FAST ALGORITHMS FOR DISTANCE-BASED SPATIAL QUERIES

By

Jingjing Lu

A thesis submitted in conformity with the requirements for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Copyright © 2001 by Jingjing Lu
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

0-612-62968-6
Abstract

Fast Algorithms for Distance-based Spatial Queries

Jingjing Lu
Master of Science
Graduate Department of Computer Science
University of Toronto
2001

Spatial databases can be used in many computing-related disciplines to represent objects in multi-dimensional spaces. One important type of query in such systems involves retrieving all objects within a specified distance of a specified query point. In some cases, the answers to such queries are more useful if they are presented to the user with objects ordered by increasing distance from the query point.

In this thesis, we propose two algorithms for answering the Sphere Intersection Query (SIQ) and the Distance Query (DQ). SIQ is a special case of the Range Query where the query window is a sphere instead of an iso-oriented rectangle. Our DQ algorithm is incremental in the sense that objects are reported one by one in ascending order of distance. This way, a query processor can use the algorithm in a pipeline fashion for complex queries. Both of these two algorithms make use of the Filter Tree, a hierarchical spatial data structure, as an index.

The performance of each of the two algorithms is evaluated through both analysis and experimentation with a prototype implementation. Experiments are performed using datasets consisting of both points and line segments.
Acknowledgements

I would like to extend my sincere gratitude to my supervisor Professor Kenneth Sevcik. I deem it a great pleasure to have the opportunity to work with Ken. I have learned a lot from his clarity of thought, his consistence and his charming personality. Without his brilliant insights and strict analysis, I would never bring this project to fruition. He is a true mentor.

I would also like to thank my second reader Professor Renee Miller, thank her for her precious time and highly valued comments.

As a personal achievement, I dedicate this thesis to my dear parents. They have shown me the power of having a spirit of generosity and unconditional love. Their continuous support and encouragement have always been my impetus to accomplish my next goal in life. All I have accomplished and I will ever be I owe to my loving mother and father.

I also am grateful to my friends at University of Toronto, thank all of them for sharing the cherishable graduate life with me. Especially, I would like to thank George Giakkoupis. George, highly intelligent, can always come up with a better solution to the question that I have struggled for days. Brett Kennedy provides me with a lot of highly insightful suggestions and help at the beginning of this project.

Last, but not the least, I would thank Cara for her moral support.
Contents

1. INTRODUCTION ........................................................................................................... 1
   1.1 SPATIAL DATA ........................................................................................................... 1
   1.2 SPATIAL QUERIES ..................................................................................................... 2
   1.3 SPATIAL ACCESS METHODS ..................................................................................... 3
   1.4 THESIS ORGANIZATION .......................................................................................... 6

2. RELATED WORK ............................................................................................................. 7
   2.1 INCREMENTAL ALGORITHMS TO (K-) NEAREST NEIGHBOR QUERIES ............. 7
      2.1.1 The k-d Tree and Broder's Algorithm ............................................................... 7
      2.1.2 The LSD-Tree and Henrich's Algorithm ........................................................... 8
      2.1.3 PMR Quadtree and Hjaltason & Samet's Algorithm ........................................ 9
      2.1.4 R-Tree Family ..................................................................................................... 10
         2.1.4.1 R-Tree .......................................................................................................... 10
         2.1.4.2 R+-Tree ....................................................................................................... 11
         2.1.4.2 R*-Tree ....................................................................................................... 12
      2.2 OVERVIEW OF THE FILTER TREE ...................................................................... 13
         2.2.1 Range Queries .................................................................................................. 15
         2.2.2 Spatial Join Queries ......................................................................................... 17

3. PROBLEM STATEMENT ................................................................................................. 19
   3.1 MOTIVATION ............................................................................................................ 19
   3.2 CRITERIA FOR EFFICIENCY .................................................................................... 20
   3.3 SPHERE INTERSECTION QUERY AND DISTANCE QUERY ................................... 21
      3.3.1 Sphere Intersection Queries ............................................................................... 22
      3.3.2 Distance Queries ............................................................................................... 23
   3.4 ACCURACY IN DISTANCE CALCULATION .............................................................. 25
Chapter 1

Introduction

With an increasing number of computer applications that rely heavily on spatial data, more and more attention has been devoted to spatial data management. Besides the original applications, like cartography, mechanical CAD and GIS (Geographic Information System), nowadays spatial data are heavily used in the areas such as robotics, visual perception, environmental and medical imaging. A great variety of spatial access methods have been proposed in the past two decades to make the access to the spatial data more efficient.

1.1 Spatial Data

Spatial data is the data that describes the space and its embedded objects including points, lines, areas and volumes. Generally, we assume that the given objects are embedded in $d$-dimensional Euclidean space $E^d$ or a suitable subspace thereof. Since indices often perform more efficiently when handling simple entries of a consistent form, we often abstract a simpler shape from the actual shape of a spatial object before inserting it into an index. For example, in a city geographic database, buildings may be stored as points; rivers and roads can be stored as a set of line segments; and parking lots, playgrounds and water areas can be stored as minimum bounding rectangles (MBR’s), the smallest rectangles that are aligned with the axes of the two-dimensional space and enclose the entities’ shapes. An index is defined to administrate the simplified shape of each object while a pointer to the actual object database is maintained.
Different techniques are used to represent the spatial data. For example: an MBR can be specified either by the coordinates of the lower left corner and the upper right corner, or by the coordinate of the centroid of the MBR and the extent in each dimension.

1.2 Spatial Queries

Spatial queries are queries against one or more spatial datasets. Since there is neither standard algebra nor a standard spatial query language defined on spatial data, we will have to define several commonly used spatial database operators, which will be addressed in this thesis.

Query 1 (Range Query $RQ$). Given a $d$-dimensional interval $I^d = [l_1, u_1] \times [l_2, u_2] \times \ldots \times [l_d, u_d]$, find all objects $o$ with spatial extent $o.G \subseteq E^d$ having at least one point in common with $I^d$: $RQ(I^d) = \{ o | I^d \cap o.G \neq \emptyset \}$.

For a range query, the window is iso-oriented; that is, its faces are parallel to the coordinate axes. A more general variant of the region query permits search regions to have arbitrary orientations and shapes.

Query 2 (Intersection Query $IQ$, Region Query, Overlap Query). Given an object $o'$ with spatial extent $o'.G \subseteq E^d$, find all objects $o$ having at least one point in common with $o'$:

$$IQ(o') = \{ o | o'.G \cap o.G \neq \emptyset \}.$$ 

As a special case of Intersection Query, we propose Sphere Intersection Query in this thesis where the object $o'$ is sphere.

Query 3 (Nearest-Neighbor Query $NNQ$). Given an object $o'$ with spatial extent $o'.G \subseteq E^d$, find all objects $o$ having a minimum distance from $o'$:

$$NNQ(o') = \{ o | \forall o'': dist(o'.G, o.G) \leq dist(o'o', o'o''G) \}.$$
CHAPTER 1. INTRODUCTION

**Query 4 (Distance Query DQ).** Given an object \( o' \) with spatial extent \( o' . G \subseteq E^d \), find up to the nearest \( N \) objects \( o_1, o_2, \ldots, o_N \) having a distance between \([dist_{\text{min}}, dist_{\text{max}}]\) from \( o' \) in the ascending order of distance:

\[
DQ(o') = \{ o_i \mid \forall \ i \leq j \leq N: dist_{\text{min}} \leq dist(o'.G, o_i.G) \leq dist(o'.G, o_j.G) \leq dist(o'.G, o_N.G) \leq dist_{\text{max}} \}.
\]

The Nearest-Neighbor Query can be regarded as a special case of Distance Query, where \( N \) (the number of records in the result set) equals 1, and \( dist_{\text{min}} \) equals 0 and \( dist_{\text{max}} \) is unlimited.

The distance between extended spatial data objects is usually defined as either the distance between the centroids or the distance between the closest points. Common distance functions for points include the Euclidean and Manhattan distances. In this thesis, we will use the distance between the centroids under Euclidean metrics. A discussion why this distance measure is more appropriate than the distance between the closest points will be presented in Chapter 3.

Besides spatial selections, as exemplified by Queries 1 through 4, the spatial join is one of the most important spatial operations and can be defined as follows [Gunther 1993]:

**Query 5 (Spatial Join).** Given two collections \( R \) and \( S \) of spatial objects and spatial predicate \( \theta \), find all pairs of objects \( (o, o') \in R \times S \) where \( \theta (o.G, o'.G) \) evaluates to true:

\[
R \bowtie \theta S = \{ (o, o') \mid o \in R \land o' \in S \land \theta (o.G, o'.G) \}.
\]

The spatial predicate \( \theta \) has a wide variety of possible interpretations, including \( \text{intersects}() \), \( \text{contains}() \), \( \text{is\_enclosed\_by}() \), \( \text{distance}() \) \( \Theta \in \{ =, \leq, <, \geq, > \} \).

1.3 Spatial Access Methods

Spatial access methods (SAMs) are typically based on point access methods (PAMs) which are designed to handle sets of data points. However, PAMs are not directly applicable to databases containing objects with a spatial extension. The main problem when designing the spatial access methods is that there exists no total ordering among
spatial objects that preserves spatial proximity. In other words, there is no mapping from two- or higher-dimensional space such that any two objects that are spatially close in the higher-dimensional space are always close to each other in the one-dimensional sorted sequence. Also, the fact that spatial data is usually complex and large and there is no standard algebra defined on spatial data makes the spatial operations more expensive than standard relational operations. All these factors make the design of efficient access methods in the spatial domain more challenging than the conventional database.

In order to deal with the special properties of the extended objects, point access methods have been modified using the following techniques: Transformation, Overlapping Regions, Clipping and Multiple Layers [Kriegel et al. 1991]. The idea of Transformation is to transform the objects into a different representation so that one-dimensional access methods can be applied to manage spatially extended objects. Essentially there are two options: one can either transform each object into a higher-dimensional point [Hinrichs 1985; Seeger and Kriegel 1988], or transform it into a set of one-dimensional intervals by means of space-filling curves. The main idea of the Overlapping Regions technique is to allow different data buckets in an access method to correspond to mutually overlapping subspaces. Typical Overlapping Region methods include the $R$-Tree [Guttman 1984], the $R^*$-Tree [Beckmann et al. 1990], the $SKD$-Tree [Ooi et al. 1987; Ooi 1990] and the $P$-Tree [Jagadish 1990c, Schwiets 1993]. In Contrast to Overlapping Regions, Clipping-based schemes such as the $R'$-Tree [Stonebraker et al. 1986; Sellis et al. 1987], do not allow any overlaps between bucket regions; they have to be mutually disjoint, which means that some objects may have to be represented in more than one bucket region. The multiple layer technique can be regarded as a variant of the overlapping regions approach, because data space of the hierarchical layers may overlap, but the data regions within each layer are disjoint. The Multilayer Grid File [Six and Widmayer 1988] and the $R$-File [Hutflesz et al. 1990] are the representative methods in this category.

Besides the spatial access methods we mentioned above, there are several methods that take advantage of more than one technique, such as Hilbert R-Tree [Kamel and Faloutsos 1994], which uses both Overlapping Regions and Space-Filling Curves; and the Filter
CHAPTER 1. INTRODUCTION

Tree [Sevcik and Koudas 1996], which makes a use of a combination of Multiple Layers and Space-Filling Curves.

Due to the complexity of the data, there are many different criteria to address quality and numerous parameters that determine performance. It is almost impossible to expect one access method to outperform all the competitors in every aspect, since every newly proposed method is designed to attack some weakness of the previous ones, possibly at the cost of being inferior with respect to other evaluation criteria.

In this thesis, the terms spatial data, spatial queries and spatial access methods refer to data, queries and access methods of two and three dimensions.

This thesis works with the Filter Tree, testing two new queries, Sphere Intersection Query and Distance Query. In both queries, we will have to calculate the distance from query center to the candidate objects to determine if the objects are inside the query range. As we mentioned previously, the distance functions are usually based on a distance metric for points. However, this is not necessarily always the case. For example, we can define the distance from a point p to a line l as

$$\text{dist} (p, l) = \min_{pl \in l} \text{dist}_p (p, pl),$$

where $\text{dist}_p (p, pl)$ denotes the distance between points p and pl, which is a point on the line. In this case, the consistency of this distance function is guaranteed by the properties of distp, namely, non-negativity and the triangle inequality.

In this thesis, we implement two prototypes based on two different distance functions under Euclidean metrics: the conventional point to point (PTP) and the point to line segment (PTLS) metrics. For the PTLS metric, we try to make a more precise approximation of the line segment. We not only keep the coordinates of the two corners of the MBR around the line segment, but also the slope of the line. This is under the assumption that the real object, such as roads and rivers will be best represented as a line between diagonally opposite corners of the MBR. In order to differentiate the line being either from the upper right corner to the lower left corner, or from the upper left corner to the lower right corner, we use 1 and -1 to indicate the sign of the slope. The point to line segment metric will be based on the distance between the center of the query range and
CHAPTER 1. INTRODUCTION

the line segment. We compare the number of I/Os, the processor time and the result sets from the same input data sets.

1.4 Thesis Organization

The remainder of the thesis is organized as follows:

Chapter 2, Related Work, presents related work in the area of spatial indexing techniques for new data types. We mainly focus on distance queries and other distance related operations on other file structures.

Chapter 3, Problem Statement, describes the problem we are going to address in this thesis. We first compare the two different representations of the spatial data, and then we state our motivation for the Sphere Intersection Query and Distance Query based on the two different representations of the data.

Chapter 4, Experimental Setup, explains the data used in the experiments, the position and the radius used in the Sphere Intersection Query and the Distance Query, the environment under which we conduct our experiments and a discussion over which data structure to choose follows.

Chapter 5, Experimental Results for Sphere Intersection Query, presents the experimental results on number of I/Os and processor time for different datasets. The analysis of the results follows.

Chapter 6, Experimental Results for Distance Query, provides the results for both I/O costs and processor costs for different datasets. At the end of the chapter, we discuss the results that we obtained.

Chapter 7, Conclusions and Future Work, provides a summary of the work done for the thesis and the significance of the results. It also addresses the possible extensions for this research in the future.
Chapter 2

Related Work

2.1 Incremental Algorithms for (k-) Nearest Neighbor Queries

Motivated by their relevance to the field of GIS, pattern recognition, learning theory and image processing, numerous algorithms have been proposed to answer nearest neighbor and $k$-nearest neighbor queries. Among the algorithms, incremental solutions play an important role in this literature [Broder 1990; Henrich 1994; Hjaltason and Samet 1995, 1999]. Priority queues are employed in the incremental algorithms for k-d Tree, LSD-Tree, PMR Quadtree and R-Tree respectively.

2.1.1 The k-d Tree and Broder’s Algorithm

A $k$-d Tree [Bentley 1975] is a binary search tree structure that stores and operates on records by recursively subdividing the universe into $k$-dimensional subspaces using iso-oriented hyperplanes. Each subspace is represented as a node in the tree containing a subset of the collection of records. The position of the hyperplane is chosen so that each of the children contains approximately half of the parent’s records.

Broder’s algorithm [Broder 1990] makes use of the k-d Tree as the index structure to answer the $k$-nearest neighbor query. It stores only the data objects in the priority queue and takes advantage of a stack to keep track of the subtrees of the spatial data structure that need to be processed.

This incremental search algorithm contains two functions: $search-init$ and $search-next$. The function $search-init$ recursively descends the tree to locate the leaf node that contains
the query point. A list of pointers to the non-terminal nodes that were traversed during the descent is then stored in a stack. Then the \textit{search-next} function will use the information in the stack to discover the nearest neighbor to the search point. The distance needs to be computed between the query point and each of the records contained in the leaf node pointed to by the head of the stack. The pointers to the records become elements of a queue, which is ordered by distance to the query center. The record at the head of the queue is a candidate result for the nearest neighbor and the distance serves as the upper bound for the distance to the actual nearest neighbor. All the records within the leaf will be examined to ensure that the first record has the minimum distance to the query point.

After that, the tree is ascended one level by popping out the top element from the stack. New candidate records are determined by recursively descending the other child of the new non-terminal node. This process is repeated until all the elements in the stack have been examined.

2.1.2 The LSD-Tree and Henrich's Algorithm

\textit{LSD-Tree} [Henrich et al. 1989] is a data structure for multi-dimensional points, it divides the data space into pairwise disjoint data cells, namely bucket regions. In each bucket region, a fixed number of objects can be stored. Each time when the threshold for any region is reached, an attempt to insert an additional object will cause a bucket split. For this purpose, a split line is determined through the bucket and the objects are stored in two buckets according to the split line. This process is repeated whenever the capacity of a bucket is exceeded.

The split lines of the LSD-Tree are maintained in a directory, which is a generalized \textit{k-d Tree}. For each split, a new node containing the position and the dimension of the split line is inserted into the directory tree. The leaves of the directory tree reference the buckets in which the actual objects are stored.

To answer the Distance Query, the author employs two priority queues [Henrich 1994], \textit{OPQ} and \textit{NPO}, to store objects and directory nodes or buckets. The elements in the
CHAPTER 2. RELATED WORK

queues are stored in a way that the one with the minimum distance to the query center \( q \) has the highest priority.

The algorithm starts at the root of the directory and searches for the bucket that contains \( q \). During each search, if we check the left child, the right child will be inserted into \( NPQ \), else if we follow the right child, the left child will be inserted into \( NPQ \). Then the objects in the corresponding bucket are inserted into \( OPQ \) and \( \delta_{\text{min}} \) is set to the minimum distance between \( q \) and the closest edge of the searching bucket region. Thereafter the objects with a distance less than or equal to \( \delta_{\text{min}} \) are taken from \( OPQ \). After exhausting all the elements in \( OPQ \), the search region has to be extended by taking the top element from \( NPQ \). If it happens to be a directory node, another round of searching starts. This process repeats until all the elements in \( OPQ \) and \( NPQ \) have been checked.

2.1.3 PMR Quadtree and Hjaltason & Samet's Algorithm

The Quadtree decomposes the universe by means of iso-oriented hyperplanes. As an edge-based variant of PM Quadtrees [Samet and Webber 1985] which deals with line segments, the PMR Quadtree [Nelson and Samet 1986] is based on a regular decomposition of the data space. Each Quadtree block is a square or a hypercube in higher dimensions. Leaf blocks contain the spatial objects (or pointers to them), whereas non-leaf blocks are decomposed into \( 2^k \) sub-blocks, where \( k \) is the number of dimensions.

By making use of a probabilistic splitting rule, PMR Quadtrees differ from other Quadtree variants in the way that object insertions trigger decomposition of Quadtree blocks. The PMR Quadtree is constructed by inserting them one-by-one into an initially empty structure consisting of one block. Each line segment is inserted into all of the blocks that it intersects or occupies in its entirety. During this process, the occupancy of each affected block is checked to see if the insertion causes it to exceed a predetermined splitting threshold. If the splitting threshold is exceeded, the block is split once into four blocks of equal size.
The authors introduced a top-down method [Hjaltason and Samet 1995] to answer the Distance Query. The algorithm will first locate the leaf nodes containing the query object \( q \); then a priority queue is maintained to record the blocks whose descendants have not been visited as well as the objects that have not yet been visited. The crucial point here is that, in addition to using the priority queue for containing blocks, objects are also inserted into the queue as leaf blocks are processed. The key for the priority queue is the distance from the query object to each element, while blocks have higher priority than objects if the distances are exactly the same. A container is examined only if it reaches the head of the queue. The objects at the head of the queue (i.e., with shorter distances) are retrieved until the queue is emptied.

2.1.4 R-Tree Family

R-Trees are an extension of B-Trees for multidimensional objects that are either points or regions. Like B-Trees, R-Trees are balanced and guarantee a space utilization of at least 50%. Due to the popularity of R-Trees, Hjaltason and Samet further extend their incremental \((k-)\) nearest neighbor algorithm to the R-Tree and R*-Tree [Hjaltason and Samet 1999].

2.1.4.1 R-Tree

An R-Tree [Guttman 1984; Greene 1989] corresponds to a hierarchy of nested \( d \)-dimensional intervals. Each node \( v \) of the R-Tree corresponds to a disk page and a \( d \)-dimensional interval \( f^d(v) \). If \( v \) is an interior node then the intervals corresponding to the descendants \( v_i \) of \( v \) are contained in \( f^d(v) \). Intervals at the same tree level may overlap. If \( v \) is a leaf node, \( f^d(v) \) is the \( d \)-dimensional minimum bounding rectangle of the objects stored in \( v \). For each object in turn, \( v \) stores only its MBR and a reference to the complete object description.

Searching in the R-Tree is similar to searching in the B-Tree. At each index node \( v \), all index entries are tested to see whether they intersect the search interval \( I_s \). We should visit all child nodes \( v_i \) with \( f^d(v_i) \cap I_s \neq \emptyset \). Due to the possibility of overlapping regions,
there may be several intervals $\mathcal{I}(v_j)$ that satisfy the search predicate. In the worst case, one may have to visit every index page.

To insert an object $o$, we insert the minimum bounding interval $\mathcal{I}(o)$ and an object reference into the tree. In contrast to searching, we traverse only a single path from the root to the leaf. At each level we choose the child node $v$ whose corresponding interval $\mathcal{I}(v)$ needs the least enlargement to enclose the data object's interval $\mathcal{I}(o)$. If several intervals satisfy this criterion, Guttman proposes selecting the descendant associated with the smallest interval. As a result, we insert the object only once; that is, the object is not dispersed over several buckets. If insertion requires an enlargement of the corresponding bucket region, we adjust it appropriately and propagate the change upwards.

As for deletion, we first perform an exact match query for the object. If we find it in the tree, we delete it. If the bounding interval can be reduced because of the deletion, we perform this adjustment and propagate it upwards.

Many researchers have worked on how to minimize the overlap during insertion. For example, the packed $R$-Tree [Roussopoulos and Leifker 1985] computes an optimal partitioning of the universe and a corresponding minimal $R$-Tree for a given scenario. The Hilbert $R$-Tree [Kamel and Faloutsos 1994] combines the overlapping regions technique with space-filling curves. It first stores the Hilbert value of the data rectangle's centroid in a $B^*$-Tree, then enhances each interior $B^*$-Tree node by the MBR of the subtree below. This facilitates the insertion of new objects considerably. But since the splitting policy takes only the objects' centroids into account, the performance of the structure is likely to deteriorate in the presence of large objects.

2.1.4.2 $R^*$-Tree

$R^*$-Tree [Stonebraker et al. 1986; Sellis et al. 1987] is proposed to overcome the problems associated with overlapping regions in $R$-Tree and packed $R$-Tree. As mentioned, point searches in $R^*$-Tree correspond to single-path tree traversals from the root to one of the leaves. But range searches usually lead to the traversal of multiple paths in both structures.
When inserting a new object $o$, we may have to follow multiple paths, depending on the number of intersections of the MBR $F(o)$ with index intervals. $F(o)$ may be split into $n$ disjoint fragments $I^d_i(o)$ ($\bigcup_{i=1}^n I^d_i(o) = F(o)$). Each fragment is then placed in a different leaf node $v_i$. If there is enough space, the insertion is straightforward. If the bounding interval $F(o)$ overlaps space that has not yet been covered, we have to enlarge the intervals corresponding to one or more leaf nodes. If a leaf node overflows it has to be split. When splitting, for R*-Tree, the splits may propagate not only up the tree, but also down the tree. This may cause further fragmentation of the data intervals. For deletion, R*-Tree first locates all data nodes where fragments of the object are stored and removes them. If storage utilization drops below a given threshold, we try to merge the affected node with its siblings or to reorganize the tree. Though the R*-Tree cannot guarantee minimum space utilization, the authors claim that analytical results indicate that R*-Trees achieve up to 50% savings in disk accesses compared to an R-Tree when searching files of thousands of rectangles.

2.1.4.2 R*-Tree

The R*-Tree [Beckmann et al. 1990] is proposed to overcome the weaknesses of the original algorithms for R-Tree. Since the insertion phase is critical for good search performance, the design of the R*-Tree therefore introduces a policy called forced reinsert: if a node overflows, it is not split right away, instead $p$ entries are removed from the node and reinserted in the tree. According to the authors, the parameter should be about 30% of the maximal number of entries per page. Besides the reinsertion strategy, Beckmann et al. also modified the policy of node-splitting by a plane-sweep paradigm [Preparata and Shamos 1985]. The goals include minimum overlap between bucket regions at the same tree level, minimum region perimeters and maximum storage utilization. The authors claim performance improvements of up to 50% compared to R-Tree.
CHAPTER 2. RELATED WORK

The paper by Berchtold et al. [1996] suggests a modification of the R-Tree called the X-Tree. The X-Tree introduces supernodes, nodes that are larger than the usual block size, to reduce overlap among directory intervals.

2.2 Overview of the Filter Tree

The Filter Tree can be viewed as a file organization that simultaneously uses hierarchical representation, size separation and space-filling curves techniques. By combining these techniques, Filter Trees can perform spatial joins with a guaranteed minimal number of block reads from disk.

In a Filter Tree for a two-dimensional space, the storage of an entity and the access to it is completely based on its MBR. For concreteness, we refer to the two dimensions as x and y for the rest of our discussion, although their interpretation in specific cases will depend on the application. The Minimum Bounding Rectangle is specified by the coordinates of its lower left corner \((x_b, y_l)\) and upper right corner \((x_h, y_h)\), where \(x_b, x_h\) (respectively, \(y_l\) and \(y_h\)) are the smallest and largest values of the \(x\) (respectively \(y\)) coordinate.

Filter Trees achieve the hierarchical representation by a recursive binary partition of the data space in each dimension. The Filter Tree represents the data space in multiple levels, where each level divides the data space into a higher number of cells than the previous level. At level \(j\), the data space is divided into \(4^j\) subsquares by the partition lines at \(k / 2^j, k = 0, 2, 4... 2^j\) in both \(x\) and \(y\) dimensions. The implementation of the Filter Tree studied in this thesis uses up to twenty levels, numbered 0 through 19. Figure 2.1 shows the hierarchical grids for the first 3 levels.

Each entity to be stored in the Filter Tree is associated with the lowest level (largest level number) at which the object can entirely reside within a single cell. If an MBR has one side of length greater than \(2^j\), then it will be associated with a level no lower than \(j\). This way, the relatively large rectangles are guaranteed to be associated with higher levels in the tree, and relatively small rectangles tend to be associated with lower levels. Figure 2.2 illustrates three objects that are stored in the data space. Object A is large and
resides at level 1 of the Filter Tree. Object B is much smaller, and fits within a 1/8 by 1/8 cell, so it is associated with level 3 of the tree. Though object C is even smaller, it happens to straddle grid line $x = \frac{1}{4}$, which forces it to be associated with level 0 of the tree.

![Diagram of Filter Tree Levels](image)

**Figure 2.1** Hierarchical Representation of Data Space: Level 0 to 3

![Diagram of Filter Tree with Objects](image)

**Figure 2.2** Three Objects Are Stored in the Filter Tree
Physical storage of both MBR's and entity records requires a total numeric ordering of the entities. A Hilbert curve is used to obtain this serialized order while retaining locality of overlapping and neighboring entities. The Hilbert curve orders the cells in such a way that cells that are physically close in the data space will have a high likelihood of being located near each other on the Hilbert curve. At all the levels except level 0, the Hilbert curve begins in the lower-left corner and ends in the lower-right corner, and each cell is ordered by its position on the curve. The Hilbert curve visits every cell of the level exactly once, and never crosses itself. Figure 2.3 shows the Hilbert curves for Levels 1 and 2 in two-dimension space. Level 0 has only a single cell, which is given Hilbert value 0. Objects are given the Hilbert value of the cell in which they are contained. For example, in Figure 2.2, object A at Level 1 is associated with Hilbert value 3, object B at level 3 with Hilbert value 23 and object C at level 6 with Hilbert value 0. Hilbert values may also be specified as binary fractions of at least $2^k$ bits, where $k$ is the number of the levels.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{hilbert_curves}
\caption{Hilbert Curves at Levels 0, 1 and 2}
\end{figure}

### 2.2.1 Range Queries

Given a query window specified by its lower left and upper right point coordinates, we expect to retrieve all entities in the tree that overlap the window. At any given level, the query window will cover certain ranges of Hilbert values. This set of ranges of Hilbert values is first calculated at one level of the Filter Tree, known as the containment level. Figure 2.4c illustrates an example of a query window where Level 2 is used as the containment level. In this case, the window covers the Hilbert ranges: 4 to 7.
Once the ranges of Hilbert values that the window covers at the containment level are found, it is simple to calculate the Hilbert values at all other levels. For example, each cell at the containment level covers at most four cells at the level below. These four cells can be found by multiplying the Hilbert value at the containment level by four, and the adding zero, one, two and three. For example, Cell 1 at Level 1 corresponds to Cells 4, 5, 6 and 7 at Level 2. So, in the case above, the ranges of Hilbert values covered by the window is 4 to 7 at Level 2. We can calculate that the window covers a subset of Cells 16 to 31 at Level 3. Figure 2.4d shows us that the window covers 18, 23 to 24 and 27 to 29.

Figure 2.4 Query Window at Level 0, 1, 2 and 3 of the Filter Tree
A similar method can be used to find the cells two levels down. This involves multiplying by 16, then adding 0 to 15 and so forth.

To find the corresponding cells at higher levels, we instead divide by four and then apply the floor function. For example, in the case above, the range 4 to 7 at Level 2 corresponds to Cells 1 at Level 1 and Cell 0 at Level 0. Figure 2.4b and Figure 2.4a illustrate this.

When the ranges of Hilbert values are calculated for each level, the data files of the Filter Tree are processed one level at a time. For each level, the index file is examined to determine which data pages must be read. The required data pages are read and each object is compared with the query window. A buffer is used to store both index and data pages.

2.2.2 Spatial Join Queries

The join operation is one of the most commonly used ways to combine information from two or more relations. Spatial join retrieves all object pairs that satisfy a spatial predicate on objects from two or more datasets.

Spatial join operations are performed in two steps. The first step, namely the filter step; retrieves a list of candidate pairs by applying the query predicate on an approximate representation, such as the MBR. The next step, called the refinement step, tests the full predicate against the actual spatial objects identified in the filter step. The first step is more critical since the purpose is to narrow the search range of the refinement step, in order to reduce the number of entity records that must be read from disk.

A spatial join between two Filter Trees involves an index sweeping process. However, the structure of the Filter Tree makes the sweeping process very efficient. For any pair of data sets, their full spatial join can be computed with the minimal amount of I/O, namely by reading each block of the entity descriptor file exactly once. This is accomplished by sweeping concurrently through the entity descriptor files for level of each participating Filter Tree in increasing Hilbert value order.
Consider the hierarchies of filters, $F_1$ and $F_2$, shown in Figure 2.5. There are four levels in each hierarchy, normalized in the same subspace. If we wish to search for matches between entity descriptors in cell 0 of $F_1$ and all the cells of $F_2$ we may restrict our search of $F_2$ and its enclosing cells at higher levels (in the direction of the arrow in Figure 2.5). No other cells need be considered, since, by the definition of the Filter Tree hierarchy, cells are disjoint. In a similar fashion, matching the descriptors in the 15th cell of level 2 in $F_1$ involves looking at the corresponding cell in $F_2$ and its enclosing cells at level 1 and level 0. There is no need to search lower levels of $F_2$ since all the matches have been reported when the 60th, 61th, 62th and 63th cells of $F_1$ are swept.

Figure 2.5 Spatial Join Example
Chapter 3

Problem Statement

3.1 Motivation

Spatial data is more complicated to handle than one-dimensional data. Consider, for example, in a GIS system, a series of line segments are used to represent roads. We may ask "which road is closest to point $p$?" and "rank the roads by distance from point $p$." For a fixed reference point $p$ and distance metric, a one-dimensional index on the distances of the roads from point $p$ will lead to an efficient execution time for this particular point. However, it would be useless for any other points or distance metrics, and it is too expensive to rebuild indices for every possible query point.

A more interesting and at the same time more complex query would be to identify the closest object to a query point where additional conditions may be imposed on the object. For example, "rank the roads by distance from city $c$ for roads that have a speed limit less than 80 km/hour." One method is to sort all the roads by their distances from query point $c$, and then apply the speed limit condition. This naïve solution turns out to be impractical, since each time we need to re-sort the distance when we answer a similar query with respect to another query point. A less radical solution is to retrieve the closest $k$ road segments and determine if any of them satisfy the speed limit criterion. However, the problem here lies in determining the value of $k$. If $k$ is too small, a failure in finding any road segments that fulfills the speed limit criterion is very likely to happen. If $k$ is too large, a good portion of our work will be wasted since the speed limit of many roads will never be checked.
A reasonable way to overcome the disadvantages indicated above is to obtain the objects incrementally, as they are needed. To our knowledge, the existing incremental solutions to the Distance Query all employ priority queues. However, the cost of priority queue operations could become significant in the incremental distance algorithms when the size of the queue increases. In some cases, if the queue gets too large to fit in memory, its contents must be stored in a disk-based structure instead of memory, making each operation even more costly.

Another related question is, how should we store the road segments? Neither two-dimensional points nor MBRs can represent all the important features of line segments. The direction of the line is lost. And at the same time the accuracy is very important in the distance-based queries. For example, in a GIS system, MBRs are widely used to approximate the real objects and it is very common to have a 1 to 1,000,000 scale. In other words, a deviation of 1 Millimeter in either the MBR approximation or the distance calculation on the map corresponds to 1 Kilometer in real life. This means an error of 1 Millimeter in our calculation may dramatically change the results that are requested by the users who query a range with radius of 100 meters in real life.

We will mainly address the following two problems in our thesis:

1. What is the most efficient technique to perform distance-based queries on spatial data? Is there a way to overcome the drawbacks of a priority queue?

2. In dealing with line segments, what are the efficient approximations for the spatial data to make distance-based queries more accurate and more efficient?

3.2 Criteria for Efficiency

We have repeatedly mentioned the word efficiency in the problem statement. This may refer to one of the two aspects of efficiency: space efficiency and time efficiency.

For space efficiency, the goal is to minimize the number of bytes occupied by the index for the amount of information that is kept. For example, we may want to choose an
MBR to approximate a line segment. The MBR can either be specified by the coordinates of all four corners or be specified by only the coordinates of the lower left corner and the upper right corner. Clearly, the latter representation of the MBR takes only half the space of the former one. However, neither of the representations record the direction of the line. So a representation that contains not only the lower left and the upper right coordinates, but also the sign of the slope, cannot be judged as less efficient though one extra bit is required for each MBR index. For the sake of time efficiency and possible further increase of the precision, we use an extra byte to store the slope in our implementation.

For time efficiency, the situation is not very clear. Elapsed time is obviously what the user cares about. However, the corresponding measurements greatly depend on implementation details, hardware utilization, and other external factors. So another performance measure: the number of disk accesses performed during a search, seems to be more objective in this context. This approach is based on the assumption that most searches are I/O-bound rather than CPU-bound, although this assumption is not always true. The great increases of the size of main memories make I/O accesses less dominant, but it is still an important goal to minimize that number of disk accesses. In our tests, we examine not only the number of I/Os but also the processor time and elapsed time in response to each query.

### 3.3 Sphere Intersection Query and Distance Query

Distance-based queries refer to the queries that can only be answered by calculating the distance between objects and comparing the distance with the query criteria. Common examples of distance-based queries include Nearest-Neighbor Query and Distance Query, which were formally defined in Chapter 1. Here we discuss two specific distance-based queries, Sphere Intersection Queries and Distance Queries. Although these queries can be answered using most spatial data structures, for concreteness we present these two queries in the context of the Filter Tree. Also, performance tests are conducted on the Filter Tree.
3.3.1 Sphere Intersection Queries

A Sphere Intersection Query can be regarded as a variation of the Range Query where the query window is a circle or sphere rather than a rectangle. The circle is specified by a query center and a radius. At the same time, a circumscribed square is determined. Like a Range Query in the Filter Tree, at each containment level, the square will cover a certain range of Hilbert values. Figure 3.1 shows us a query sphere where Level 3 is used as the containment level. At level 3, Hilbert values 18 to 20, 22 to 25 and 29 are intersected by the circumscribed square.

![Figure 3.1 Query Circle at Containment Level 3](image)

As in Range Queries, a square is used as the query window to narrow down the range of Hilbert Values that must be searched. Hence this square determines, at each level, which data pages must be read. Once the data pages are read in, the distance between each object of the page and the query center is calculated and compared with the radius.

After the comparison of the distance, the objects that fulfill the distance criterion will be copied to a page in the Buffer, when the page is full, it is written to the output file. A more thorough description of the Sphere Intersection query, as performed on the Filter Tree, is given in Appendix A.
3.3.2 Distance Queries

The Distance Query is closely related to the Sphere Intersection Query in two-dimensional Euclidean context. It can be divided into two steps. First, we perform a more generalized Sphere Intersection Query. Here more generalized means that we have to compare the distance with two values of the radius, the inner circle radius and the outer circle radius. The result set will include the records having the distance between the inner radius and the outer radius. Figure 3.2 illustrates the distances from the query point O to points a, b and c, which are the centres of MBRs A, B and C. Only b satisfies the distance criteria since only \( \text{dist}(O, O_b) \in [\text{Inner Radius}, \text{Outer Radius}] \). Second, instead of writing all the records that fulfill the distance requirement back to the disk immediately, we need to report the candidate records in an ascending order of distance from the query point.

As discussed in Chapter 3.1, the two naïve ways are not efficient in handling the Distance Query, particularly if additional conditions are imposed on the object. Our approach is that, right after the distance to the query center is calculated, it is compared with the query distance criteria. The top results are inserted into a sorted list, which contains a fixed number of elements in the ascending order of distance. This incremental algorithm will guarantee that effort in calculating the distance will not be wasted. The structure of the Filter Tree makes this step very efficient:

- We do not need to differentiate the leaf nodes and non-leaf nodes, which increases the complexity of the distance calculation for other tree-like structures. Just recall that in the Filter Tree, the entity descriptors are sorted according to the Hilbert value and packed into contiguous blocks of secondary storage. Only entries that specify the ranges of the Hilbert value are kept for each level of the tree. Each time, we simply scan through the blocks that fulfill the Hilbert value requirement. And this way, we will guarantee that each block will be read in at most once.

- We do not need to build the priority queue to store all the nodes that we have to process as required by the incremental algorithm for other tree-like structures. For the Filter Tree, again, we only process the candidate entities within the Hilbert range. While for \textit{R-Tree} [Hjaltason and Samet 1999], for instance, all the leaf nodes that
have intersection with the query range need to be processed, and all the entities contained in the leaf node need to be inserted into the priority queue. In some cases, only a small portion of the entities in the priority queue reside in the query range.

- The performance of our incremental algorithm should be stable regardless of the size of the datasets. However this is not always true for other distance query algorithms. As we pointed out in Chapter 3.1, as the size of the dataset increases, it's very likely that the queue size increases correspondingly, hence each operation becomes more costly.

At the same time, new constraints on the query criteria can be easily enforced. For example, when dealing with extraordinarily large datasets, we can set an upper bound for the number of rows that we have to process, or alternatively, an upper bound for the time each query may take. Of course, these two constraints can be applied at the same time in order to guarantee that the query can be accomplished in a reasonable time.

We will introduce two Distance Query algorithms for the centroid and slope datasets in detail in Appendix B.

---

The border and slope of the MBR

---

The distance between the query center and the centroid of the MBR

Figure 3.2 Examples for Distance Query

24
A typical Distance Query will need to retrieve only a very small fraction of the records that are processed. For example, in a GIS application that contains 50,000 records of line segments, it makes sense to ask “find the nearest 20 roads from city O within 5 kilometers”. Because a general user will be very unlikely to ask for thousands of output records, the user may be interested in the names of the roads with the shortest distance.

3.4 Accuracy in Distance Calculation

Accuracy formally means the degree of conformity of a measure to a standard or a true value. Due to the discrepancy in the approximation of the spatial data and sometimes the large scale between the reality and the model, a seemingly minor error in the approximation can be significant in real life.

This then leads to two sub-problems that need to be addressed in the distance-based query domain.

1. Is there a more accurate approximation of the MBR that can minimize the discrepancy for line segments?

2. Which distance function should be chosen to minimize the error?

3.4.1 Distance for MBRs without Slope

As mentioned in Chapter 3.1, the conventional representation for MBRs, by specifying the lower left and upper right coordinates, loses the slope information. To answer the distance-based queries, we choose to calculate the distance between the query center and the centroid of the MBR. This distance is more appropriate than the distance between the query center and the nearest point of MBRs. This is true because it will have a greater error when the line segment does not intersect the nearest corner of the MBR. And for a randomly distributed dataset, the chance for this mismatch is 50%. Figure 3.3 shows that the distance from query center O to line segment C. Suppose the line passes through the lower right and upper left corners, the distance between O and the upper right corner \((x_h,y_h)\) has the shortest distance, and it is smaller than the radius. However, the true
distance and the distance to $O_c$ are greater than the radius, which causes an erroneous
response to the query. So it is safer and more consistent to use the distance between
centroid and the query center. Though this may also cause some mistakes, it is the best
we can achieve without slope information.

![Diagram](image)

**Figure 3.3 The Distance Between the Query Center and Centroid of the MBR
 Causes Big Discrepancy**

It is reasonable to ask whether a set of line segments have to be stored as MBRs. This
question originates from the fact that, for a set of connected line segments, the upper right
coordinate of the previous segment is the lower left coordinate of the latter segment. It is
repetitive to store the same point twice. As an alternative, we can store the successive
corners of each line segment and no corner need be repeated. It is easy to observe that
this representation takes only half the space that MBR takes up, however, it only works
with connected line segments while the MBR representation is more globally adaptive.

**3.4.1 Distance for MBRs with Slope**

Besides storing the coordinates for the MBRs, we propose another representation that
includes an integer to indicate the sign of the slope of the line segment associated with

26
the MBR. This representation is based on the assumption that the line segment is likely to lie close to the diagonal line of the two corners, and the slope of the line keeps either monotonically increasing or monotonically decreasing in one single MBR. Figure 3.4a shows an example for which the diagonal line $A_1B_1$ closely approximates the real object; Figure 3.4b shows us an example in which the diagonal line is not an accurate representation. To deal with the situation in Figure 3.4b, it might be appropriate to break the curve into two or more segments, and use a single MBR to represent each segment. Figure 3.4c demonstrates the break up scheme.

![Figure 3.4 Examples of Line Segment Representation with Slope Information](image)

---

**Figure 3.4 Examples of Line Segment Representation with Slope Information**

![Figure 3.5 The Distance Between the Query Center to the MBR](image)

---

**Figure 3.5 The Distance Between the Query Center to the MBR**
Chapter 3. Problem Statement

In this case, the distance is no longer simply between two points. Instead, as demonstrated in Figure 3.5, the distance can be either to a corner or to the diagonal line of the MBR. For example, the distance from O to object A is no longer the distance from O to the centroid of object A; more precisely, it should be between O and slope of A. The distance from O to object B is the distance between O and lower left corner of B.
Chapter 4

Experimental Setup

In order to test and compare the performance of the Sphere Intersection Query and the Distance Query on the Filter Tree, we conducted a series of experiments using both synthetic and real data sets. All the experiments were performed on a Sun SPARCstation 5 machine with a total of 64MB memory running Sun Solaris 8 operating system.

4.1 The Questions and Anticipated Results

As mentioned in Section 3.3, we have demonstrated that the structure of the Filter Tree makes the distance-based queries very efficient in theory. We are still interested in the following questions.

1. Are the distance-based queries sensitive to the size of the datasets?

2. What is the effect of the size and position of the query window on the response time?

3. Will the performance be stable when datasets are highly skewed?

4. For a distance query, how does the sorted list size affect the response time?

4.2 Synthetic Data Used

In order to test the performance of our algorithms with different levels of skewed datasets, we set up a set of synthetic data with a diagonal distribution in two-dimensional space. Here diagonal distribution means that the centroids of the entities are distributed along the main diagonal. The levels of skew were used in a manner that the lower
skew number, the more clustered the centroids of the entities are around the main diagonal. The sizes of the entities are created randomly with a uniform distribution between $10^{-6}$ and $10^{-4}$ where the data space is the unit square. To create the three datasets, points were generated randomly and were deleted with higher probability the farther they were from the main diagonal. For each skew number, the probability of being discarded increased with the distance from the main diagonal, but given any distance from the main diagonal, the lower the skew number, the more likely points were to be deleted.

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Number of Points Discarded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagonal1</td>
<td>839</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>396</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>35</td>
</tr>
</tbody>
</table>

Table 4.1 Diagonal Datasets: Number of Points Discarded for Each 100 Points Retained

Figure 4.1 Diagonal Datasets:

a. Diagonal 1

b. Diagonal 2

c. Diagonal3
Table 4.1 shows, for each of the three skews used, the number of points discarded to generate a set of one hundred random points. The files of this set are referred to as Diagonal1, Diagonal2 and Diagonal3. These files were created with the following sizes: 10,000, 25,000, 50,000 and 100,000 because the sizes are similar to the sizes of the real datasets used.

In Diagonal1, with the highest skews, almost all points are quite close to main diagonal. In Diagonal2, most of data space is occupied, but is denser around the main diagonal. In Diagonal3, most of data space is occupied, and only the upper-left corner and lower-right corner are sparse. This is very close to a uniform distribution. The distributions of three Diagonal sets are shown in Figure 4.1.

4.3 Real Data Used

The real data that is used in the experiments consists of five data sets, referred to as EC, MA, OR, GT and CT respectively. Each dataset contains the longitudes and latitudes of the objects, in our case, line segments.

EC represents the borders of countries on the European Continent. It consists of 25,959 rows, and each row describes a line segment. These line segments form 337 disjointed regions.

MA and OR are derived from the TIGER/Line File [Bureau of the Census 2000]. MA contains 66,908 line segments for Worcester, Massachusetts. OR contains 69,228 line segments for Lane County, Oregon.

GT and CT are the geographical data sets for Great Toronto Area and City of Toronto. GT has 4,632 line segments and makes up 199 regions. Compared to GT, CT is much more detailed, consisting of 96,844 line segments that make up 5370 regions in Toronto City.
a. EC (Europe Map)

b. MA (Worcester, MA)
c. OR (Lane, OR)

d. GT (Greater Toronto Area)
Due to the diversity of the formats of our datasets in representing the longitudes and latitudes of the real objects, it is essential to normalize the data to $[0,1]$ interval for both $x$ and $y$ axes, so that they can be used consistently in our experiments.

In order to normalize the datasets, we first search for the smallest and the largest coordinates for both $X$ and $Y$ axes, letting $X_s$ and $X_b$ refer to the smallest and biggest value respectively. ($Y_s$ and $Y_b$ are the corresponding values for $Y$ axis.) Then we transform all the points to $[0, X_b-X_s]$, $[0, Y_b-Y_s]$ area and scale them down to $[0,1]$, $[0,1]$ area. (The transformation and scaling can be achieved in one single step.)

It is straightforward to determine the slope of the line segment by comparing both $X$ and $Y$ coordinates of the two corners of the MBR from the raw data. If the slope is negative, in other words, the upper left and lower right coordinates are given, we need to retain the lower left and upper right coordinates for the MBR. However, the uniqueness of the line representation is maintained since the slope has been recorded.
Thus, the sizes of the real datasets vary from 4,632 rows to 96,844 rows. This enables us to conduct our experiments on a wide range of sizes, therefore providing a sound basis to examine the sensitivity of our algorithms to the size of the datasets. Plots of the five real datasets are shown in Figure 4.2.

We conducted our experiments on all the five real datasets, and the results are consistent. However, in order to make the presentation more concise, we randomly choose only three result sets out of five to discuss in each experiment.

4.4 Query Windows Used

In order to test the effect of the size and position of the query window on the response time, we set up nine query windows. Table 4.2 illustrates the coordinate of the query center, the radius of the query and the fraction of the data space each query area covers.

<table>
<thead>
<tr>
<th>Windows</th>
<th>Center Coordinate $(x, y)$</th>
<th>Radius</th>
<th>Fraction of Data Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>(0.5, 0.5)</td>
<td>0.05</td>
<td>0.79%</td>
</tr>
<tr>
<td>Q2</td>
<td>(0.5, 0.5)</td>
<td>0.10</td>
<td>3.14%</td>
</tr>
<tr>
<td>Q3</td>
<td>(0.5, 0.5)</td>
<td>0.15</td>
<td>7.07%</td>
</tr>
<tr>
<td>Q4</td>
<td>(0.5, 0.5)</td>
<td>0.20</td>
<td>12.57%</td>
</tr>
<tr>
<td>Q5</td>
<td>(0.5, 0.5)</td>
<td>0.25</td>
<td>19.93%</td>
</tr>
<tr>
<td>Q6</td>
<td>(0.25, 0.25)</td>
<td>0.15</td>
<td>7.07%</td>
</tr>
<tr>
<td>Q7</td>
<td>(0.25, 0.75)</td>
<td>0.15</td>
<td>7.07%</td>
</tr>
<tr>
<td>Q8</td>
<td>(0.75, 0.25)</td>
<td>0.15</td>
<td>7.07%</td>
</tr>
<tr>
<td>Q9</td>
<td>(0.75, 0.75)</td>
<td>0.15</td>
<td>7.07%</td>
</tr>
</tbody>
</table>

Table 4.2 Query Windows

The first five query centers are located at the centroid of the data space with the radii of 0.05, 0.10, 0.15, 0.20 and 0.25 respectively. The last four query centers reside at the centroids of the four sub-squares of level 1 of the Filter Tree. Here we choose the radius to be 0.15, which is the median value for the first five query windows. Figure 4.3 illustrates all these nine windows.

As we can see in Figure 4.3, the nine query windows have covered an essential part of the data space. Ranging from 0.79% to 19.93% of the space, Q1 to Q5 are used to test the
effect of different query window size on the response time. Q6 to Q9 are intended to examine whether the distance-based query on the Filter Tree will be affected by the position of the query window. We believe that these nine query windows are representative enough for our experimental purposes.

Figure 4.3 Query Windows
Chapter 5

Experimental Results: Sphere Intersection Query

We conduct our experiments for Sphere Intersection Query (SIQ) using two different distance metrics: point to point (PTP) and point to line segment (PTLS) metrics. For each of these two distance metrics, we test our algorithms on both synthetic datasets and real datasets. We record the number of I/Os and the processor time, and compare the costs for these two implementations. The purpose of these two experiments is to:

1. Determine whether the performance of our algorithms is stable over a range of test conditions;

2. Determine how the result sets of the two SIQ algorithms differ;

3. Determine whether the slope implementation is substantially more expensive than the centroid one.

Experiments 1 and 2 test the SIQ Centroid and SIQ Slope algorithms respectively on varying sizes of the synthetic datasets. Experiments 3 and 4 examine the effect of varying the query window size on the response time. Experiments 5 and 6 test if the SIQ algorithms are sensitive to the position of the query window.

5.1 Experiments 1 and 2: Varying the Size of the Dataset

In Experiments 1 and 2, our tests on the SIQ operation were conducted on the Diagonal datasets. The sizes of the Datasets vary from 10,000 to 100,000. The medium sized query window Q3 is used.
Table 5.1 gives the number of Buffer References and Buffer Reads when performing a Sphere Intersection Query. Buffer References measure how many times one data page is

<table>
<thead>
<tr>
<th>File</th>
<th>Size</th>
<th>Buffer References</th>
<th>Buffer Reads</th>
<th>Reads per Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagonal1</td>
<td>10,000</td>
<td>143</td>
<td>83</td>
<td>0.58</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>25,000</td>
<td>246</td>
<td>186</td>
<td>0.76</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>50,000</td>
<td>418</td>
<td>348</td>
<td>0.83</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>100,000</td>
<td>748</td>
<td>672</td>
<td>0.90</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>10,000</td>
<td>132</td>
<td>66</td>
<td>0.50</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>25,000</td>
<td>246</td>
<td>176</td>
<td>0.72</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>50,000</td>
<td>395</td>
<td>324</td>
<td>0.82</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>100,000</td>
<td>675</td>
<td>616</td>
<td>0.91</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>10,000</td>
<td>119</td>
<td>37</td>
<td>0.31</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>25,000</td>
<td>174</td>
<td>107</td>
<td>0.61</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>50,000</td>
<td>273</td>
<td>206</td>
<td>0.75</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>100,000</td>
<td>435</td>
<td>385</td>
<td>0.89</td>
</tr>
</tbody>
</table>

Table 5.1 Buffer References and Buffer Reads for SIQ

**Figure 5.1 Buffer Reads per Reference**
searched for some range of Hilbert values; Buffer Reads count the number of times a logical read is performed to read the pages into the buffer. Table 5.1 shows that both Buffer References and Buffer Reads increase as the size of the datasets increases, the Buffer Reads tend to increase at a faster rate than Buffer References. Figure 5.1 illustrates this.

We observe that the above results are consistent with our expectation. The Filter Tree we implemented makes use of a buffer of size 64 pages, where index pages are kept in memory with higher priority than data pages. Normally, the index pages will take approximately half the buffer space and will reside there until the end of each operation. The rest of the room in the buffer will be used to accommodate the data pages when they are required in the Hilbert Value search. If the data page is still in the buffer when it is requested again, this page is referenced more than once. However, the chance for one page to be re-referenced tends to decrease when the dataset becomes very big, since that page is more likely to have been replaced in memory by other data pages. This explains why in our tests, only approximately 10% of the pages are referenced again when the dataset contains 100,000 records.

<table>
<thead>
<tr>
<th>File</th>
<th>Size</th>
<th>Total Time (ms)</th>
<th># Rows Output</th>
<th># Rows per Millisecond</th>
<th># Pages Output</th>
<th>Time per Output Page (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagonal1</td>
<td>10,000</td>
<td>286</td>
<td>1617</td>
<td>5.7</td>
<td>10</td>
<td>28.6</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>25,000</td>
<td>791</td>
<td>5074</td>
<td>6.4</td>
<td>30</td>
<td>26.4</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>50,000</td>
<td>1600</td>
<td>10486</td>
<td>6.6</td>
<td>62</td>
<td>25.8</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>100,000</td>
<td>3185</td>
<td>21180</td>
<td>6.6</td>
<td>125</td>
<td>25.5</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>10,000</td>
<td>247</td>
<td>1435</td>
<td>5.8</td>
<td>9</td>
<td>27.4</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>25,000</td>
<td>734</td>
<td>4612</td>
<td>6.3</td>
<td>28</td>
<td>26.2</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>50,000</td>
<td>1499</td>
<td>9799</td>
<td>6.5</td>
<td>58</td>
<td>25.8</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>100,000</td>
<td>3010</td>
<td>20002</td>
<td>6.6</td>
<td>118</td>
<td>25.5</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>10,000</td>
<td>108</td>
<td>481</td>
<td>4.5</td>
<td>3</td>
<td>36.0</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>25,000</td>
<td>348</td>
<td>1950</td>
<td>5.6</td>
<td>12</td>
<td>29.0</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>50,000</td>
<td>745</td>
<td>4490</td>
<td>6.0</td>
<td>27</td>
<td>27.6</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>100,000</td>
<td>1517</td>
<td>9528</td>
<td>6.3</td>
<td>57</td>
<td>26.6</td>
</tr>
</tbody>
</table>

Table 5.2 Varying Size of the Dataset for SIQ (Centroid)
CHAPTER 5. EXPERIMENTAL RESULTS: SPHERE INTERSECTION QUERY

<table>
<thead>
<tr>
<th>File</th>
<th>Size</th>
<th>Total Time (ms)</th>
<th># Rows Output</th>
<th># Rows per Millisecond</th>
<th># Pages Output</th>
<th>Time per Output Page (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagonal1</td>
<td>10,000</td>
<td>305</td>
<td>1617</td>
<td>5.3</td>
<td>11</td>
<td>27.7</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>25,000</td>
<td>859</td>
<td>5074</td>
<td>5.9</td>
<td>34</td>
<td>25.3</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>50,000</td>
<td>1738</td>
<td>10486</td>
<td>6.0</td>
<td>70</td>
<td>24.8</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>100,000</td>
<td>3485</td>
<td>21181</td>
<td>6.1</td>
<td>142</td>
<td>24.5</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>10,000</td>
<td>259</td>
<td>1435</td>
<td>5.5</td>
<td>10</td>
<td>25.9</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>25,000</td>
<td>779</td>
<td>4612</td>
<td>5.9</td>
<td>31</td>
<td>25.1</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>50,000</td>
<td>1627</td>
<td>9799</td>
<td>6.0</td>
<td>66</td>
<td>24.7</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>100,000</td>
<td>3270</td>
<td>20002</td>
<td>6.1</td>
<td>134</td>
<td>24.4</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>10,000</td>
<td>115</td>
<td>489</td>
<td>4.3</td>
<td>4</td>
<td>28.8</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>25,000</td>
<td>375</td>
<td>1950</td>
<td>5.2</td>
<td>13</td>
<td>28.8</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>50,000</td>
<td>808</td>
<td>4490</td>
<td>5.6</td>
<td>30</td>
<td>26.9</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>100,000</td>
<td>1646</td>
<td>9528</td>
<td>5.8</td>
<td>64</td>
<td>25.7</td>
</tr>
</tbody>
</table>

Table 5.3 Varying Size of the Dataset for SIQ (Slope)

![Graph](image)

a. Average Time per Output Page for SIQ Centroid
b. Average Number of Objects Output per Millisecond for SIQ Centroid

c. Average Time per Output Page for SIQ Slope
d. Average Number of Objects Output per Millisecond for SIQ Slope

**Figure 5.2 Varying the Size of the Datasets**

Figure 5.2 is derived from Table 5.2 and Table 5.3, it shows the effects of varying size of the datasets on SIQ operation. Figure 5.2a and Figure 5.2c demonstrate that for both centroid and slope implementation, as the size of the input file increases, the average time spent on each output page tends to decrease. Accordingly, Figure 5.2b and Figure 5.2d show that the average number of rows produced per millisecond increases.

We observe that the Total Time (TT) can be divided into two parts, namely Overhead Time (OHT) and Output Time (OPT). We define the following formula:

\[ TT = OHT + OPT; \]

The above formula can also be expressed in the following form:

\[ TT = OHT + AOPT \times NOP; \]

where AOPT denotes Average Output Time per page and NOP denotes Number of Output Pages. We expect that the OHT and AOPT to be nearly constant for each query range if the NOP is in a reasonable range. Here reasonable means no less than three pages, since if the result set contains only one or two pages, and if the page contains only a small portion
of the potential capacity of a data page, then the average time per page becomes meaningless.

Figure 5.3 Average Output Time and Overhead Time for SIQ
Based on the results we obtained from Table 5.2 and Table 5.3, we derive two linear fits that interpolate the AOHT and OHT for Diagonal1 dataset. As shown in Figure 5.3a and Figure 5.3b, the AOPT and OHT are almost constant. This helps to explain why the average time per output page will decrease in Figure 5.2a and Figure 5.2c: suppose that OHT is a constant, so the larger the number of output pages, the smaller that page’s share of the overhead is. However, this effect becomes weaker and weaker as the number of output pages grows. That’s why in Figure 5.2a and Figure 5.2c the decreasing rate in average time per output page slows down as the output size increases. This property will make our estimation for the execution time more precise.

<table>
<thead>
<tr>
<th>File</th>
<th>Size</th>
<th>SIQ Slope Total Time (ms)</th>
<th>SIQ Centroid Total Time (ms)</th>
<th>(Time Difference) / (SIQ Slope Total Time) %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagonal1</td>
<td>10,000</td>
<td>305</td>
<td>281</td>
<td>7.9%</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>25,000</td>
<td>859</td>
<td>791</td>
<td>7.9%</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>50,000</td>
<td>1738</td>
<td>1600</td>
<td>7.9%</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>100,000</td>
<td>3485</td>
<td>3185</td>
<td>8.6%</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>10,000</td>
<td>259</td>
<td>247</td>
<td>4.6%</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>25,000</td>
<td>779</td>
<td>734</td>
<td>5.8%</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>50,000</td>
<td>1627</td>
<td>1499</td>
<td>7.9%</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>100,000</td>
<td>3270</td>
<td>3010</td>
<td>8.0%</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>10,000</td>
<td>115</td>
<td>108</td>
<td>6.1%</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>25,000</td>
<td>375</td>
<td>348</td>
<td>7.2%</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>50,000</td>
<td>808</td>
<td>745</td>
<td>7.8%</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>100,000</td>
<td>1646</td>
<td>1517</td>
<td>7.8%</td>
</tr>
</tbody>
</table>

**Table 5.4 Comparison of the Centroid and Slope Results**

Table 5.4 shows the ratio of the Time Difference, RTD, (SIQ Slope Total Time – SIQ Centroid Total Time) to the time of SIQ Slope Total Time. It indicates that SIQ Slope is always 4.6% to 8.6% more expensive than SIQ Centroid. This is because SIQ Slope is more complex when calculating distance. And it also indicates that, as the input dataset becomes larger, the relative expense for SIQ Slope is non-decreasing.
5.2 Experiments 3 and 4: Varying the Query Window Size

In Experiments 3 and 4, we study the effect of varying the query window size on the response time for both synthetic datasets and real datasets. For synthetic data, we choose Diagonal datasets with 50,000 rows; for real data, GT, CT and EC datasets are used in these two experiments. We choose five Query Windows Q1, Q2, Q3, Q4 and Q5, with the radius ranging from 0.05 to 0.25, and the areas they cover ranging from 0.79% to 19.93% of the unit square.

Figure 5.5a and Figure 5.5c show that the Average Time per Output Page for SIQ Centroid on both synthetic datasets and real datasets tends to decrease as the query window expands. Figure 5.5b and Figure 5.5d illustrate the trend for Average Number of records per Millisecond for SIQ Centroid as the query window increases.

a. Average Time per Output Page for SIQ Centroid Diagonal
b. Average Number of Objects Output per Millisecond for SIQ Centroid Diagonal

c. Average Time per Output Page for SIQ Centroid (Real Datasets)
d. Average Number of Objects Output per Millisecond for SIQ Centroid (Real Datasets)

**Figure 5.5 Varying the Query Window Size for SIQ Centroid**

a. Average Time per Output Page for SIQ Slope (Diagonal)
b. Average Number of Objects Output per Millisecond for SIQ Slope (Diagonal)

c. Average Time per Output Page for SIQ Slope (Real Datasets)
d. Average Number of Objects Output per Millisecond for SIQ Slope (Real Datasets)

Figure 5.6 Varying the Query Window Size for SIQ Slope

Figure 5.6 shows the corresponding experimental results for SIQ Slope. We observe that the results are consistent between SIQ Centroid and SIQ Slope. Figure 5.5 and Figure 5.6 also indicate that SIQ Centroid and SIQ Slope algorithms are stable on both synthetic and real datasets. Furthermore, as in Experiments 1 and 2, as the output size increases, the Average Time per Output Page declines. We take this as an indication that the SIQ algorithm is not sensitive to the size of the query windows; the output size is the primary determinant of response time.

Table 5.5 illustrates a comparison of relative time difference and relative output difference between SIQ Slope and SIQ Centroid for real datasets. ROD stands for the Ratio of Objects Output Number Difference, which is equal to (SIQ Slope Objects Output Number – SIQ Centroid Objects Output Number) to SIQ Slope Objects Output Number.
Table 5.5 Time Difference and Output Difference between SIQ Slope and SIQ Centroid for Varying Size of Query Window

Consistent with our results shown in Table 5.4, the percentage of RTD varies between 4.3% and 8.6%. Except for one extreme case, ROD is substantially smaller compared to RTD. This is because the real data we used is very detailed. In other words, the difference between the two distance metrics is not very remarkable for these datasets.

One more observation regarding the output number difference is that, the number of records produced by the Slope implementation is never less than that of the Centroid implementation. Recall that, when we know the slope, we measure to the closest point on the inferred diagonal line. That distance is guaranteed to be less than or equal to the distance to the centroid, with equality occurring only if the inferred line happens to be perpendicular to the line from the query point to the centroid.

5.3 Experiments 5 and 6: Varying the Query Window Position

The purpose of these two experiments is to check the effect of varying the query window position on the response time. For synthetic datasets, we still make use of Diagonal Datasets with 50,000 line segments; for real datasets, we choose MA, OR and EC. Four
a. Average Time per Output Page for SIQ Centroid (Diagonal)

b. Average Time per Output Page for SIQ Slope (Diagonal)

Figure 5.7 Varying Position of the Query Window
query windows; Q6, Q7, Q8 and Q9 are used in these tests. They are of the same radius and each of these query windows covers 7.07% of the whole area.

Figure 5.7 shows the Average Time per Output Page for varying positions of the Query Window for both SIQ Centroid and SIQ Slope implementations. They indicate that for each query window, the larger the number of output pages the smaller the Average Time per Output Page. Once again this is consistent with our previous tests.

<table>
<thead>
<tr>
<th>File</th>
<th>Query Window</th>
<th>Total Time (ms)</th>
<th># Rows Output</th>
<th># Rows per Millisecond</th>
<th># Pages Output</th>
<th>Time per Output Page (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MA</td>
<td>Q6</td>
<td>2110</td>
<td>12890</td>
<td>6.1</td>
<td>76</td>
<td>27.8</td>
</tr>
<tr>
<td>MA</td>
<td>Q7</td>
<td>88</td>
<td>84</td>
<td>1.0</td>
<td>1</td>
<td>n.a.</td>
</tr>
<tr>
<td>MA</td>
<td>Q8</td>
<td>1207</td>
<td>7233</td>
<td>6.0</td>
<td>43</td>
<td>28.1</td>
</tr>
<tr>
<td>MA</td>
<td>Q9</td>
<td>368</td>
<td>1821</td>
<td>5.0</td>
<td>11</td>
<td>33.5</td>
</tr>
<tr>
<td>OR</td>
<td>Q6</td>
<td>698</td>
<td>3977</td>
<td>5.7</td>
<td>24</td>
<td>29.1</td>
</tr>
<tr>
<td>OR</td>
<td>Q7</td>
<td>536</td>
<td>2971</td>
<td>5.5</td>
<td>18</td>
<td>29.8</td>
</tr>
<tr>
<td>OR</td>
<td>Q8</td>
<td>66</td>
<td>0</td>
<td>n.a.</td>
<td>0</td>
<td>n.a.</td>
</tr>
<tr>
<td>OR</td>
<td>Q9</td>
<td>1595</td>
<td>9423</td>
<td>6.0</td>
<td>56</td>
<td>28.5</td>
</tr>
<tr>
<td>EC</td>
<td>Q6</td>
<td>98</td>
<td>241</td>
<td>2.5</td>
<td>2</td>
<td>n.a.</td>
</tr>
<tr>
<td>EC</td>
<td>Q7</td>
<td>256</td>
<td>1363</td>
<td>5.3</td>
<td>9</td>
<td>28.4</td>
</tr>
<tr>
<td>EC</td>
<td>Q8</td>
<td>335</td>
<td>1633</td>
<td>4.9</td>
<td>10</td>
<td>33.5</td>
</tr>
<tr>
<td>EC</td>
<td>Q9</td>
<td>515</td>
<td>3146</td>
<td>6.1</td>
<td>19</td>
<td>27.1</td>
</tr>
</tbody>
</table>

Table 5.6 Varying Query Window Position for SIQ (Centroid)

<table>
<thead>
<tr>
<th>File</th>
<th>Query Window</th>
<th>Total Time (ms)</th>
<th># Rows Output</th>
<th># Rows per Millisecond</th>
<th># Pages Output</th>
<th>Time per Output Page (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MA</td>
<td>Q6</td>
<td>2310</td>
<td>13030</td>
<td>5.6</td>
<td>87</td>
<td>26.6</td>
</tr>
<tr>
<td>MA</td>
<td>Q7</td>
<td>97</td>
<td>94</td>
<td>1.0</td>
<td>1</td>
<td>n.a.</td>
</tr>
<tr>
<td>MA</td>
<td>Q8</td>
<td>1319</td>
<td>7308</td>
<td>5.5</td>
<td>49</td>
<td>26.9</td>
</tr>
<tr>
<td>MA</td>
<td>Q9</td>
<td>403</td>
<td>1856</td>
<td>4.6</td>
<td>13</td>
<td>31</td>
</tr>
<tr>
<td>OR</td>
<td>Q6</td>
<td>754</td>
<td>4048</td>
<td>5.4</td>
<td>27</td>
<td>27.9</td>
</tr>
<tr>
<td>OR</td>
<td>Q7</td>
<td>582</td>
<td>3014</td>
<td>5.2</td>
<td>21</td>
<td>27.7</td>
</tr>
<tr>
<td>OR</td>
<td>Q8</td>
<td>69</td>
<td>0</td>
<td>n.a.</td>
<td>0</td>
<td>n.a.</td>
</tr>
<tr>
<td>OR</td>
<td>Q9</td>
<td>1755</td>
<td>9536</td>
<td>5.4</td>
<td>64</td>
<td>27.4</td>
</tr>
<tr>
<td>EC</td>
<td>Q6</td>
<td>105</td>
<td>243</td>
<td>2.3</td>
<td>2</td>
<td>n.a.</td>
</tr>
<tr>
<td>EC</td>
<td>Q7</td>
<td>272</td>
<td>1363</td>
<td>5.0</td>
<td>10</td>
<td>27.2</td>
</tr>
<tr>
<td>EC</td>
<td>Q8</td>
<td>371</td>
<td>1642</td>
<td>4.4</td>
<td>11</td>
<td>33.7</td>
</tr>
<tr>
<td>EC</td>
<td>Q9</td>
<td>554</td>
<td>3150</td>
<td>5.7</td>
<td>21</td>
<td>26.4</td>
</tr>
</tbody>
</table>

Table 5.7 Varying Query Window Position for SIQ (Slope)
CHAPTER 5. EXPERIMENTAL RESULTS: SPHERE INTERSECTION QUERY

One more observation on Figure 5.7 is that, by symmetry, we would expect the lines for Q6 and Q9, Q7 and Q8 to be identical respectively, however due to the randomness of the synthetic datasets, there is a small discrepancy between the corresponding results. Since Q7 and Q8 are placed at the sparse corners of the data space, the Number of Output (NOP) does not fall in a reasonable range for Diagonal 1 and Diagonal 2, so for Q7 and Q8, only the data for Diagonal 3 is available.

Similar conclusions can be drawn from Table 5.6 and Table 5.7, which contains results based on real datasets.

5.4 Experiments Summary

We have the following conclusions from the experiments that we conducted.

1. The SIQ algorithms on the Filter Tree preserve a stable performance regardless of the Input Data Size, the Query Window Size and the Query Position.
   - With a sufficient number of output pages (≥3), the size of the output seems to be the primary determinant of the response time. We can rather precisely estimate the response time if we know the output size.
   - The SIQ Centroid and SIQ Slope implementations share similar behavior on both synthetic and real datasets.

2. The SIQ Slope is always 4% to 9% more expensive than SIQ Centroid in the term of time efficiency. While due to the high quality of the real data, the relative output difference (ROD) is only approximately 1%. Note that the number of objects in the result set is usually on the order of thousand. This makes a difference when the accuracy of the query is assessed.
Chapter 6

Experimental Results: Distance Query

The Distance Query (DQ) algorithm we proposed on the Filter Tree employs sorted list to store the output in order of ascending distance. The algorithm can be divided into two steps:

1. Fill up the output list without sorting it. We quick-sort the list for the first time either when the list is full or the total number of output records is smaller than the list size. After sorting, the algorithm will terminate if the latter case is true; otherwise the distance of the last record will be set up as the distance criterion.

2. If the candidate record has smaller distance than the distance criterion, it will replace the last element in the list, and then a bubble sort after the replacement will put the list in order again by simply moving the last element upward to its appropriate position in the list.

Similar to our experiments in Section 5, we conduct our tests for Distance Query on PTP and PTLS metrics using both synthetic and real data. The number of I/Os, the number of records processed, the number of sorts and the processor time are recorded to evaluate the performance of the two implementations.

Experiments 1 and 2 examine the DQ Centroid and DQ Slope algorithms on varying sizes of the synthetic data. Experiments 3 and 4 test the effect of varying the query window size on the response time. Besides testing stability and time efficiency of the DQ algorithms, Experiment 5 and 6 are designed especially to examine the effect of varying the sorted list size to the response time.
6.1 Experiments 1 and 2: Varying the Size of the Dataset

In Experiment 1 and 2, we make use of Diagonal datasets to test our DQ algorithm. The sizes of the Datasets vary from 10,000 to 50,000. We choose the medium sized Q3 as the query window and 50 (elements) as the default sorted list size.

<table>
<thead>
<tr>
<th>File</th>
<th>Size</th>
<th>Buffer References</th>
<th>Buffer Reads</th>
<th>Reads per Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagonal1</td>
<td>10,000</td>
<td>167</td>
<td>104</td>
<td>0.62</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>25,000</td>
<td>307</td>
<td>243</td>
<td>0.79</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>50,000</td>
<td>531</td>
<td>463</td>
<td>0.87</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>10,000</td>
<td>151</td>
<td>88</td>
<td>0.58</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>25,000</td>
<td>293</td>
<td>228</td>
<td>0.78</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>50,000</td>
<td>492</td>
<td>431</td>
<td>0.88</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>10,000</td>
<td>138</td>
<td>54</td>
<td>0.39</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>25,000</td>
<td>209</td>
<td>141</td>
<td>0.67</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>50,000</td>
<td>325</td>
<td>271</td>
<td>0.83</td>
</tr>
</tbody>
</table>

Table 6.1 Buffer References and Buffer Reads for DQ

Table 6.1 shows the number of Buffer References and Buffer Reads for the Distance Query algorithm. Perfectly consistent with Sphere Intersection Query, DQ sustains the trend that as the data size increases, both Buffer References and Buffer Reads increase. Table 6.1 also indicates that Buffer Reads increase at a relatively higher rate compared to Buffer References.

<table>
<thead>
<tr>
<th>File</th>
<th>Size</th>
<th>DQS # Sorts</th>
<th>DQS Total Time (ms)</th>
<th>DQC Total Time (ms)</th>
<th>(Time Difference) / (DQS Total Time)%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagonal1</td>
<td>10,000</td>
<td>189</td>
<td>215</td>
<td>193</td>
<td>10.2%</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>25,000</td>
<td>235</td>
<td>392</td>
<td>333</td>
<td>15.1%</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>50,000</td>
<td>254</td>
<td>651</td>
<td>532</td>
<td>18.3%</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>10,000</td>
<td>193</td>
<td>197</td>
<td>181</td>
<td>8.1%</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>25,000</td>
<td>248</td>
<td>367</td>
<td>317</td>
<td>13.6%</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>50,000</td>
<td>284</td>
<td>605</td>
<td>503</td>
<td>16.9%</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>10,000</td>
<td>134</td>
<td>135</td>
<td>129</td>
<td>4.4%</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>25,000</td>
<td>203</td>
<td>257</td>
<td>233</td>
<td>9.3%</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>50,000</td>
<td>277</td>
<td>435</td>
<td>378</td>
<td>13.1%</td>
</tr>
</tbody>
</table>

Table 6.2 Relative Time Difference for DQ Slope & DQ Centroid on Varying Dataset Sizes
Table 6.2 shows that as dataset size increases, the Number of Sorts, the Total Response Time for both DQ Slope and DQ Centroid keep increasing. Furthermore, as illustrated in Figure 6.2, the Relative Time Difference (RTD) grows for the increasing size of data.

Figure 6.1 Relative Time Difference between DQ Slope and DQ Centroid on Varying Dataset Sizes

We observe that the Total Time (TT) can be divided into three parts, namely Overhead time (OHT), Distance Calculating Time (DCT) and List Sorting Time (LST).

\[ TT = OHT + DCT + LST; \]

We expect OHT to be a constant for each query range; DCT linearly depends on the number of rows being processed; LST depends on the number of sorts in each query. In other words, the above formula can be expressed in the following format:

\[ TT = OHT + DCTPR \times NRP + LST; \]

where DCTPR denotes Distance Calculating Time per Record and NRP denotes Number of Rows Processed.

Since the OHT should be identical for the same query range and the number of sorts is very similar between DQ Slope and DQ Centroid on Diagonal datasets, we expect the
time difference between DQ Slope and DQ Centroid only correlates to the number of rows processed. This expectation is confirmed by observing that the ratio of Time Difference to Number of Rows Processed is almost constant in Table 6.3.

<table>
<thead>
<tr>
<th>File</th>
<th>Size</th>
<th># Rows Processed</th>
<th>DQS Total Time (ms)</th>
<th>DQC Total Time (ms)</th>
<th>(Time Difference) / (# Rows Processed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagonal1</td>
<td>10,000</td>
<td>2337</td>
<td>215</td>
<td>193</td>
<td>9.4e-3</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>25,000</td>
<td>6664</td>
<td>392</td>
<td>333</td>
<td>8.9e-3</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>50,000</td>
<td>14009</td>
<td>656</td>
<td>532</td>
<td>8.9e-3</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>10,000</td>
<td>1831</td>
<td>197</td>
<td>181</td>
<td>8.7e-3</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>25,000</td>
<td>5844</td>
<td>367</td>
<td>317</td>
<td>8.6e-3</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>50,000</td>
<td>12431</td>
<td>605</td>
<td>503</td>
<td>8.2e-3</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>10,000</td>
<td>720</td>
<td>135</td>
<td>129</td>
<td>8.3e-3</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>25,000</td>
<td>2727</td>
<td>257</td>
<td>233</td>
<td>8.8e-3</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>50,000</td>
<td>6169</td>
<td>435</td>
<td>378</td>
<td>9.2e-3</td>
</tr>
</tbody>
</table>

Table 6.3 Time Difference vs. Number of Rows Processed for DQ Slope and DQ Centroid on Varying Dataset Sizes

Moreover, Table 6.3 indicates for each specific query window, the number of rows processed is the primary determinant of response time for varying dataset sizes.

6.2 Experiments 3 and 4: Varying the Query Window Size

In Experiments 3 and 4, we study the effect of response time on our DQ Slope and DQ Centroid algorithms. Five Query Windows Q1, Q2, Q3, Q4 and Q5 are used to test both synthetic data and real data. For synthetic data, we choose three Diagonal datasets with 50,000 rows; for real datasets, we make use of EC, MA and CT datasets.

In Table 6.4, once again we find the ratio of Time Difference to the Number of Rows Processed very stable for both synthetic data and real data. This ratio further confirms that the number of rows being processed is the dominant factor in determining response time. The Time Difference mainly lies in the difference of complexity in two different distance functions.

As illustrated in Figure 6.3, when the query radius increases, the Relative Time Difference is monotonically non-decreasing for the both Diagonal datasets and real
datasets. And for synthetic data, the RTD can take up to 19.4% of the DQ Slope Response Time, this ratio is as high as 20.7% of the DQ Slope Response Time for real datasets.

![Graph showing relative time difference between DQ slope and DQ centroid for varying query window sizes.](image1)

**Figure 6.2a.** Relative Time Difference between DQ Slope and DQ Centroid for Varying Query Window Sizes (Synthetic Data)

![Graph showing relative time difference between DQ slope and DQ centroid for varying query window sizes.](image2)

**Figure 6.2b.** Relative Time Difference between DQ Slope and DQ Centroid for Varying Query Window Sizes (Real Datasets)
### Chapter 6. Experimental Results: Distance Query

#### Table 6.4a. Time Difference vs. Number of Rows Processed for DQ Slope and DQ Centroid (Synthetic Datasets)

<table>
<thead>
<tr>
<th>File</th>
<th>Query Window</th>
<th># Rows Processed</th>
<th>DQS Total Time (ms)</th>
<th>DQC Total Time (ms)</th>
<th>(Time Difference) / (# Rows Processed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagonal1</td>
<td>Q1</td>
<td>3295</td>
<td>279</td>
<td>249</td>
<td>9.1e-3</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>Q2</td>
<td>8062</td>
<td>494</td>
<td>424</td>
<td>8.7e-3</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>Q3</td>
<td>14009</td>
<td>656</td>
<td>532</td>
<td>8.9e-3</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>Q4</td>
<td>18531</td>
<td>848</td>
<td>688</td>
<td>8.6e-3</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>Q5</td>
<td>24110</td>
<td>1045</td>
<td>842</td>
<td>8.4e-3</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>Q1</td>
<td>1971</td>
<td>223</td>
<td>207</td>
<td>8.1e-3</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>Q2</td>
<td>6310</td>
<td>448</td>
<td>396</td>
<td>8.2e-3</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>Q3</td>
<td>12431</td>
<td>605</td>
<td>503</td>
<td>8.2e-3</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>Q4</td>
<td>17233</td>
<td>828</td>
<td>686</td>
<td>8.2e-3</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>Q5</td>
<td>22968</td>
<td>1025</td>
<td>837</td>
<td>8.2e-3</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>Q1</td>
<td>653</td>
<td>150</td>
<td>144</td>
<td>9.2e-3</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>Q2</td>
<td>2567</td>
<td>294</td>
<td>271</td>
<td>9.0e-3</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>Q3</td>
<td>6169</td>
<td>435</td>
<td>378</td>
<td>9.2e-3</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>Q4</td>
<td>9983</td>
<td>636</td>
<td>551</td>
<td>8.5e-3</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>Q5</td>
<td>15487</td>
<td>814</td>
<td>677</td>
<td>8.8e-3</td>
</tr>
</tbody>
</table>

#### Table 6.4b. Time Difference vs. Number of Rows Processed for DQ Slope and DQ Centroid (Real Datasets)

<table>
<thead>
<tr>
<th>File</th>
<th>Query Window</th>
<th># Rows Processed</th>
<th>DQS Total Time (ms)</th>
<th>DQC Total Time (ms)</th>
<th>(Time Difference) / (# Rows Processed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EC</td>
<td>Q1</td>
<td>317</td>
<td>105</td>
<td>103</td>
<td>6.3e-3</td>
</tr>
<tr>
<td>EC</td>
<td>Q2</td>
<td>1094</td>
<td>167</td>
<td>158</td>
<td>8.2e-3</td>
</tr>
<tr>
<td>EC</td>
<td>Q3</td>
<td>3863</td>
<td>297</td>
<td>266</td>
<td>8.0e-3</td>
</tr>
<tr>
<td>EC</td>
<td>Q4</td>
<td>6368</td>
<td>412</td>
<td>356</td>
<td>8.8e-3</td>
</tr>
<tr>
<td>EC</td>
<td>Q5</td>
<td>10608</td>
<td>575</td>
<td>484</td>
<td>8.6e-3</td>
</tr>
<tr>
<td>MA</td>
<td>Q1</td>
<td>1605</td>
<td>175</td>
<td>161</td>
<td>8.7e-3</td>
</tr>
<tr>
<td>MA</td>
<td>Q2</td>
<td>5319</td>
<td>376</td>
<td>329</td>
<td>8.8e-3</td>
</tr>
<tr>
<td>MA</td>
<td>Q3</td>
<td>12813</td>
<td>635</td>
<td>533</td>
<td>8.0e-3</td>
</tr>
<tr>
<td>MA</td>
<td>Q4</td>
<td>20858</td>
<td>1119</td>
<td>936</td>
<td>8.8e-3</td>
</tr>
<tr>
<td>MA</td>
<td>Q5</td>
<td>31950</td>
<td>1536</td>
<td>1280</td>
<td>8.0e-3</td>
</tr>
<tr>
<td>CT</td>
<td>Q1</td>
<td>4962</td>
<td>364</td>
<td>323</td>
<td>8.3e-3</td>
</tr>
<tr>
<td>CT</td>
<td>Q2</td>
<td>17631</td>
<td>976</td>
<td>837</td>
<td>7.9e-3</td>
</tr>
<tr>
<td>CT</td>
<td>Q3</td>
<td>50554</td>
<td>2206</td>
<td>1766</td>
<td>8.7e-3</td>
</tr>
<tr>
<td>CT</td>
<td>Q4</td>
<td>83900</td>
<td>3429</td>
<td>2733</td>
<td>7.6e-3</td>
</tr>
<tr>
<td>CT</td>
<td>Q5</td>
<td>92397</td>
<td>3515</td>
<td>2787</td>
<td>7.9e-3</td>
</tr>
</tbody>
</table>

It is interesting to observe that the increasing speed of RTD is getting slower when Q3 and bigger query windows are used. This is largely due to the increase of the denominator.
(Slope Total Time). That is why even though the Time Difference is increasing the effect of RTD finally reaches a plateau.

6.3 Experiments 5 and 6: Varying the Size of Sorted List

The purpose of Experiments 5 and 6 is to check the effect of varying sizes of sorted list on the Distance Query response time. For synthetic data, we use the Diagonal datasets with 50,000 line segments; for real data, we choose EC, MA and CT. Four sorted lists, with sizes 10, 25, 50 and 100 respectively, are set up in these two tests.

<table>
<thead>
<tr>
<th>File</th>
<th>Sorted List Size</th>
<th>DQS # Sorts</th>
<th>DQC # Sorts</th>
<th>DQS Total Time (ms)</th>
<th>DQC Total Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagonal1</td>
<td>10</td>
<td>77</td>
<td>77</td>
<td>555</td>
<td>431</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>25</td>
<td>156</td>
<td>156</td>
<td>588</td>
<td>466</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>50</td>
<td>254</td>
<td>253</td>
<td>656</td>
<td>532</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>100</td>
<td>453</td>
<td>453</td>
<td>909</td>
<td>790</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>10</td>
<td>66</td>
<td>66</td>
<td>506</td>
<td>398</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>25</td>
<td>148</td>
<td>148</td>
<td>540</td>
<td>433</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>50</td>
<td>284</td>
<td>284</td>
<td>605</td>
<td>503</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>100</td>
<td>479</td>
<td>482</td>
<td>895</td>
<td>788</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>10</td>
<td>86</td>
<td>86</td>
<td>319</td>
<td>265</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>25</td>
<td>173</td>
<td>173</td>
<td>355</td>
<td>300</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>50</td>
<td>277</td>
<td>277</td>
<td>435</td>
<td>378</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>100</td>
<td>429</td>
<td>428</td>
<td>662</td>
<td>610</td>
</tr>
</tbody>
</table>

Table 6.5a. Varying Sorted List Size for DQ Slope & DQ Centroid (Synthetic Data)

<table>
<thead>
<tr>
<th>File</th>
<th>Sorted List Size</th>
<th># Rows Processed</th>
<th>Time Difference</th>
<th>(Time Difference) / (DQS Total Time) %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagonal1</td>
<td>10</td>
<td>14009</td>
<td>124</td>
<td>22.3%</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>25</td>
<td>14009</td>
<td>122</td>
<td>20.7%</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>50</td>
<td>14009</td>
<td>124</td>
<td>18.9%</td>
</tr>
<tr>
<td>Diagonal1</td>
<td>100</td>
<td>14009</td>
<td>119</td>
<td>13.1%</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>10</td>
<td>12431</td>
<td>108</td>
<td>21.3%</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>25</td>
<td>12431</td>
<td>107</td>
<td>19.8%</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>50</td>
<td>12431</td>
<td>102</td>
<td>16.9%</td>
</tr>
<tr>
<td>Diagonal2</td>
<td>100</td>
<td>12431</td>
<td>107</td>
<td>12.0%</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>10</td>
<td>6169</td>
<td>54</td>
<td>16.9%</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>25</td>
<td>6169</td>
<td>55</td>
<td>15.5%</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>50</td>
<td>6169</td>
<td>57</td>
<td>13.1%</td>
</tr>
<tr>
<td>Diagonal3</td>
<td>100</td>
<td>6169</td>
<td>52</td>
<td>7.9%</td>
</tr>
</tbody>
</table>

Table 6.5b. Varying Sorted List Size for DQ Slope & DQ Centroid (Synthetic Data)
We observe in Table 6.5, as the sorted list size increases, the Total Response Time will increase for both DQ Slope and DQ Centroid. This is consistent with our intuitive expectation because keeping a larger list in order involves more sorts of the list. At the same time, the cost for each sort is generally more expensive for a larger list. This expectation is also confirmed by our experimental results.

Furthermore, we notice from Table 6.5b that the Time Difference between DQ Slope and DQ Centroid within each dataset is almost constant. Again this is consistent with our intuition that the time difference between DQS and DQC for varying sorted list size is primarily the time difference in distance calculation between the PTP and PTLS distance metrics. Because the Number of Rows Processed for each dataset is constant and the Number of Sorts Difference between DQ Slope and DQ Centroid for Diagonal datasets is very small, the effect of Number of Sorts in the Time Difference can be ignored. Figure 6.4a illustrates the Time Difference between DQS and DQC for Varying Sorted List Size. And Figure 6.4b shows the decreasing trends of RTD as the sorted list size increases.
Figure 6.3b. Relative Time Difference between DQ Slope and DQ Centroid for Varying Sorted List Size (Synthetic Data)

<table>
<thead>
<tr>
<th>File</th>
<th>Sorted List Size</th>
<th>DQS # Sorts</th>
<th>DQC # Sorts</th>
<th>DQS Total Time (ms)</th>
<th>DQC Total Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EC</td>
<td>10</td>
<td>100</td>
<td>103</td>
<td>215</td>
<td>181</td>
</tr>
<tr>
<td>EC</td>
<td>25</td>
<td>161</td>
<td>161</td>
<td>244</td>
<td>212</td>
</tr>
<tr>
<td>EC</td>
<td>50</td>
<td>236</td>
<td>238</td>
<td>298</td>
<td>267</td>
</tr>
<tr>
<td>EC</td>
<td>100</td>
<td>323</td>
<td>325</td>
<td>485</td>
<td>452</td>
</tr>
<tr>
<td>MA</td>
<td>10</td>
<td>88</td>
<td>90</td>
<td>523</td>
<td>412</td>
</tr>
<tr>
<td>MA</td>
<td>25</td>
<td>185</td>
<td>187</td>
<td>559</td>
<td>449</td>
</tr>
<tr>
<td>MA</td>
<td>50</td>
<td>286</td>
<td>288</td>
<td>635</td>
<td>533</td>
</tr>
<tr>
<td>MA</td>
<td>100</td>
<td>554</td>
<td>556</td>
<td>962</td>
<td>856</td>
</tr>
<tr>
<td>CT</td>
<td>10</td>
<td>340</td>
<td>354</td>
<td>2002</td>
<td>1517</td>
</tr>
<tr>
<td>CT</td>
<td>25</td>
<td>509</td>
<td>522</td>
<td>2073</td>
<td>1599</td>
</tr>
<tr>
<td>CT</td>
<td>50</td>
<td>756</td>
<td>760</td>
<td>2206</td>
<td>1766</td>
</tr>
<tr>
<td>CT</td>
<td>100</td>
<td>1241</td>
<td>1256</td>
<td>2997</td>
<td>2501</td>
</tr>
</tbody>
</table>

Table 6.6a. Varying Sorted List Size for DQ Slope and DQ Centroid (Real Data)
Table 6.6b. Varying Sorted List Size for DQ Slope and DQ Centroid (Real Data)

Table 6.6 shows the DQ Slope and DQ Centroid results for varying size of sorted list on real datasets. We can see that the Number of Sorts, the Total Response Time, Time Difference and RTD all keep the similar trends as Diagonal dataset. However, the Time Difference for MA and CT datasets is not perfectly stable. We observe that the Number of Sorts Difference between DQS and DQC for these two datasets is no longer small enough to be ignored. In other words, the Time Difference in sorting the list makes some contribution to the Total Time Difference. If we take this factor into consideration, we can still draw the conclusion that the time difference between DQ Slope and DQ Centroid is almost constant.

Figure 6.4a illustrates the Time Difference between DQS and DQC for real datasets for Varying Sorted List Size, and Figure 6.4b shows the decreasing trends of RTD as the sorted list size increases.
Figure 6.4a. Time Difference between DQ Slope and DQ Centroid for Varying Sorted List Size (Real Datasets)

Figure 6.4b. Relative Time Difference between DQ Slope and DQ Centroid for Varying Sorted List Size (Real Datasets)
6.4 Experiments Summary

We can draw the following conclusions from the experimental results above:

1. The DQ algorithms have a stable performance regardless of the Input Data Size, the Query Window Size and Sorted List Size.
   - DQ Slope and DQ Centroid have similar behavior on Response Time, Number of I/Os and Number of Sorts on both synthetic data and real data.
   - The Response Time can be precisely predicted after knowing the size of the Query Window, the Number of Records to be processed and the size of output list; and these three factors are the primary determinant of Response Time.

2. The Relative Time Difference (RTD) between DQ Slope and DQ Centroid can be as high as 24.2% of the Total Response Time. For both synthetic data and real data, the time difference is mainly caused by the different distance functions.
Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis, we have presented two algorithms to answer two distance-based queries, namely the Sphere Intersection Query (SIQ) and the Distance Query (DQ). Both of these two algorithms make use of the Filter Tree as the spatial index. Experiments were performed on both centroid datasets and slope datasets.

For both Sphere Intersection Query and Distance Query, the experimental results show that the total response time can be precisely estimated. For SIQ, the primary response time determinant is the output size; for DQ, the total response time is primarily determined by the number of records that we have to process, the size of the output list and the choice of the distance function. We observe that stable performance is achieved mainly due to the structure of the Filter Tree. Unlike all the other DQ algorithms, our algorithm is straightforward. There is no need to differentiate the leaf nodes and the non-leaf nodes in the Filter Tree as in other tree structures. We simply scan through the files that contain the candidate cells within the possible Hilbert value range.

The Total Response Time for the Slope datasets is always 3% to 24.2% longer than the corresponding Centroid datasets for both SIQ and DQ. The percentage of difference keeps increasing as the number of output records increases. This difference is mainly because of the difference in complexity in calculating distance using slope and centroid datasets.
7.2 Future Work

7.2.1 Data with Higher Dimensionality

In the implementation of SIQ and DQ algorithms, we need to calculate the Hilbert values in two-dimensional space. This can be an important factor in total response time. We retained the algorithm that Kennedy proposed in his Master’s thesis [Kennedy 1999]. Though the author claimed that it is forty times faster than the algorithm used by Koudas [Koudas 1997], which can handle the data with arbitrary number of dimensions, it cannot easily be extended to higher dimensions. In other words, the response time could be greatly affected for higher dimensional data.

7.2.2 Distance Join and Semi-Distance Join

After successfully answering distance-based queries on the Filter Tree, it will be very interesting to perform distance-based join operations. Hjaltason and Samet proposed a Distance Join [Hjaltason and Samet 1998] that computes a subset of the Cartesian product of two sets of objects based on distance predicates, and sorts the result set in a specified order.

Sevcik and Koudas claim that it is also more time efficient to perform spatial join operation on the Filter Tree than R-Tree [Sevcik and Koudas 1996]. Intuitively, we would expect the similar performance for Distance Join. However, to achieve comparably superior performance, some details need to be carefully investigated. We recall that for a full spatial join on the Filter Tree, minimal amount of I/Os is guaranteed. This is because the cells in each level of the hierarchy are disjoint, so if we would like to search for the matches between two relations in one cell of relation one, we can restrict our search to the corresponding cell and its enclosing cells at higher levels. However, this is not always true for distance-based joins since sometimes the query range may overlap with cells other than its enclosing cells at higher levels. In this case, the index sweeping algorithm cannot be easily applied, hence we cannot guarantee that each block of the entity index file is read only once for a full spatial join.
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


Appendix A

The Sphere Intersection Query Algorithm Using the Filter Tree

Algorithm for Range Calculation

- calculate total size of the input file
- look up containment level in in-memory table based on this
- read index pages into the buffer

loop through each cell along each border of the query window
  - calculate the Hilbert Value of both the cell just inside and just outside the query window
  - subtract the Hilbert Value of the cell just outside the query window from the Hilbert Value of the cell just inside the query window
  - if the difference is 1
    - this is the start of a Hilbert range; add this Hilbert Value to a list of starting values of Hilbert ranges
  - end if
  - if the difference is -1
    - this is the end of a Hilbert range; add this Hilbert Value to a list of ending values of Hilbert ranges
  - end if
- end loop

sort the list of starting values of Hilbert ranges and the list of ending values of Hilbert ranges, determining pairs of numbers that specify the Hilbert ranges of interest

loop through all levels from 0 to 19
  - loop through each of the ranges of Hilbert Values that intersect the query window
    - call RangeSearch
  - end loop
- end loop
Algorithm for Range Search

read first page of the index file for this level
if there is only one index page for this level of the Filter Tree
   loop through every cell in the index page
   if this cell’s Hilbert Value is within a set of query ranges
      read the corresponding data page
      call SIQ for this data page and the query window
   end if
end loop
else
   loop through all pointers in the index page
   read index page pointed to by current pointer
   loop through every cell in the index page
   if this cell’s Hilbert Value is within a set of query ranges
      read the data page
      call SIQ for this data page and the query window
   end if
end loop
end loop

Algorithm for SIQ

loop through each object in the data page
   if the object’s Hilbert Value is within the start and end of the range
      call Distance$^1$ for each object to the query center
      if the Distance is no bigger than the radius
         write the coordinates of the object to an in-memory page
         if this page is now full, write to disk
      end if
   end if
end loop

Algorithm for Distance (Centroid)

calculate the distance between the query center and the centroid of the object

$^1$ There are two Distance functions, namely Distance Centroid and Distance Slope. Call corresponding distance function when performing Centroid or Slope Sphere Intersection Query.
Algorithm for Distance (Slope)

if the slope of the MBR is positive
    if the query center is greater than the upper-right corner of the MBR
        the distance is from the center to the upper-right corner
    else if the query center is smaller than the lower-left corner of the MBR
        the distance is from the center to the lower-left corner
    else calculate the distance from the center to the line of the slope
        if the vertical point is outside the MBR
            the distance is the shorter one of the distance from the center to the upper-right corner or to the lower-left corner
        else the distance is from the center to the line
    end if
else if the slope of the MBR is negative
    if the x coordinate of the query center is smaller and the y coordinate of the query center is greater than the upper-left corner of the MBR
        the distance is from the query center to the upper-left corner
    else if the query center's x value is bigger and the y value is smaller than the lower-right corner of the rectangle
        the distance is from the query center to the lower-right corner
    else calculate the distance from the query center to the line of the slope
        if the vertical point is outside the MBR
            the distance is the shorter one of the distance from the center to the upper-left corner or to the lower-right corner
        else the distance is from the query center to the line
    end if
end if
Appendix B

The Distance Query Algorithm Using the Filter Tree

Algorithm for Range Calculation

calculate total size of the input file
look up containment level in in-memory table based on this
read index pages into the Buffer

loop through each cell along each border of the query window
  calculate the Hilbert Value of both the cell just inside and just outside the query window
  subtract the Hilbert Value of the cell just outside the query window from the Hilbert Value of the cell just inside the query window
  if the difference is 1
    this is the start of a Hilbert range; add this Hilbert Value to a list of starting values of Hilbert ranges
  end if
  if the difference is -1
    this is the end of a Hilbert range; add this Hilbert Value to a list of ending values of Hilbert ranges
  end if
end loop
sort the list of starting values of Hilbert ranges and the list of ending values of Hilbert ranges,
determining pairs of numbers of numbers that specify the Hilbert ranges of interest

initialize all the elements in the sorted list with distance equal to 1
loop through all levels from 0 to 19
  loop through each of the ranges of Hilbert Values that intersect the query window
    call RangeSearch
  end loop
end loop
Algorithm for Range Search

read first page of the index file for this level
if there is only one index page for this level of the Filter Tree
    loop through every cell in the index page
        if this cell's Hilbert Value is within the query ranges
            read the corresponding data page
            call DQ for this data page and the query window
        end if
    end loop
else
    loop through all pointers in the index page
    read index page pointed to by current pointer
    loop through every cell in the index page
        if this cell's Hilbert Value is within the query ranges
            read the data page
            call DQ for this data page and the query window
        end if
    end loop
end if

Algorithm for DQ

loop through each object in the data page
if the object’s Hilbert Value is within the start and end of the range
    call Distance for each object to the query center
    if the Distance is smaller than the last element in the list
        if the output list is not full
            insert the new object into the list
        else if the list is filled up for the first time
            sort the list;
            replace the new object with the last element in the list and sort the list;
        else if the list is full
            replace the last element in the list with the new object and sort the list;
        end if
    end if
end loop

Algorithm for Distance (Centroid)

calculate the distance between the query center and the centroid of the object
Algorithm for Distance (Slope)

if the slope of the MBR is positive
    if the query center is greater than the upper-right corner of the MBR
        the distance is from the center to the upper-right corner
    else if the query center is smaller than the lower-left corner of the MBR
        the distance is from the center to the lower-left corner
    else calculate the distance from the center to the line of the slope
        if the vertical point is outside the MBR
            the distance is the shorter one of the distance from the center to the upper-right corner or to the lower-left corner
        else the distance is from the center to the line
    end if
else if the slope of the MBR is negative
    if the x coordinate of the query center is smaller and the y coordinate of the query center is greater than the upper-left corner of the MBR
        the distance is from the query center to the upper-left corner
    else if the query center’s x value is bigger and the y value is smaller than the lower-right corner of the rectangle
        the distance is from the query center to the lower-right corner
    else calculate the distance from the query center to the line of the slope
        if the vertical point is outside the MBR
            the distance is the shorter one of the distance from the center to the upper-left corner or to the lower-right corner
        else the distance is from the center to the line
    end if
end if