataSet 2* | *DataSet 3* | *DataSet 4* | *DataSet 5* | *DataSet 6* |
|---|---|---|---|---|---|---|
| **first stage** | 9.5022 | 14.6804 | 30.1852 | 82.0021 | 104.323 | 336.479 |
| **Total** | 36.5430 | 62.5122 | 113.984 | 245.552 | 345.738 | 960.004 |
| % | 26.01 | 23.48 | 26.48 | 33.40 | 30.17 | 35.05 |

Table 2 Processing in the first stage takes around 30% of the total processing time

For example, for the third data set with the size of 165 × 269, the total time needed with **R-Join** is 113.984 secs, out of which 30.1852 secs is spent on generating an approximate intermediate results, and 83.799 secs are used to perform the final checking. The same data set applied to **Boundary-Join**, a total of 1667.04 secs is needed which are almost 10 times that of **R-Join**. This is a dramatic improvement, especially in the sense of the extra 30.1852 secs it spends, i.e. the additional spatial data operation taking only 27% of the performance time, generated 90% decrease of the total spatial join time on the data structure without any spatial indexing technique.

### 8.1.3 Optimize Object Accessing Space by GCS

Both PM Quadtree and R-tree make hierarchical representation of the space. Object locational information was elaborately used. Grid Coordinate System, however, organizes the space directly by dividing the space into flat, non-overlapping uniform cells, and object is represented by marking all the cells it occupies. All the objects are therefore allocated into one or more cells according to their cell occupancy, and intersection checking is based on this 2-d space division.

To obtain exact object cell occupancy would cost too much considering what is to be achieved at this stage. Instead, approximate cell occupancy is calculated.

Though computation can take advantage of the object simple approximation, precision is limited by object approximation.

From the discussion in Chapter 7, polygon chain is chosen as the processing unit and segment bounding rectangle is the basic superimposing unit.

**Figure 5** and **Figure 6** below picture the performance curves of **Grid-Join** under uniformly and normally generated data respectively, compared with that of **Boundary-Join**. *Grid resolution* was 61×61. Overall, there is about 95% decrease generated by adding the GCS on the space, which shows its striking impact on the join performance. Again, performance of **Grid-Join** under different data distribution varies only a little with respect to similar data size. This is because the space division of GCS is independent of data. Different data distributions causes different allocations of segments in the cell index array, but the amount of computation of cell occupancy as well as the final checking is not affected.

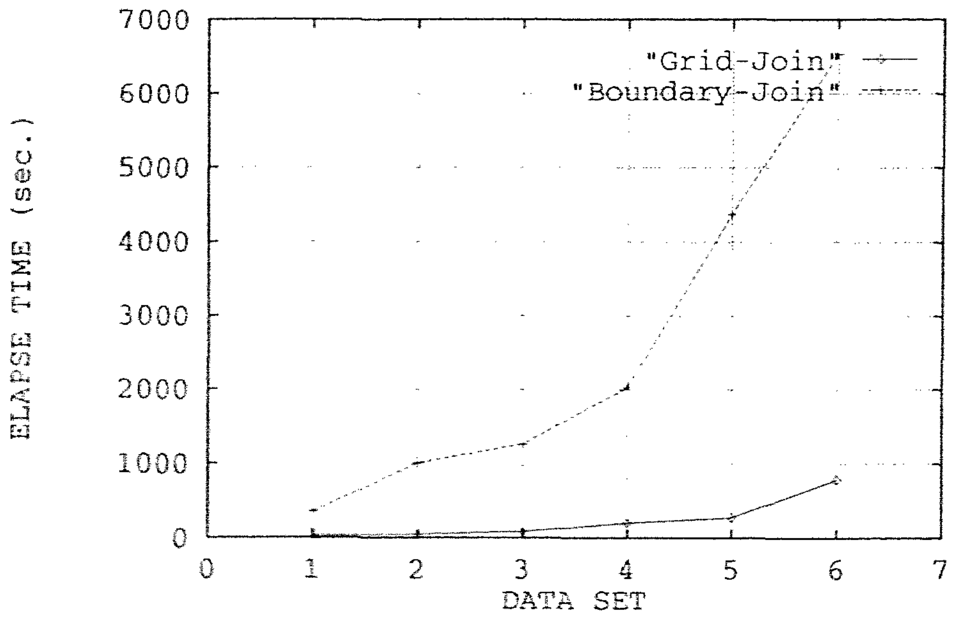

Figure 5 **Grid-Join** improves over **Boundary-Join** on uniformly generated data sets

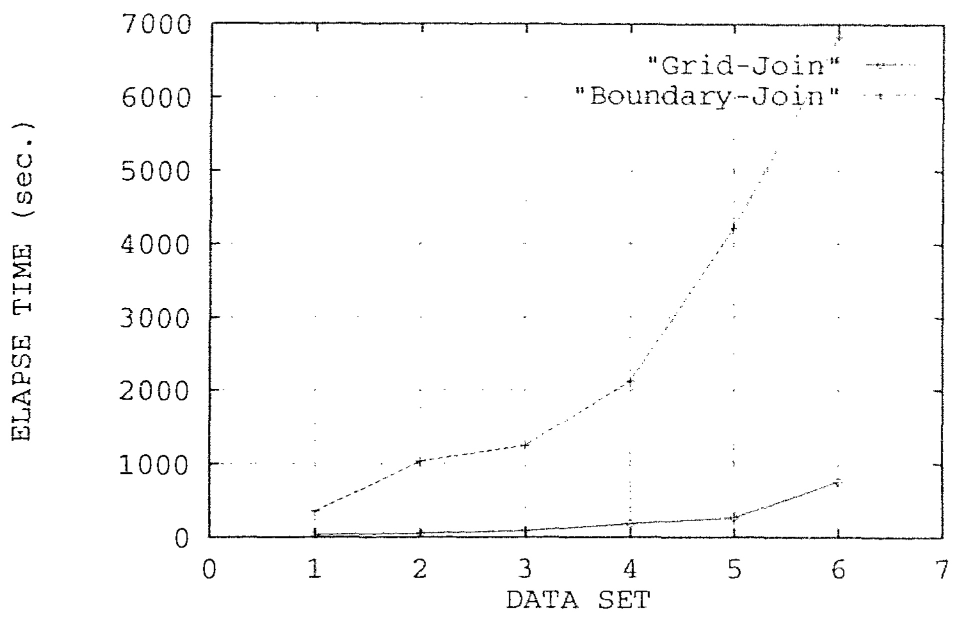

Figure 6 **Grid-Join** improves over **Boundary-Join** on normally generated data sets

**Table 3** above presents the similar comparison we had with R-tree in the above sub-section, about how overall performance is improved by the conversion effort in the first stage which takes only small percentage of the total time. Notice that 80% of the time was spent on the final checking, while converting the one map into its grid representation took only about 20% of the total time.

| ***Elapse Time (sec)*** | *DataSet 1* | *DataSet 2* | *DataSet 3* | *DataSet 4* | *DataSet 5* | *DatSet 6* |
|---|---|---|---|---|---|---|
| **Total** | 33.0405 | 47.0607 | 86.5414 | 195.253 | 239.614 | 776.85 |
| **Conver-sion** | 6.5802 | 8.6402 | 12.9502 | 21.2406 | 28.3609 | 124.011 |
| % | 19.92 | 18.36 | 14.96 | 10.88 | 11.84 | 15.96 |

Table 3 Conversion takes only 20% of the total join time

## Section 8.2 Comparing PM Quadtree, R-tree, and GCS in Spatial Join

Careful study of the three SIMs shows surprisingly that GCS outperforms both PM Quadtree and R-tree in spatial join. **Figure 7** and **Figure 8** below picture the three performance curves under the two data distributions, noticeably with that of **Grid-Join** having the lowest cost, and **R-Join** in between of the **PM-Join** and **Grid-Join** in both of the figures. **PM-Join** showed worse performance under normally generated data compared with **Grid-Join** and **R-Join**.

All of them were implemented by using ObjectStore as the storage manager, and tested over the same testing data. The timing was measured under same machine load and over contiguous time period. The algorithm itself makes the difference mainly.

In the following sub sections, we will make a general analysis of the three algorithms from the aspect of implementation technique.

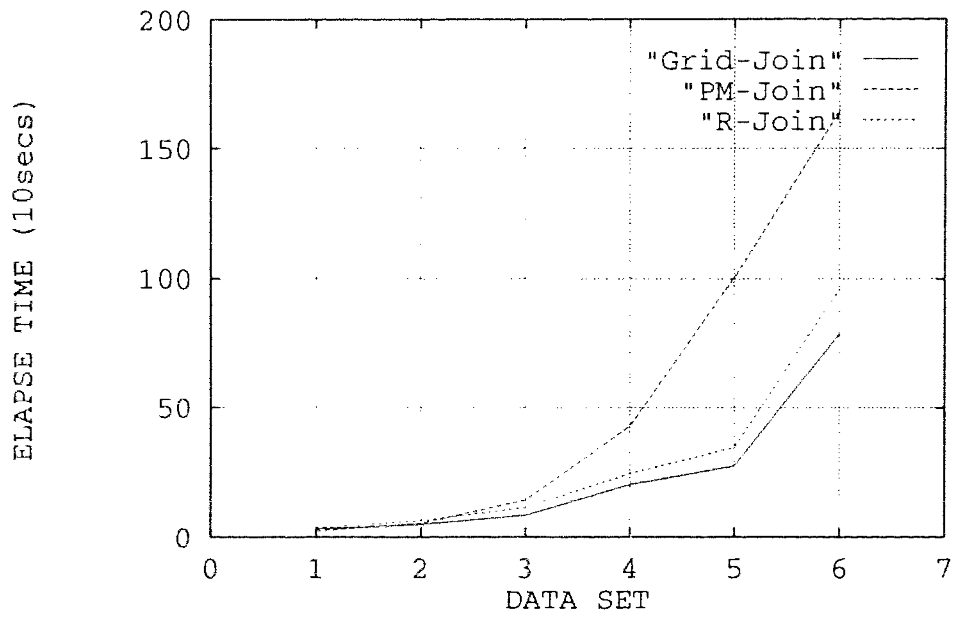

Figure 7 Compare performance of **PM-Join, R-Join, and Grid-Join** on uniformly generated data

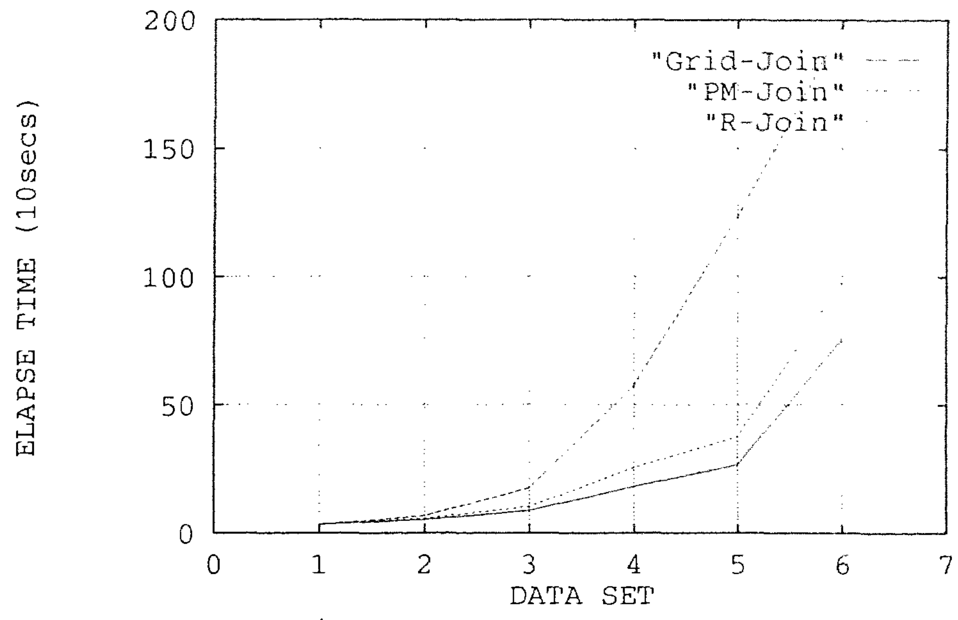

Figure 8 Compare performance of **PM-Join, R-Join, and Grid-Join** on normally generated data

### 8.2.1 PM Quadtree vs. R-tree and GCS

The most significant computational cost for **PM-Join** is *segment clipping* and *leaf node splitting*. For **R-Join**, it is mainly *MBR calculating* and *rectangle intersecting checking*. Since segment is the approximation unit in **Grid-Join**, which makes the approximation inherent in itself, the only costly computation is the *grid cell index computation*. The basic operation for all of the three is *segment intersection checking*.

*Segment clipping* is repeatedly applied for each segment and each inner node along the path due to the way the PM3 Quad-tree organizes the space. Clipping of the square requires not only range checking, but most frequently the clipping of the 4 square sides to decide whether the segment is crossing the square, or clipping on the boundary, or outside of the square. *Leaf node splitting* is relatively cheaper, because it involves only creating new PM3 node and dictionary copying, but it creates more inner node therefore more *segment clipping*. The improvement shows in algorithm **PMNSJ**(*Parallel Traversal Without Splitting Join*) over **PMSJ**

(*Parallel Traversal With Splitting Join* ) when we reduce the number of node splitting. However, all of the algorithms shows that *segment clipping* takes around 60% of the total time.

For example, with **PM-Join** , when total time needed to perform the join was 231.68 secs, 147.79 secs was spent simply on the *segment clipping.* The preciseness of the segment representation results in the smaller amount of *dictionary checking*, i.e., *segment intersection checking*, compared with that of **R-Join** and **Grid-Join**, in which there are much more *segment intersection checking* based on approximate results.

Unfortunately, the high cost of clipping can not be compensated by the precise dictionary checking. Even after we add data structure in **PM-Join** to reduce the redundancy occurred in the *dictionary checking*, performance was not greatly improved, for the main cost was not reduced. Therefore we conclude that spatial join can not bring out the best of PM Quadtree, for example, it's dynamic measure, and fine description of the object location.

### 8.2.2 R-tree vs. GCS

Unlike **PM-Join, R-Join and Grid-Join** make use of approximate information: **R-Join** operates on the polygon approximation (MBR), while **Grid-Join** operates on the segment approximation. Traversal of the segments is necessary for both of them to obtain the approximation, so it does not bring the major performance difference.

Both methods characterize in two clearly separated step of processing, i.e., generating a candidate set in the first stage, and performing actual checking in the second stage. Since *segment intersection checking* is the basic operation performed by both in the second stage, only the size of the candidate sets and the way of generating the candidate sets cause the main difference between the two.

**Table 4** and **Table 5** below lists the measurement on each step of the algorithms **R-Join and Grid-Join**.

| ***Elapse Time for Approx. Result*** | *DataSet 1* | *DataSet 2* | *DataSet 3* | *DataSet 4* | *DataSet 5* | *DataSet 6* |
|---|---|---|---|---|---|---|
| **Grid -Join** | 6.8502 | 8.6402 | 12.9502 | 21.2406 | 28.3609 | 124.011 |
| **R-Join** | 9.5002 | 14.6804 | 30.1852 | 82.0021 | 104.323 | 336.479 |

Table 4 Time spent to generate approximate results for **R-Join and Grid-Join** respectively

| ***Elapse Time for Final Check-ing*** | *DataSet 1* | *DataSet 2* | *DataSet 3* | *DataSet 4* | *DataSet 5* | *DataSet 6* |
|---|---|---|---|---|---|---|
| **Grid-Join** | 26.4603 | 38.4205 | 73.5912 | 174.012 | 211.253 | 652.839 |
| **R-Join** | 27.0428 | 47.8318 | 83.7988 | 163.549 | 241.415 | 623.523 |

Table 5 Time spent to perform the final checking on the approximate results for **R-Join and Grid-Join** respectively

In **Grid-Join**, *Grid cell index computation* is the Cartician Product of the index tuples from the division of segment coordinates by the grid size. The candidate set is therefore generated after computing each polygon chains' cell occupancy. As for **R-Join**, more complex computation has to be performed to get the candidate set. Besides building trees for each of the map, two trees intersects with each other at every level starting from the root to the leaf level to generate groupings of possible intersecting polygons. *Rectangle intersecting checking* is the main computation during this first step. As the data size grows larger, more splitting happens during the construction of the tree, and the tree becomes deeper. All of the above causes more cost for **R-Join** in the first stage than that of **Grid-Join,** which is shown in the **Table 4**. For example, with data set4, only 21.2406 secs is needed for **Grid-Join** to generate candidate set, for **R-Join** however, a total of 82.0021 secs was spent. The contrast becomes more distinctive with larger data sets.

The time spent on performing the final checking are very close shown in **Table 5** above. **R-Join** should yield a more precise approximate result than **Grid-Join**. But they each choose different processing unit. It is polygon in **R-Join**, and chain in **Grid-Join**. Polygon by polygon checking is more costly than chain by chain checking because it is based on chain by chain checking. There is more redundant checking considering that a polygon usually consists of 2 or more chains, so each checking is equivalent to 2 or 3 number of chain checking. Even if the intermediate result set is smaller, and more precise, this redundancy would bring down the total performance close to that of the chain based checking, as shown in **Table 5**. Therefore, GCS is more suitable for spatial join than both R-tree and PM Quadtree.

Overall, both **Grid-Join and R-Join** present good performance, although **Grid-Join** outperforms both **PM-Join and R-Join** with its simplicity and efficiency. **Grid-Join** also has the potential of extending to parallel implementation due to its unit independent feature. **R-Join** also produces fairly good results. They are both feasible to be adapted in spatial databases or GIS to enhance the system's functionality.

# Chapter 9
# Concluding Remarks and Summary

## Section 9.1 Conclusions

This thesis tackles the problem of polygon *spatial join* on the vector data model by extensively utilizing spatial indexing methods (SIM) for the simple spatial objects. The problem can be defined as finding all the pairs of polygon objects that overlap each other over their boundaries from the two given polygonal data sets. The polygon spatial join is one of the most important and complex operations in systems that deal with 2–dimensional objects. Applications of *spatial join* can be found largely in GIS, where geographical data is organized by "*layers*" and the joining of the *layers* creates synthesized information related to the same geographical area. Furthermore, it can be extended to obtain intersecting points and additional spatial properties to realize the *overlay* operation, which is also very important in GIS.

We solve the problem by extensively utilizing present popular SIMs such that complex objects and object relations can be handled efficiently. This is based on the observation that the spatial join relies on the object spatial occupancy, and these SIMs decompose the space from which the spatial data is drawn in such a way that the spatial properties of spatial objects can be developed and stored. Two of the representative SIMs, namely PM Quadtree and R-tree, were used. The PM Quadtree represents a disjoint hierarchical partition of space, and the R-tree represents an overlapping hierarchical partition. We observed that these SIMs are access methods for simple spatial objects like line segments and rectangles. Few studies have been seen on handling complex objects using these methods, especially studies on performances of these indexing methods in complex operations like the spatial join. We present a study of this problem in the context of polygon spatial join which requires the handling of simple polygons and compositions of search results.

We also propose the use of the Grid Coordinate System (GCS) — a spatial indexing method for simple spatial objects as a version of Grid File based on the object spatial occupancy instead of on the transformed multidimensional point space. Unlike the R-tree and PM Quadtree's hierarchical partitioning of the space, GCS presents a non-disjoint, non-hierarchical uniform grid space. We show how cell indices can be computed and objects are grouped into relating cells according to their approximations' cell occupancy in the GCS.

We design and implement polygon spatial join algorithms based on PM Quadtree, R-tree and GCS respectively. We also design and implement the spatial join algorithms with no spatial indexing involved for comparisons. The spatial join results of polygons' can be derived by the topology implied in the vector data model with segment-based PM Quadtree. Spatial join based on the R-tree and GCS, however, are realized by incorporating a "buffering" technique into the processing. The "buffering" technique generates approximate result by processing the "boxes" which are surrogates of the complex objects. With previous experiments, lack of topology of these object surrogates leads to an exhaustive process to obtain this conservative result. When organized by the R-tree and GCS, the relative positions of these surrogates can be derived out of their spatial extent, and therefore facilitates the retrieval of objects of interest during the spatial join processing. In other words, SIM optimizes the "buffering" technique.

The "buffering" technique enables a two-step processing of complex objects for R-tree and GCS. We justify that this will lead to a reasonable performance of spatial join by using SIMs of simple objects. It is based on the observation that the performance of polygon spatial join is seriously bounded by the redundant *segment intersection checking* operation under vector representation. Object approximations combined with their efficient organization would yield a conservative yet fairly concise result to eliminate those potentially unqualified object pairs. The redundant *segment intersection checking*s are therefore greatly reduced.

As the spatial join is a complex query, the operation is not only search-based, but also involves extensive computation of the search results. To compare the performance of these SIMs is a sophisticated experiment in the sense that SIMs are used differently in each of the algorithms designed because of their different ways of partitioning and representing the space, and various distributions of the underlying data. Since each SIM has multiple factors affecting its performance, and the experiment is carried out in a multiple client/server network environment, it is not complete to measure the performance of each SIM only in terms of disk accesses as it is with search-based operations such as range query and point query.

We believe that the measurement of total execution time is more accurate in this particular context, and we show an extensive comparison of the three SIMs after initially experimenting on each of the SIMs to select a best representative for the comparison of SIMs. We generate random polygonal nets with different data distributions as the test data for all of the experiments.

In the following section, we summarize what we have achieved from the aspects we described above.

## Section 9.2 Thesis Summary

### 9.2.1 Utilizing PM Quadtree Extensively to Realize Spatial Join

The PM Quadtree regularly and recursively decomposes the space into quadrants until a fine representation of segments is obtained. The algorithms for *spatial join* based on PM Quadtree are *Parallel Traversal With Splitting* (**PMSJ**), *Parallel Traversal Without Splitting* (**PMNSJ**), and *PM Quadtree Index Join* (**PMIJ**).

**PMSJ** and **PMNSJ** are characterized by their simultaneously searching on two PM quadtrees, and checking with the corresponding nodes. A grey node implies further traversing of the tree, while a leaf node leads to the objects of interest. The leaf node is further split with **PMSJ** if the corresponding node is not leaf. **PMNSJ** proceeds by checking through all the dictionaries in the sub-trees with grey nodes as the root. **PMIJ** uses quadtree as an index to search for the most likely intersecting segments. A result matrix is used in all of the three algorithms to keep track of the polygon intersection information. Testing on the random polygonal net shows **PMIJ** has better performance.

There are three PM Quadtree variants:PM1, PM2, and PM3. The PM2 and PM3 Quadtrees are obtained by successively weakening the definition of what constitutes a valid leaf node. The PM3 Quadtree has the weakest requirement for ending blocks which results in a search with the least depth and least number of leaves. It decomposes space to a fine extent, but the segments are not as seriously fragmented as they are in the PM1 and PM2 Quad-trees. Algorithms based on the PM3 Quadtree therefore should yield better performance than that of PM1 and PM2 Quad-trees.

A practical solution, **PM3IJ** is based on **PMIJ** of its PM3 quadtree variant. To reduce the redundant dictionary checking, an extra data structure is used as a "filter" to keep track of the possible intersecting segments during the traversal of the quadtree for each chain segment without repetition. This results in a two step processing with deferred dictionary checking. **PM3IJ** showed its improvement over the rest of the algorithms based on PM3 Quadtree, therefore, we use **PM3IJ** as the representative of PM Quadtree based spatial join, and we name it **PM-Join** for the later comparison.

### 9.2.2 Utilizing R-tree Extensively to Realize Spatial Join

The R-tree decomposes the space dynamically dependant on the input data. Since it processes only rectangles, the complex objects are reduced to their minimum bounding rectangles (MBR), and therefore can be represented by R-tree. Any operations on the tree would generate only a conservative result. Since MBR preserves object spatial extent, the tree structure constructed from object MBRs provides additional topology in terms of object intersections.

The spatial join algorithms based on R-tree feature two step processing with the first step generating possible intersecting polygon pairs through the use of R-trees, and the second step performing the chain-by-chain intersection checking. Polygons are chosen as the basic processing unit. The different processing of R-trees in the first step leads to two spatial join algorithms: *R-tree Index Spatial Join* (**RIJ**) with *index join* as the first step, and *R-tree Parallel Comparison Spatial Join* (**RPJ**) with *parallel join* as the first step.

*Index join* uses a R-tree as an index to search for the interesting MBRs, while *parallel join* produces candidate pairs by comparing two R-trees in parallel. Each node in one tree is compared with all the nodes at the corresponding level in the other tree because sibling nodes at each level could overlap. The outcomes of the comparison at each level are kept in two local stacks and passed on to the next level for corresponding checking until both leaf levels are reached.

Analysis of both algorithms were emphasized on their first step, i.e., *index join* and *parallel join* respectively. If $n_1$ and $n_2$ are the number of MBRs in each data set, and the tree is of order(M,m), then the *index join* has the best case complexity $O[n_1 \times \log_m n_2]$, and the worst case complexity $O[n_1 \times (n_2+n_2')]$, where $n_1'$ and $n_2'$ stand for the maximum number of non-leaf nodes in the corresponding R-trees. *Parallel join* has the best case complexity $O[\mathrm{Max}(\log_m n_1 \times \log_m n_2]$, and the worst case complexity $O[(n_1+n_1')(n_2+n_2')]$. In the average case, *parallel join* generates candidate set faster than *index join* does.

*Node size* M is an optimization parameter of the algorithms. Generally, a large *node size* results in better performance time than a small *node size*. It should be chosen experimentally with regard to different data size. Experiments on the randomly generated data with around 1000 polygons in each data set show better performance when *node size* is chosen larger than 10. As with PM Quadtree, we use the algorithm *R-tree Parallel Comparison Spatial Join* for the later comparison, and we name it **R-Join** to distinguish it from PM Quadtree based **PM-Join**.

### 9.2.3 Grid Coordinate System (GCS) and its Spatial Join

Grid Coordinate System exhibits a uniformly partitioned space composed of disjoint cell units. Each cell unit has a unique number decided by its column and row number. By superimposing the GCS on the object approximation space, objects are divided into groups according to their approximations' occupancy of cell units. Object cell occupancy can be calculated by the Cartisian Product of the object's occupying row numbers and column numbers. The calculation of cell occupancy together with the object approximation causes redundancy and inaccuracy. But the conversion of objects to grid representation is simplified and fast, and reduces computation of an operation by performing the operation on the objects belonging to the same group.

Spatial join based on GCS is realized by the algorithm *Cell Spatial Join* (**CSJ**). **CSJ** also features two-step processing. It applies the *Net-Conversion* algorithm first to transfer the object approximation into a grid representation, and then checks for the actual intersections according to the grid representation. The segment is chosen as the basic approximate object to be clipped on the GCS, and the chain is the processing unit to reduce the amount of calculation in each grid cell. The grid representation is implemented as a cell index array. The cell index array is referenced by cell indices, and each array element points to a linked chain list. Intersection checking is performed cell by cell, and result is recorded in the result matrix.

General analysis of **CSJ** also indicates that the underlying data distribution and the data size affect the algorithm's performance . *Grid resolution* can be adjusted to suit data sets with different sizes and distributions in order to obtain low *chain volume* and a reasonable number of (*cell index, chain index*) pairs.

*Grid size* is decided by *grid resolution*. To obtain a low *chain volume*, the segments should not be overly fragmented by the chosen GCS, i.e., average *segment length* should not be too large compared with *grid size*. It should be about the size of the grid, or larger but not too much. *Grid resolution* is bounded by the average *segment length*. Besides, a very large *grid size* means too many number of computation units.

The experiments on the random net verify that best performance can be achieved under a certain *grid resolution* we call *optimal resolution*. Performance drops on both sides of the *optimal resolution*. Carefully designed experiments also reveal that under *optimal resolution*, the number of (*cell index, chain index*) pairs is approximately 1 or 2 times of the total grid number, and the average *segment length* is greater than 2 but less than 10 times of the *grid size*.

We use **CSJ** as the representative of teh GCS based spatial join in the later comparison, and we name it **Grid-Join** to distinguish it from both PM Quadtree based **PM-Join** and R-tree based **R-Join**.

### 9.2.4 Comparison of Algorithms and Their Underlying Spatial Indexing Methods

The **PM-Join, R-Join** and **Grid-Join** all improve over the **Boundary-Join** dramatically. The **Boundary-Join** is a spatial join algorithm without utilizing any spatial indexing method. This shows that introducing SIM can indeed improve spatial join under vector data model. Empirical results also show that both the **R-Join** and the **Grid-Join** outperform **PM-Join** with fairly good performance on all of the random testing polygonal nets. The **Grid-Join** shows slightly better performance than the **R-Join** in this context.

The GCS appears ideal for uniformly distributed data, while the PM Quadtree is suited for the arbitrarily distributed data. In general, since spatial data is not usually uniformly distributed as the randomly generated data set we had, the PM Quadtree's regular decomposition approach is more flexible, therefore, PM Quadtree based retrieval and simple operations should yield better average performance. However, when the operation requires the composition of search operations and results, like spatial join, the way that the objects are represented in each SIM decides the way that data are further processed, and therefore greatly affects the overall performance. It is observed that the **PM-Join** involves expensive operations like segment clipping and node splitting which are applied frequently, while the computations involved in **Grid-Join** and **R-Join** like calculating cell occupancy and rectangle intersection checking, are relatively simpler and less frequent. The fine description of objects by PM3 Quadtree can not be fully utilized by the algorithms of spatial join based on PM Quadtree.

The GCS overcomes the PM Quadtree's computational overhead by using an approach similar to R-tree's. But the R-tree decomposes the space dynamically so that different data environments can be accommodated, whereas with the GCS, the decomposition induced are static. Therefore, R-tree can generate a more accurate candidate set. However, **R-Join** does not outperform **Grid-Join** dramatically since it takes less time for GCS to generate a candidate set, although it is less accurate, by taking advantage of its simple computation of cell occupancy. Furthermore, final intersection checking still occupies most of the total spatial join time. Overall, both the GCS and the R-tree are feasible for the spatial join operation.

## Section 9.3 Future Work

All of the algorithms designed can be extended to realize the *Total Spatial Join* as defined in the introductory chapter by adding a point-in-polygon algorithm to determine the containment or enclosure relation of polygons. But it should be applied on different result sets with different spatial join algorithms. For example, for R-tree based algorithms, containment and enclosure test should be applied only on the non-intersecting pairs of the candidate set, since the candidate set includes possible containing or enclosure pairs. While for both PM quadtree based and GCS based algorithms, the test has to be applied also on the pairs that are not in the candidate set. So R-tree is more suitable when both *partial spatial join* and *total spatial join* are intended.

To realize *spatial overlay* by extending the *spatial join* algorithm is not straightforward. Generally the spatial join algorithms can be modified to record all the intersecting points, and the resulting polygonal net can be traced out by processing the two polygonal nets together with the resulting set. More sophisticated methods can be studied by using the PM Quadtree and R-tree directly, and generating a resulting PM Quadtree or R-tree which represents the overlaid map.

When modifying the algorithms to accommodate *line* objects, we could have an extensive performance comparison of these spatial indexing methods in the context of performing set operation on the large number of line segments. As line data is much simpler, performance relies more on the indexing structure themselves, hence the conclusion could be different from that of polygon's.

Experiments can also be extended by implementing the algorithms on top of existing spatial databases, and making use of their underlying index structure directly, for example, a spatial database which uses R-tree as its secondary index to speed up object retrieval. Indices can be loaded into memory directly to generate a much smaller candidate set, then perform the actual intersection checking by loading in the actual objects. In this case, the number of disk access plays an important role on the algorithm performance. Therefore it requires different performance measurements.

Although the spatial indexing methods for simple objects can be extended to process complex spatial objects, these extended methods should be compared with the typical spatial indexing methods of complex objects such as the Cell Tree, in terms of retrieval, insertion and deletion of complex objects as well as implementation complexity and storage since indexing methods for complex objects are inherently more complicated. The comparison appears rare in the literature. Techniques other than approximation to extend the indexing structure of simple objects for the processing of complex objects should also be explored.

# References

[ARON89] S. Aronoff, Geographical Information Systems: A Management Perspective, WDL Publications, c1989.

[BEN75] J. L. Bentley, Multidimensional Binary Search Trees Used for Associative Searching, *Communications ACM*, Vol. 18, No. 9, pp. 509–517, 1975.

[BEN79] J. L. Bentley, Multidimensional Binary Search Trees in Database Applications, *IEEE Transactions on Software Engineering*, Vol. SE–5 No. 4, July 1979.

[BIMA90] H. Blanken, A. Ijbema, P. Meek, B. V. D. Akker, The Generalized Grid File: Description and Performance Aspects, *Proc. of 6th International Conference on Data Engineering*, pp. 380–388, Feb. 1990.

[BKSS90] N. Beckmann, H. Kriegel, R. Schneider, B. Seeger, The $R^*$-tree: An Efficient and Robust Access Method for Points and Rectangles, *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 322–331, May 1990.

[BW90] I. Bracken, C. Webster, Information Technology in Geography and Planning, Routledge 1990.

[CODD70] E. F. Codd, A Relational Model for Large Shared Data Banks, *Communications ACM*, Vol. 13, No. 6, pp. 377–387, 1970.

[FRAN90] Wm. R. Franklin, Calculating Map Overlay Polygons' Areas Without Explicitly Calculating the Polygons-Implementation, *Proc. of the 4th International Symposium on Spatial Data Handling*, Vol. 1, pp. 151–160, 1990.

[GH91] R. Gupta, E. Horowitz, Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD, Reading, *Prentice Hall Series in Data and Knowledge Base Systems*, 1991.

**[GRE89]** D. Greene, An Implementation and Performance Analysis of Spatial Data Access Methods, *Proc. of 5th International Conference on Data Engineering*, pp. 606–615, May 1989.

**[GUN89]** O. Gunther, The Design of the Cell-tree: An Object-Oriented Index Structure for Geometric Databases, *Proc. of the 5th International Conference on Data Engineering*, pp. 598–605, Feb. 1989

**[GB90]** O. Gunther, A. Buchmann, Research Issues in Spatial Databases, *SIGMOD RECORD*, Vol. 19, No. 4, pp. 61–68, December 1990.

**[GUN88]** O. Gunther, Efficient Structures for Geometric Data Management, *Lecture Notes in Computing Science 337*, Springer 1988.

**[GUTT84]** A. Guttman, R-Trees: A Dynamic Index Structure for Spatial Searching., *Proc. ACM SIGMOD*, pp. 47–57, June 1984.

**[GS90]** M. F. Goodchild, Y. Shiren, A Hierarchical Data Structure for Global Geographic Information Systems, *Proc. of 4th International Symposium of Spatial Data Handling*, Vol. 2, pp. 911–917, 1990.

**[HS92]** E. G. Hoel, H. Samet, A Qualitative Comparison Study of Data Structures for Large Line Segment Databases, *Proc. ACM SIGMOD*, pp. 205–214, June 1992.

**[KSSS90]** H. Kriegel, M. Schiwietz, R. Schneider, B. Seeger, Performance Comparison of Point and Spatial Access Methods, *Design and Implementation of Large Spatial Databases*, Lecture Notes in Computer Science 409, pp. 89–114, July 1989.

**[KW87]** A. Kemper, M. Wallrath, An Analysis of Geometric Modeling in Database Systems, *ACM Computing Surveys*, Vol. 19, No. 1, pp. 47–119, March 1987.

**[ML84]** D. M. Mark, J. P. Lauzon, Linear Quadtrees for Geographic Information Systems, *Proc. of the 2nd International Symposium on Spatial Data Handling*, Vol. 2, 1984.

**[MO86]** F. Manola, J. A. Orenstein, Toward a General Spatial Data Model for an Object-Oriented DBMS, *Proc. of the 12th International Conference on Very Large Data Bases*, pp. 328–335, August 1986.

**[NHS84]** J. Nievergelt, H. Hinterberger, K.C. Sevcik, The Grid File: An Adaptable, Symmetric Multikey File Structure, *ACM Transactions on Database Systems*, Vol. 9, No. 1, pp. 38–71, March 1984.

**[NOR88]]** V. T. Noronha, A Survey of Hierarchical Partitioning Methods for Vector Images, *Proc. of the 3rd International Symposium on Spatial Data Handling*, Vol. 1, pp. 185–200, 1988.

**[NW79]** G. Nagy, S. Wagle, Geographic Data Processing ,*Computing Surveys*, Vol. 11, No. 2, pp. 139–163, June 1979.

**[OHMS92]**J. A. Orenstein, S. Haradhvala, B. Margulies, D. Sakahara, Query Processing in the ObjectStore Database System, *Proc. ACM SIGMOD* , pp. 403–412, June 1992.

**[OM88]** J. A. Orenstein, F.A. Manola, PROBE Spatial Data Modeling and Query Processing in an Image Database Application, *IEEE Transactions on Software Engineering*, Vol. 14, No. 5, pp. 611–629, May 1988.

**[OOST90]** P. V. Oosterom, Reactive Data Structures for Geographical Information Systems, Reading, ADDIX, Wijk bij Duurstede, 1990.

**[OREN84]** J. A. Orenstein, T. H. Merrett, A Class of Data Structures for Associative Searching *Proc. of 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pp. 181–190, 1984.

**[OREN86]** J. A. Orenstein, Spatial Query Processing in an Object-Oriented Database System *Proc. ACM SIGMOD*, pp. 326–335, 1986.

**[OREN89]** J. A. Orenstein,Redundancy in Spatial Databases *Proc. of ACM SIGMOD Conference on Management of Data*, pp. 294–305, June 1989.

**[OSM89]** B. C. Ooi, R. Sacks-Davis, K. J. McDonell, Extending A DBMS for Geographic Applications *IEEE Proc. of 5th International Conference on Data Engineering*, pp. 590–597, Feb. 1989.

**[PS85]** F. P. Preparata, M. I. Shamos, Computational geometry: An Introduction , Reading, Springer-Verlag, 1985.

**[RF88]** N. Roussopoulos, C. Faloutsos, An Efficient Pictorial Database System for PSQL *IEEE Transactions on Software Engineering*, Vol. 14, No. 5, pp. 639–650, May 1988.

**[ROB81]** J. T. Robinson, The K-D-B-tree: A Search Structure for Large Multidimensional Dynamic Indexes. *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 10–18, 1981.

**[SAM84]** H. Samet, The Quadtree and Related Hierarchical Data Structures *ACM Computing Surveys 16*, pp. 187–260, June 1984.

**[SAM88]** H. Samet, Hierarchical Representations of Collections of Small Rectangles *ACM Computing Sueveys 20*, pp. 271–309, 1988.

**[SAM90a]** H. Samet, The Design and Analysis of Spatial Data Structures, Reading, Addison-Wesley, MA, 1990.

**[SAM90b]** H. Samet, Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS, Reading, Addison-Wesley, MA, 1990.

**[SE90]** J. Start, J. E. Estes, Geographic Information Systems: An Introduction, Reading, Prentice Hall, 1990.

**[SRF87]** T. Sellis, N. Roussopoulos, C. Faloutsos, The R+_Tree: A Dynamic Index for Multi-Dimensional Objects. *Proc. of the 12th International Conference on Very Large Data Bases*, pp. 507–518, Sept. 1987.

**[SW89]** H. Samet, R. E. Webber, A Comparison of the Space Requirements of Multidimensional Quadtree-based File Structures , *Visual Computer*, 1989.

**[TJS88]** A. T. Teng, S. A. Joseph, A. R. Shojaee, Polygon Overlay Processing: A Comparison of Pure Geometric Manipulation and Topological Overlay Processing *Proc. of the 3rd International Symposium on Spatial Data Handling*, Vol. 1, pp. 102–119, 1988.

**[ULL88]** J. D. Ullman, Principles of Database and Knowledge-base Systems, Vol. 1, *Principles of Computer Science Series*, Reading, 14. Computer Science Press, 1988.