THESIS

MAINTAINING MULTIMEDIA DATA IN A GEOSPATIAL DATABASE

by

Mitchakima D. Banks

September 2012

Thesis Advisor:	Mathias Kölsch
Second Reader:	Arijit Das

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2012	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE Maintaining Multimedia Data in a Geospatial Database			**5. FUNDING NUMBERS**
6. AUTHOR(S) Mitchakima D. Banks			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ______N/A______.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

The maintenance and organization of data in any profession, government or commercial, is becoming increasingly more challenging. Adding components, whether those components are two- or three dimensional, further increases the complexity of databases. It is harder to determine which database software to choose to meet the needs of the organization. This thesis evaluates the performance of two databases as spatial functions are executed on columns containing spatial data using benchmark testing.

Evaluating the performance of spatial databases makes it possible to identify performance issues with spatial queries. The process of conducting a performance evaluation of multiple databases, in this thesis, focuses on the measurement of each elapsed time within each database.

The work already implemented in evaluating the performance of spatial databases did not explore a database's performance as it returned large and small result sets. The overhead of returning large or small result sets was not considered. Therefore, a custom test was developed to engage the aspects of prior work found beneficial. Using a database the researchers built with well over one million records, the elapsed time in adding records was measured. The elapsed time of the spatial functions queries was measured next.

The results showed areas where each database excelled given multiple conditions. A different look at PostgreSQL and MySQL as spatial databases was offered. Given their results, as each database produced result sets from zero to 100,000, it was learned that the performance of each database could differ depending on the volume of information it is expected to return.

14. SUBJECT TERMS Database, Spatial, Benchmark			**15. NUMBER OF PAGES** 57
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

MAINTAINING MULTIMEDIA DATA IN A GEOSPATIAL DATABASE

Mitchakima D. Banks
Lieutenant, United States Coast Guard
B.S., Voorhees College, 2000

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2012

Author:	Mitchakima D. Banks
Approved by:	Mathias Kölsch Thesis Advisor
	Arijit Das Second Reader
	Peter J. Denning Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The maintenance and organization of data in any profession, government or commercial, is becoming increasingly more challenging. Adding components, whether those components are two- or three dimensional, further increases the complexity of databases. It is harder to determine which database software to choose to meet the needs of the organization. This thesis evaluates the performance of two databases as spatial functions are executed on columns containing spatial data using benchmark testing.

Evaluating the performance of spatial databases makes it possible to identify performance issues with spatial queries. The process of conducting a performance evaluation of multiple databases, in this thesis, focuses on the measurement of each elapsed time within each database.

The work already implemented in evaluating the performance of spatial databases did not explore a database’s performance as it returned large and small result sets. The overhead of returning large or small result sets was not considered. Therefore, a custom test was developed to engage the aspects of prior work found beneficial. Using a database the researchers built with well over one million records, the elapsed time in adding records was measured. The elapsed time of the spatial functions queries was measured next.

The results showed areas where each database excelled given multiple conditions. A different look at PostgreSQL and MySQL as spatial databases was offered. Given their results, as each database produced result sets from zero to 100,000, it was learned that the performance of each database could differ depending on the volume of information it is expected to return.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Originally, this thesis resulted from a project in which imagery was captured and automatically stored. Historical data was integrated with it and made available through the network to permit a TIVO-style replay of events anywhere within a city limit, with a focus and higher spatial and temporal resolution on hotspots. The purpose was to allow law enforcement personnel to go back in time to a location and recover imagery/video of interest in an expedited timely manner.

While working on the project, the question of database performance arose. Deciding on an open-source relational database management system (RDBMS), MySQL or PostgreSQL, to handle a spatial database of this magnitude, was a decision that had to be made. So what is so special about spatial databases? Which database is the best database? No doubt exists that both are admirable proven solutions, but how do they perform against each other? Does the multimedia data add any additional complexity to the database?

The goal of this thesis is to compare and analyze the usage of two database options, PostgreSQL and MySQL, as spatial databases, in a practical situation. The intent is not to prove that one database will always outperform the other. However, whether one database will perform better than the other is examined based on specific performance requirements, and some performance considerations are provided when deciding which database is right in any given situation.

Some background information about databases is given in Chapter II. Chapter III describes the benchmark testing used to determine which database outperforms the other. The specifics of the experiment are given in Chapter IV, including some SQL example code that could be used to replicate results. The actual results of the experiment are illustrated using graphs and tables and explained in Chapter V. Chapter VI takes the results of Chapter V along with lessons learned along the way, to discuss the outcome of this work and give recommendations for future work. Appendices are provided to offer some of the Java code used to generate data.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

This chapter provides background information about relational databases, examines the uniqueness of spatial relational databases, as well as determines what is special about these databases. Many of the definitions relating to spatial and non-spatial databases are presented. Beginning with the general discussion about databases and how they work, information about spatial databases, how they work and what makes them special is discussed next.

A. SPATIAL DATA

To give some background into what a spatial database is and how it differs from a non-spatial database, it is important to start from the beginning by describing what a non-spatial database is and then move into the specifics of a spatial database.

1. Non-Spatial Database

A database is a collection of tables of information to manage, as well as the definitions of the relationships that exist between those tables. Databases offer an organized tool for storing, managing and retrieving information in tables. A table is a collection of information managed by columns and rows that are uniquely identified by their name. Each column is basically a category of information, or attributes, and each row is a record of the column information.

2. Spatial Database

A spatial database is a relational database management system (RDBMS) that supports spatial data types in the same way as any other data in the database [1]. Relational databases connect data from different files, using a key field, or common data element. Each table in a relational database uses a key field to uniquely identify each row and connect one table to another. An RDBMS is the software used by a relational database. A spatial database stores and queries data related to objects in space. Spatial data includes points, lines, and polygons. Spatial data consists of two types, geometry and geography. While these two types behave very similarly, they are different. For the purposes of this research, the difference focused on between the two data types is the fact that geometry data types relate to planar, or “flat-earth” data, while geography data types relate to geodetic, or “round-earth” data.

a. Geometry

Geometry data types work with planar models. The earth is treated as a flat projection, without taking the curvature of the earth into account. It is primarily used for describing short distances. The Cartesian coordinate system is used with geometry data types, and it specifies points in a plane by an ordered pair of numeric coordinates. They are measured in units of length, and can be expressed using all possibilities of real numbers. Unlike in a geodetic system, in the planar system, ring orientation of a polygon is not an important factor (i.e., polygon((2, 2), (2, 4), (0, 4), (0, 2), (2, 2)) is the same as polygon((2, 2), (0, 2), (2, 4), (0, 4), (2, 2))).

Figure 1 shows a typical graph used in geometry, in which two polygons, polygon A and polygon B, are intersecting. In comparison to Figure 2, a geography graph with two polygons intersecting is provided.

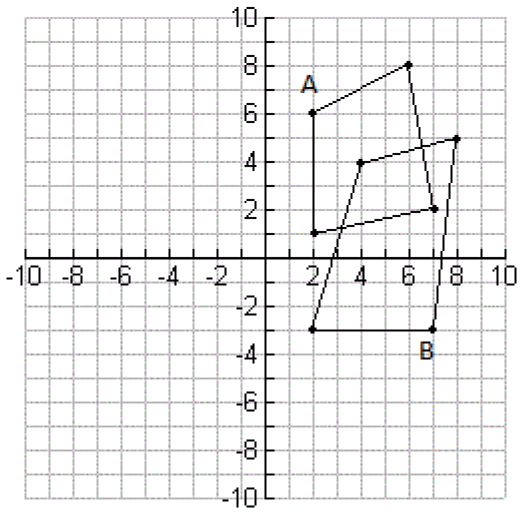

Figure 1. Geometry graph with polygon A and polygon B intersecting.

b. Geography

Geography data types work with geodetic information, and it factors the curved surface of the earth into its calculations. The position information is given in ellipsoid coordinates expressed in degrees of longitude and latitude. Each spatial instance has a Spatial Reference ID (SRID). "The SRID is the spatial reference identification system. The SRID is part of a set of standards developed for cartography, surveying, and geodetic storage" [2]. The SRID is not necessary for geometry data. Both location and orientation must be accurately described in geography.

True geodetic support, which is support for true measurement along a spherical coordinate), is not offered by MySQL [3]. PostgreSQL does offer it, but only for point-to-point non-indexed distance functions. Figure 2 shows a geography graph in which two polygons, polygon A and polygon B, are intersecting.

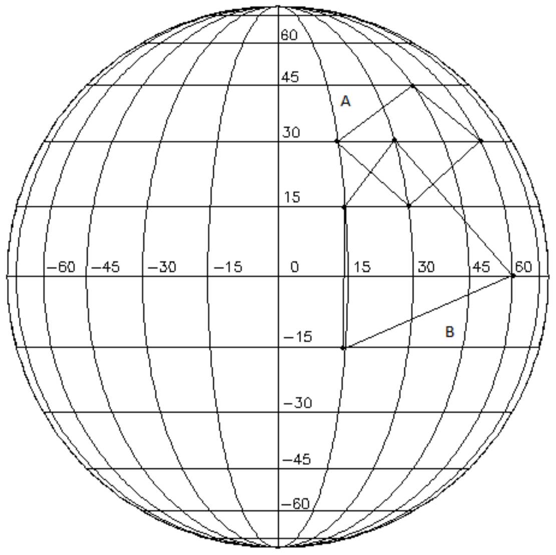

Figure 2. Geography or "round earth" graph with polygon A and polygon B Intersecting.

B. INDEXING

Indexing is a way to improve the speed of data retrieval. Like the index in a book, a data structure can be referred to when searching for an item of information. Within relational databases, an index eliminates the need to scan each record in a table sequentially. Database indexes can occupy as much disk memory as data, and can be the cause of slower writes and increased storage space. However, for the purpose of this thesis, the full tradeoff of indexing, if any exists, is not explored. Indices are primarily used to enhance database performance, but inappropriate use can result in slower performance. This section begins with a general discussion of indices and then elaborates on the spatial indexing used within the experiment.

No provision exists for creating indices within the SQL standard. All database software includes the technology needed to index columns. Therefore, the indexing differences within the two database software options discussed in this thesis are explored individually. Table 1 shows which indices both MySL and PostgreSQL support.

	B-/B+ tree	**R-/R+ tree**	**GiST**	**GIN**	**fulltesxt**	**Spatial**
MySQL	Yes	MyISAM tables only	No	No	MyISAM tables only	MyISAM tables only
PostgreSQL	Yes	GiST	yes	yes	yes	PostGIS

Table 1. Indices supported.

1. Non-Spatial Indexing

Without indices, MySQL starts reading the first record, then reads the entire table searching for relevant rows, at a cost of O(n). If it is anticipated that most queries will be run on a specific column, indexing that column can quickly yield the location the system should search within the table, without having to look at all the data. "Most MySQL indexes (PRIMARY KEY, UNIQUE, INDEX, and FULLTEXT) are stored in B-trees" [4], which is explained later in this section.

Normally, all indices are created on a table at the time the table is created [4], but CREATE INDEX can be used to add indices to existing tables.

```
CREATE INDEX example_index ON sampleTable (column1);
```

Within PostgreSQL, B-tree, GiST, and GIN, are provided, as seen in Table 1. Each index is explained later in this section. Indexing is not necessary for full text searching. Full text searching consists of examining each word in each cell of a database for matches, which is contrary to searching a database for matches on parts of the original data, or metadata. When a column is searched on a regular basis, an index is more desirable.

a. B-tree Indexing

In B-tree indexing, or generalized binary search tree, each node can have more than two child nodes, but the lower and upper bounds on that number of child nodes are typically fixed. Thus, in a 2-3 B-tree, the internal child nodes may only be between 2 and 3 [5], as seen in Figure 3. In the worst and average case scenario, the cost of a B-tree search is O(log n).

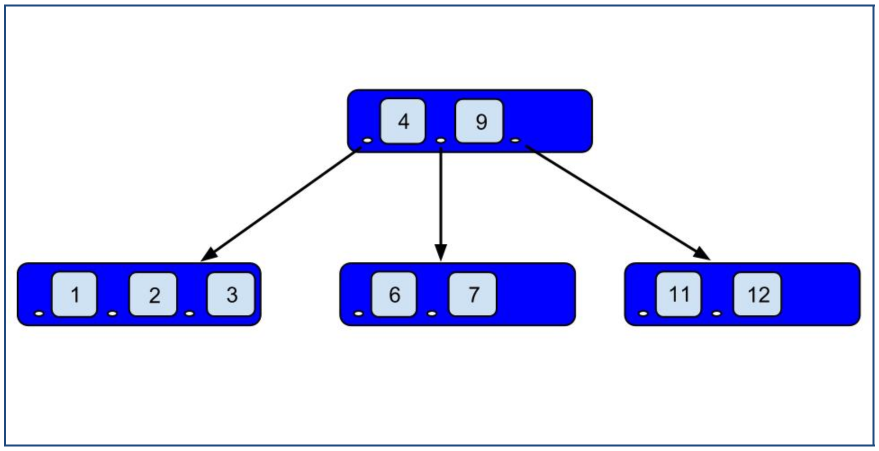

Figure 3. 2-3 B-tree configuration.

The following is SQL code to create a B-tree index on a column named column1 in a sample table named sampleTable:

```
CREATE UNIQUE INDEX example_Index ON sampleTable (column1);
```

b. GiST Indexing

GiST, Generalized Inverted Search Tree index, is a more general index. It can be used for broader purposes, like geographical points and polygons, than B-tree indexes. Hellerstein, Naughton, and Pfeffer [6] describe GiST as being "lossy" because each document is represented in the index by a fixed-length signature.

> The signature is generated by hashing each word into a random bit in an n-bit string, with all these bits OR-ed together to produce an n-bit document signature. When two words hash to the same bit positions, the result is false matches. If all the words in the query have matches, whether they are real or false, the table row must be retrieved to see if each match is correct [PostgreSQL does this automatically]. [7]

The issue with "lossiness" in GiST causes performance degradation due to useless fetches, but again, this thesis does not focus on the full tradeoff of indexing.

c. GIN Indexing

GIN, Generalized Inverted Index, is not "lossy."

> It is a way to index one record with several keys. GIN index accepts an array of expressions as a parameter and uses each element of the array as a key. This gives a many-to-many mapping between keys and records. Each key potentially points to many records, and each record is potentially pointed to by many keys. [8]

2. Indexing Spatial Data

Spatial databases employ spatial indices to optimize search results from spatial queries. Spatial indexing helps the computer make sense of the unique layout of the world according to a spatial database. It is a set of rules that assists in organizing the information in a database. In MySQL, spatial column types use R-tree indexing, explained in Section a, while PostgreSQL spatial columns use GiST, further explained in Section 1.b.

a. R-tree Indexing

The key idea behind R-tree (Rectangle tree) indexing is that the database structure groups nearby objects and represents them with a minimum bounding rectangle in the next higher level of the tree, as seen in Figure 4. In the worst case scenario of R-tree indexing, the cost to search is O(n). In the best case, the cost is $O(\log_M n)$, where M is the maximum number of children per node, and n is the number of levels in a tree.

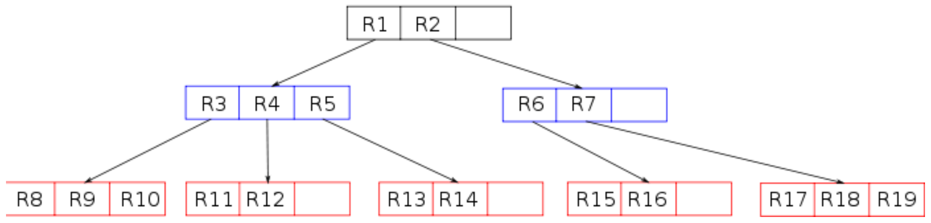

Figure 4. 2D R-tree From [9].

C. ARCHITECTURE

In comparing the architecture of the two databases described in this thesis, some differences arose, specifically in the comparison of both databases' storage engines, which is significant only since ignoring storage engines could result in undue influence of one database software outperforming the other. For the purposes of this thesis, the storage engine influences on performance are not explored.

1. Storage Engines

PostgreSQL is a unified database server with a single storage engine. MySQL has two layers, an upper SQL layer and a set of storage engines. In the comparison of the two databases, it is necessary to specify which storage engines are being used with MySQL because suitability, performance and (even basic) feature availability are greatly affected. The most commonly used storage engines in MySQL are InnoDB for high performance on large workloads with lots of concurrency, and MyISAM for lower concurrency workloads or higher concurrency read-mostly data [10]. Read-mostly data is the data indexed, because it is read most. Another important feature about MyISAM is that it supports spatial indexing.

D. BENCHMARKS

One of the goals of this thesis is to be able to show performance of two databases, MySQL and PostgreSQL. To assess relative performance, benchmarks are used to measure both basic spatial database operations and normal real-world applications. In general, database benchmarks attempt to simulate, as closely as possible, real-world workloads, while reflecting an organization's actual workload. Objects in a GIS are stored in two-dimensional space, which adds to the complexities faced in simulating a databases' workload when the data is three-dimensional, latitude, longitude, and elevation. Within this thesis, this third dimension is not added.

1. Current Benchmarking

Numerous whitepapers exist that describe benchmarks on databases. However, the number of benchmark tests used for spatial database performance is considerable smaller. Some of the current benchmark tests explored in creating the benchmark for this thesis are discussed below.

Ray-Simion-Brown [1] created a benchmark named Jackpine. Jackpine has micro benchmarks to test basic spatial operations in isolation and macro workload scenarios, such as map search and browsing, geocoding, reverse geocoding, flood risk, land information management, and toxic spill analysis. Jackpine was more in-depth than this thesis process required, in that it examined aspects of topology relationship and use cases upon which this thesis did not focus.

The benchmark test created by Stonebreaker-Frew-Gardels-Meredith [11] was closer to being achievable and relevant to this research; however, this benchmark examined raster data, point data, polygon data, and directed graph data. In representing each of these classes, the authors added a variable not an issue in this thesis, cost. The SEQUIOA 2000 factored in costs to its performance results, which added some additional weight to the result set.

Powers' benchmark test [12] was the closest benchmark test to what was attempted to accomplish in this thesis. It used 20 queries to assess performance, which are broken down as follows.

- 9 general spatial queries
- 5 hierarchical queries
- 4 queries to explore time series data
- 1 linked the time series data to its spatial location
- 1 combined spatial, monitoring locations and time series data

Using these performance categories, Powers' work [12] was used to test both MySQL and PostgreSQL.

Myllymaki and Kaufman [13] implemented a location-based service test that queried proximity (range queries), k-nearest neighbor, and sorted-distance. It measured "elapsed time of location updates, spatial queries, and spatial index creation and maintenance" [12].

2. Reason for Custom Benchmark

The Jackpine macro benchmark illustrated performance of spatial function queries for different databases, but some aspects of performance are omitted in determining performance. A number of aspects of this benchmark testing were not necessary for the performance test in this thesis. Powers [1] benchmark tests were closer to what the researchers of this thesis were trying to achieve, but it was not sufficient as-is for determining performance. Thus, the basis of this thesis benchmark tests is greatly influenced by this work. What happens when instead of inserting 1,000 records, 10,000 records are added? Do the databases still perform the same when the result set returned by each database is altered? This aspect of performance is explored more within the research for thesis, but was omitted from the other benchmark tests discussed.

THIS PAGE INTENTIONALLY LEFT BLANK

III. METHODOLOGY

The approach used in this thesis for measuring performance is a simple benchmark test developed to show the performance of both PostgreSQL and MySQL in the “normal” operations of a geospatial database. This benchmark test is not intended to stress either database. For the purposes of this thesis, stressing the database would include adding multiple users and testing the performance as each user is added. It would also include performing multiple complex queries simultaneously. This benchmark will in no way do this.

Much like Power’s work [1], MySQL and PostgreSQL are compared using a small dataset and queries. However, the goal of thesis was to provide more than a preliminary investigation of these databases with the aim to establish which system performs best. Although indexing is a major part of database optimization, this benchmark does not fully test indexing and all the tradeoffs of indexing, nor does it test every indexing option available to PostgreSQL and MySQL.

This chapter is divided into two main sections. The first section describes the performance of each database as it executes a fundamental database operation, that of adding data into the database. Each database executes an INSERT, to add information into their prospective database. The second section shows the performance of four queries executed on each of the two databases. Then, the latency and throughput of these databases are measured. In each case, the queries are described in words and expressed in SQL.

A. BENCHMARKING DATA INSERTED TO THE TABLE

This section describes how the performance of each database was evaluated as 10, 100, 500, 1,000, 5,000, 10,000, and 100,000 records are added to a database that has over 1 million records already stored. Beyond the PRIMARY KEY, no indexing was applied to the databases as records were being loaded.

It is important to begin timing the insert after the connection has been made because the raw time it takes to load data is more accurately measured without the time it takes to connect to the database that adds additional time to the processing of data, which is the case for all the data being measured.

1. PostgreSQL

The following is the SQL code to insert records into the PostgreSQL database.

INSERT INTO image **VALUES** (1, now(), ‘image1.jpg’, ‘images/image1.jpg’, ‘image 1.kml’, ‘point‘(X, Y)’ ‘, ‘polygon‘((X1, Y1), (X2, Y2), (X3, Y3), (X4, Y4), (X1, Y1))’)’;

2. MySQL

The following is the SQL code to insert records into the MySQL database. Note that the code is very similar to the PostgreSQL code. The difference is in how each database stores points and polygons. To insert a point, PostgreSQL separates the X- and Y- Coordinate with a comma, while MySQL separates the X- and Y Coordinate with a space. To insert a polygon, PostgreSQL separates the X- and Y Coordinate with a comma and separates the X- Y- pair with parenthesis and a comma, while MySQL separates the X- and Y- Coordinate with a space and separates the X- Y- pair with a comma:

INSERT INTO image **VALUES** (1, now(), ‘image1.jpg’, ‘images/image1.jpg’, ‘image 1.kml’, ‘point‘(X Y)’ ‘, ‘polygon‘((X1 Y1, X2 Y2, X3 Y3, X4, Y4, X1 Y1))’)’;

B. BENCHMARKING DATA RETRIEVED FROM TABLE

This section describes the tests of spatial performance performed for this thesis. Spatial performance could be tested in a number of ways. For instance, general spatial queries could be performed that accepted each polygon in each record and returned its area. The number and/or a list of polygons that intersected each of the given polygons in the database could also be returned. The decision was made to keep the testing simple and more controlled by measuring how much time each database took in returning a result set at known milestones for four spatial functions. Sets of records at different intervals must be present in the database for testing purposes, which began by providing the X- and Y- Coordinates for a polygon not in the database, then measuring how much time it took the database to search and return a response that the polygon was not found. A polygon was then queried for that had exactly 10 matches in the database, and then how much time it took the database to search and return a response that included the ten polygons was measured. The same was performed for 100 matches, 500 matches, 1,000 matches, 5,000 matches, 10,000 and 100,000 matches.

1. CONTAINS (Polygon)

The first query was the CONTAINS (polygon) query. This query is labeled CONTAINS (polygon), because it is based on the relationship of one polygon to another polygon. Another query within this method tests the relationship of a given point and a polygon. The CONTAINS (polygon) query finds all the polygons contained within a given polygon. For both PostgreSQL and MySQL, this query includes both polygons equal to testPolygon and polygons smaller than testPolygon. It does not include polygons larger than testPolygon. The following shows the SQL code to run a CONTAINS (polygon) query for PostgreSQL and MySQL.

PostgreSQL

```
testPolygon = “Polygon'((-120.764653,35.716847), (-
120.764651,35.716847), (-120.76465,35.716845), (-
120.764653,35.716845), (-120.764653,35.716847))’”
SELECT * FROM image
WHERE " + testPolygon + " <@ corners
```

MySQL

```
testPolygon = “Polygon(( -120.764653 35.716847,-
120.764651,35.716847),(-120.76465,35.716845),(-
120.764653,35.716845,-120.764653 35.716847))”
SELECT * FROM image1
WHERE CONTAINS(GeomFromText(‘ + testPolygon + '), corners);
```

2. EQUALS

The second query was the EQUALS query. The EQUALS query finds all the polygons exactly equal to the testPolygon. The following shows the SQL code to run an EQUALS query for PostgreSQL then MySQL.

PostgreSQL

```
SELECT * FROM image
WHERE " + testPolygon + " ~= corners
```

MySQL

```
SELECT * FROM image1
WHERE EQUALS(GeomFromText('" + testPolygon + "'), corners)
```

3. OVERLAPS and INTERSECTS

The third query was the OVERLAPS and INTERSECTS query. The OVERLAPS and INTERSECTS query finds all the polygons that overlaps a given polygon (testPolygon). For PostgreSQL, this query includes polygons equal to testPolygon, polygons smaller than testPolygon, and polygons larger than testPolygon. As long as some portion of a polygon overlaps testPolygon, it is considered as overlap. For MySQL, this is not the case. MySQL excludes polygons equal spatially. For this reason, MySQL measures INTERSECTS, because the PostgreSQL's definition of OVERLAPS and MySQL's definition of INTERSECTS are more closely alike. The following shows the SQL code to run an OVERLAPS query for PostgreSQL and an INTERSECTS query for MySQL.

PostgreSQL

```
SELECT * FROM image
WHERE box(" + testPolygon + ") && box(corners)
```

MySQL

```
SELECT * FROM image1
WHERE INTERSECTS(GeomFromText('" + testPolygon+ "'), corners)
```

4. CONTAINS (Point)

The final query was the CONTAINS (point) query. The CONTAINS (point) query finds all the polygons contained within a certain radius (for PostgreSQL) or bounding box (for MySQL) of a center point, labeled testPoint. This query is the most complicated query found within the method because an additional function was needed to compute the bounding box for MySQL. For PostgreSQL, both the point and the polygons are placed inside a circle and all circles created around the polygons equal to or smaller than the circle placed around testPoint were returned as matches. The result set does not include polygons in which the circle placed around the polygon was larger than the circle placed around the testPoint. The following shows the SQL code to run a CONTAINS (point) query for PostgreSQL and MySQL.

PostgreSQL

```
SELECT * FROM image
WHERE circle(" + testPoint + ", 2.0) && circle(corners)
```

MySQL

```
SELECT * FROM image1
WHERE INTERSECTS(GeomFromText(AsText(Envelope
(GeomFromText('LineString(" + lowerX +" "+ lowerY + ", "
+ centerX +" "+ centerY +", " + upperX + " " + upperY +")')))),
corners)
```

MySQL has additional variables: lowerX, lowerY, centerX, center, upperX, upperY. These variables create a lineString that defines a bounding box (Envelope). Unlike PostgreSQL, MySQL does not have a function that defines a circle around a point.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. EXPERIMENT SETUP

Chapter III described the custom benchmark used to evaluate some spatial features and one non-spatial function of two open source databases. In this section, the experimental setup is described. The experimental setup goes in depth into the creation of the databases and discusses the implementation of the benchmark testing.

A. OVERVIEW

In creating a GIS that maintains multimedia data, it is first important to note that the actual multimedia data is not housed within the databases. Since the multimedia data is not dynamic, and storing images in a database requires the database to transport large amounts of data, the databases hold links to the multimedia data, not the actual data. Also, storing images in the database does not allow the data to be accessible by external image viewers. Multimedia data has not added any additional complexities to the database for this thesis.

1. Hardware and Connection

The machine running the database client and conducting the benchmark tests was a MAC OS X version 10.7.4, with two quad-core intel xeon processors. The system has a processor speed of 2.4GHz, and 16GB of memory, which was connected to using JDBC. The JDBC connection pool was created before any data was entered in the databases or any queries run to ensure that the overhead of a connection to the database was not accounted for in the metrics. Both databases were running at the same time, but no benchmark tests were performed simultaneously. The hardware load was comparable in all cases and no other processes were running.

2. Database

The databases created and evaluated were PostgreSQL9.2 and MySQL5.5. The databases were installed “as-is.” No tuning was performed on any of the databases. The tables have an “image_id”, that is a primary key; “image_date”, which is the date the image was taken; “image_filename”, which is the name of the actual image;

"image_directory", combines with the "image_name" to create the link to the image; "kml_filname", which is the name of the file that holds the metadata about the image and combines with the "image_directory" to create the link to the metadata; and two spatial columns, called "Center" and "Corners." "Center" is a point and "Corners" is a polygon. The following shows the PostgreSQL statement used to create the geometry database.

```
CREATE TABLE image ( image_id NUMBER(38,0),
 image_date DATE() default now(),
 image_filename VARCHAR ( 100 ),
 image_directory VARCHAR ( 512 ),
 kml_filename VARCHAR ( 100 ),
 center POINT,
 corners POLYGON,
 PRIMARY KEY (image_id)
);
```

The MySQL database was also created as a geometry database, with spatial features and a MyISAM storage engine, with the following SQL code.

```
CREATE TABLE image ( image_id int(38) unsigned primary key not null,
 image_date DATE,
 image_filename VARCHAR ( 100 ),
 image_directory VARCHAR ( 512 ),
 kml_filename VARCHAR ( 100 ),
 center geometry,
 corners geometry) ENGINE=MyISAM;
```

Both databases were initially created without an index. The index was added later in the experiment to eliminate any latency issues.

B. BUILDING THE DATABASE

To create the database, the records needed to be fabricated to look as close to actual spatial data as possible. The two columns this thesis is most concerned with are the "Center" column and the "Corners" column because these are the two spatial columns. Therefore, all spatial queries are run against these columns. The following sections discuss how the "Center" data and the "Corners" data were created.

1. “Center”

The “Center” column was created with two sets of data, random data and data called milestone-data for the purposes of this thesis. The random data is the data not expect to be utilized. Their purpose is to add extra records to the database so that performance can be tested on a database with a significant number of records (1.2 million). The milestone-data set is deliberately placed into the database to be returned when a query is run against the database. This data is the most important data in this experiment because this data assists in measuring the database performance.

a. Random Data

Within the database, 1,212,251 records were created to make the database larger. While these records are not measured in the experiment, the goal was to have at least enough variety in this dataset to make indexing necessary. This random data was created by using coordinates that would in no way intersect with the milestone datasets. The java code used to generate the “Center” columns’ X- and Y- coordinates for the random data can be found in Appendix A.

The code in Appendix A generates the X- and Y- coordinates for this thesis with an outer loop and inner loop pair. The initial X- coordinate is created. Then, a specified number of Y- coordinates are created (in this case 200) to accompany the X- coordinate, which continues until 200 X- coordinates with 200 Y- coordinates each (i.e., 40,000 records). Due to the milestone-data needed to be mixed in with the random data, only a few thousand random data records are created without interruption. Next, some milestone-data records are added in, and then random data records again, which continues until enough records are added to the database.

b. Milestone Data

A total of 116,610 records were created for measuring each database’s performance. These records were created in groups of 10, 100, 500, 1,000, 5,000, 10,000 and 100,000. They were created very similarly to the random records, except no random number generator exits with the milestone-data records because these numbers needed to be deliberately placed in the database as discussed in Chapter III. A closer look at the code used to create the dataset to query EQUALS, in Appendix B, shows the same inner and outer loop as the random data records, without the X- and Y- coordinates.

The code in Appendix B generated the X- and Y- coordinates for the "Center" column of a dataset of 10 records, used to query EQUALS. Table 2 shows the changes necessary to create the other milestone-datasets, which are needed to query EQUALS. As stated in Section one of Chapter IV.B, these datasets were spread out throughout the database.

"m" value	Center X	Center Y
10	-100.6565	65.6972
100	-100.6565	35.6972
500	-100.6565	15.6972
1000	-100.6565	85.6972
5000	-35.6972	15.6565
10000	-35.6972	65.6565
0[1]	-15.6972	65.6565

Table 2. Changes required to create the datasets.

The milestone data generated to create the datasets for the other benchmark tests can be found in Appendix C.

2. "Corners"

The "Corners" column is formed by taking the X- and Y- coordinates from the data provided in the "Center" column as its arguments. A bounding box measuring five

[1] The 0 dataset was created to ensure that performance of the database would be measured when no records were returned.

miles by five miles is created around the point obtained from the data in the "Center" column. Then, the coordinates created by this bounding box are used as the coordinates of the "Corners" column.

C. TESTING THE DATA

The performance of the databases was tested while performing spatial relationship queries first. The CONTAINS(polygon) query was run first that searched for a polygon not expected to be in the database. The amount of time it took the databases to return that response and ensured that it did return the appropriate "No data found" response was then recorded. Next, the performance of each database was measured as it searched for a polygon for which it was expected to find ten matches. The performance times were recorded, and ensured that each database returned the appropriate response. This action was repeated for datasets of 100, 500, 1,000, 5,000, 10,000, and 100,000. The exact same test for EQUALS, OVERLAPS and INTERSECTS, and CONTAINS(point) was run, and the performance of the databases measured, and ensured that the appropriate responses were returned.

After all the queries were executed and the data recorded, a performance test was run to ascertain how long it took each database to load 10, then 100, and 500, 1,000, 5,000, 10,000, and last, 10,0000 additional records. The same code initially used to generate random data records was executed, and a timing wrapper was put around the code to measure the time taken by each database to execute the test. The information was recorded and the data generated by this test was erased.

THIS PAGE INTENTIONALLY LEFT BLANK

V. BENCHMARK RESULTS

A. RESULTS OF DATA INSERT

This section provides the results produced from loading data into each database. Both a graph and table are provided to give the most accurate picture possible of the results.

1. Data Load Results

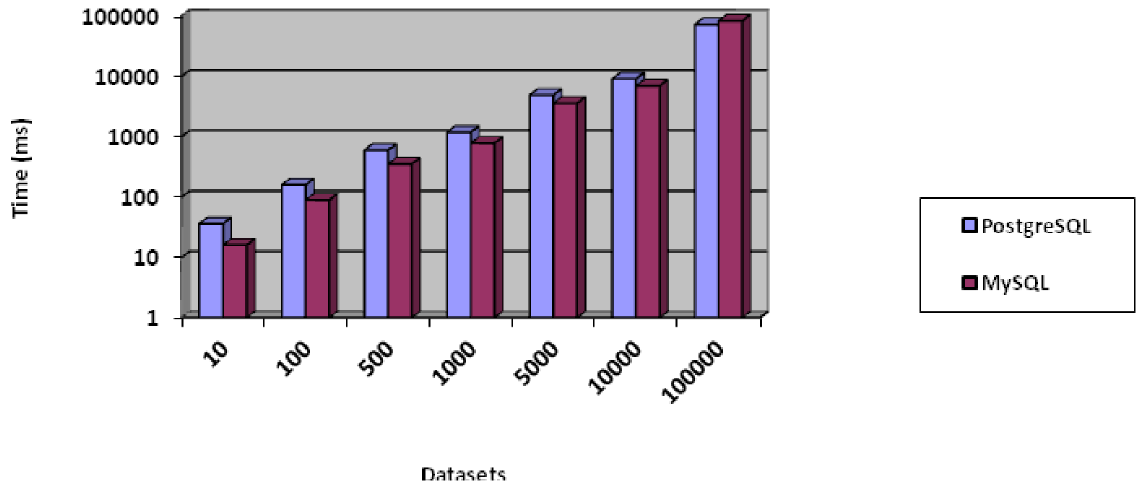

Figure 5. Load data graph.

	Datasets						
	10	**100**	**500**	**1000**	**5000**	**10000**	**100000**
PostgreSQL	36ms	159ms	596ms	1180ms	4865ms	9055ms	71951ms
MySQL	16ms	89ms	352ms	780ms	3571ms	6916ms	82582ms

Table 3. Benchmark time for loading data.

The results above, in Figure 5 and Table 3, show the amount of time it took each database to load 10 data records, 100 records, 500 records, 1,000 records, 5,000 records, 10,000 records, and 100,000 records. The time was measured in milliseconds and the data was plotted in log time.

MySQL performed this task faster than PostgreSQL for data sets less than 100,000. Once the data sets reached 100,000, a shift in performance occurred. PostgreSQL began to outperform MySQL. Adding smaller amounts of data into a PostgreSQL database resulted in additional overhead. Base on this information, it could be argued that if employing PostgreSQL, it would be best to do bulk data loading, rather than one or two records at a time.

B. RESULTS FROM QUERY SET

This section gives the results produced from executing spatial relationship queries on each database. Both graphs and tables to give the most accurate picture possible of the results are provided.

1. CONTAINS (Polygon)

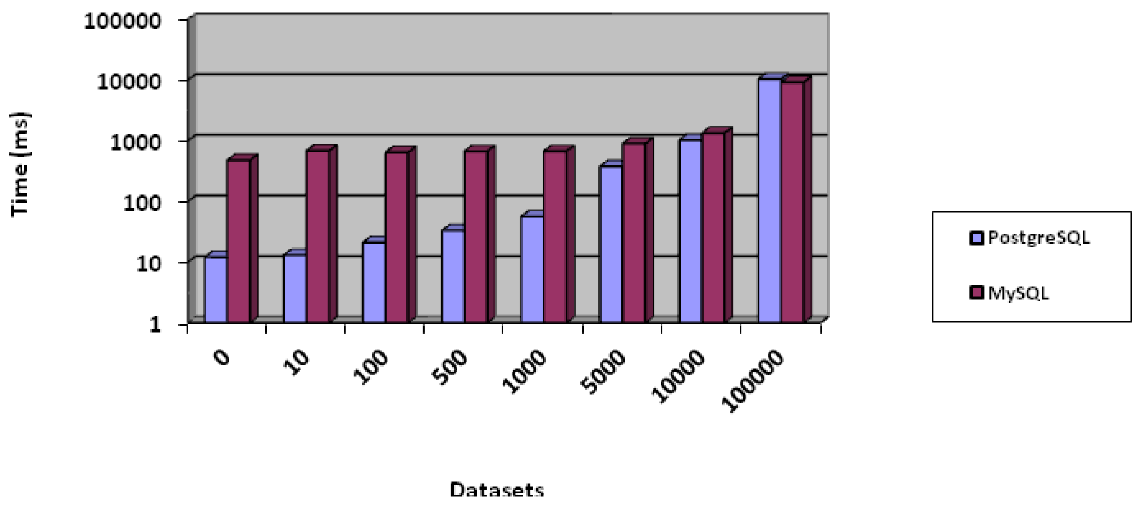

Figure 6. Query1: One polygon spatially contains another.

	Datasets						
	10	**100**	**500**	**1000**	**5000**	**10000**	**100000**
PostgreSQL	12ms	13ms	21ms	33ms	56ms	377ms	1018ms
MySQL	477ms	681ms	641ms	667ms	668ms	896ms	1330ms

Table 4. Query1: One polygon spatially contains another.

Figure 6 and Table 4 show the amount of time it took PostgreSQL and MySQL to run a query using a CONTAINS function. As the table and graph show, PostgreSQL was significantly faster than MySQL at running this query, when the result sets were small. As the result set became larger, the delta of the amount of time it took PostgreSQL and MySQL to return results got smaller. Once a result set of 100,000 was reached, MySQL performed better than PostgreSQL. The time was measured in milliseconds and the graph was plotted in log time.

PostgreSQL performed well in executing queries of one polygon containing another polygon for small datasets. It did not do as well for a larger dataset. It was anticipated that the results of this query would be an indicator for the results of the subsequent queries.

2. EQUALS

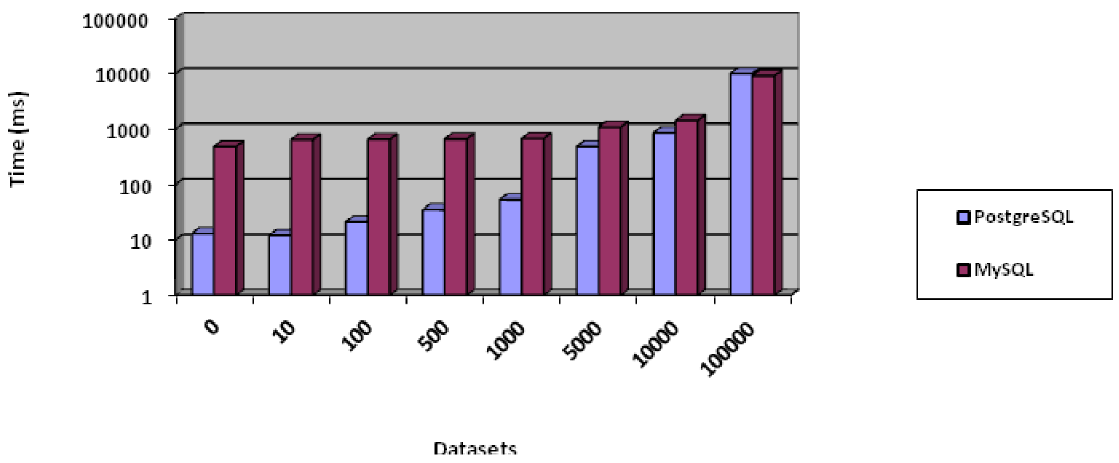

Figure 7. Query2: Two polygons equal.

	Datasets						
	10	**100**	**500**	**1000**	**5000**	**10000**	**100000**
PostgreSQL	13ms	12ms	21ms	35ms	53ms	482ms	856ms
MySQL	481ms	643ms	652ms	662ms	675ms	1080ms	1419ms

Table 5. Query2: Two polygons equal.

In Figure 7 and Table 5, PostgreSQL, again, was significantly faster than MySQL, until datasets in the 100,000 range were reached. As the result set became larger, the delta between the databases again diminished. At 100,000, MySQL performed better than PostgreSQL. The time was measured in milliseconds and the graph was plotted in log time.

These results shown for this query were similar to the results of CONTAINS (polygon). PostgreSQL outperformed MySQL in executing queries when two polygons were equal when the datasets were small. Performance decreased as the dataset became larger.

3. OVERLAPS and INTERSECTS

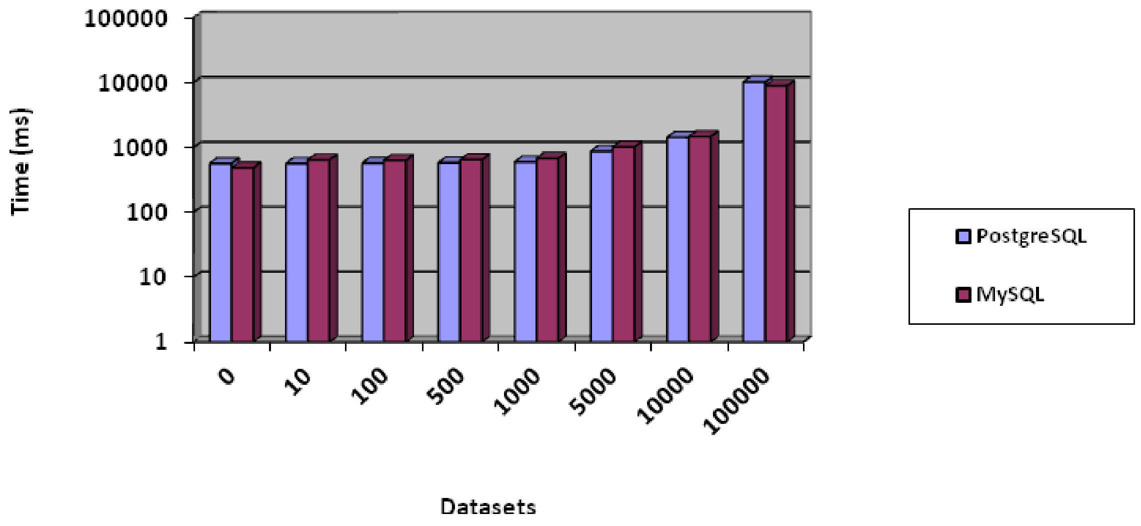

Figure 8. Query3: One polygon overlaps or intersects another.

	Datasets						
	10	**100**	**500**	**1000**	**5000**	**10000**	**100000**
PostgreSQL	563ms	564ms	571ms	582ms	596ms	873ms	1426ms
MySQL	483ms	641ms	633ms	647ms	677ms	1017ms	1467ms

Table 6. Query3: One polygon overlaps or intersects another.

Figure 8 and Table 6 show a performance difference in PostgreSQL. In querying for polygons that spatially overlap or intersect, PostgreSQL and MySQL performance was very close. When no results were returned, and with result sets of 100,000, MySQL performed better than PostgreSQL. With regard to all the other result sets, PostgreSQL outperformed MySQL. The time was measured in milliseconds and the graph was plotted in log time.

The results of this query were unexpected. PostgreSQL performance in executing an OVERLAP query was almost five times worst than its performance in executing a CONTAINS (polygon) query and an EQUALS query. MySQL remained consistent with its performance between the three queries. The reason for these unexpected results is not known, and is a question to explore in future work.

4. CONTAINS (Point)

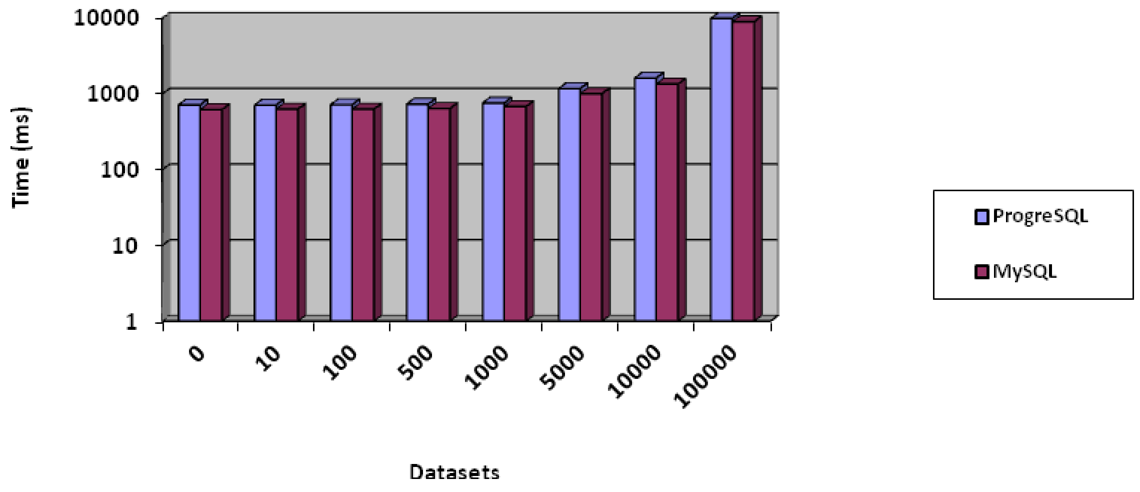

Figure 9. Query 4: Polygons contained within a specified radius of a point.

	Datasets						
	10	**100**	**500**	**1000**	**5000**	**10000**	**100000**
PostgreSQL	717ms	714ms	721ms	735ms	760ms	1176ms	1617ms
MySQL	622ms	632ms	632ms	644ms	684ms	1009ms	1351ms

Table 7. Query4: Polygons contained within a specified radius of a point.

In querying for polygons contained within a five-mile radius of a point, as shown in Figure 9 and Table 7, PostgreSQL and MySQL performance was very close. Unlike the other benchmark tests, MySQL outperformed PostgreSQL in this area. The time was measured in milliseconds and the graph was plotted in log time.

The results of this query were not in line with what was anticipated. Since PostgreSQL has a circle function to calculate a radius around a point, it was anticipated that this built-in functionality would yield faster results than the equation created to calculate a line and a bounding box around a point, which was not the case. The assessment of these results indicates significant overhead in using PostgreSQL circle function.

C. ANALYSIS OF THE RESULTS

Overall, both databases had areas in which they excelled and areas in which they did not perform well. While in most cases PostgreSQL performed better than MySQL, MySQL appeared to be more consistent. To explain further what is meant by "more consistent," with regard to MySQL, the overhead in returning 10 results was not much different than the overhead in returning 5,000 results. PostgreSQL, on the other hand, differed greatly in the amount of time taken in returning 10 results as opposed to 5,000 results. As the number of result set grew, PostgreSQL overhead grew. The initial assessment that the first query would be an indicator of the results of the subsequent queries was wrong. In loading data, MySQL performed this task faster than PostgreSQL, until 100,000 records were loaded. At that time, PostgreSQL achieved better results. The reverse is true in executing queries.

MySQL performed better than PostgreSQL in all instances in which the result set was as large as 100,000. It is believed that the reason for the difference in performance with PostgreSQL was the amount of false-positives produced by the GiST index. When the result sets were small (i.e., less than 1,000), PostgreSQL was able to retrieve the table rows and check to see that each match was correct, due to false-positive matches created by the index, relatively quickly, yielding better performance than MySQL in most of the result sets. As the datasets grew, it took more time for PostgreSQL to find and remove these false-positive matches produced by its GiST index.

MySQL has a huge startup cost, or overhead, in performing these spatial queries. This startup cast is more noticeable in executing the CONTAINS and EQUALS queries, where PostgreSQL overhead was much lower. It was possible to draw any reasonable conclusions as to why.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSION

This thesis attempted to compare PostgreSQL and MySQL and prove that in a practical application of a spatial database, one spatial database would perform better than the other. In determining which database was “best,” an attempt was made to provide some insight about the differences of PostgreSQL and MySQL.

A benchmark of the performance of two spatial databases, PostgreSQL and MySQL, was provided. This benchmark test was created using measures from current benchmark tests. Two databases of over 1.2 million records were created. One database was created in PostgreSQL and the other in MySQL. The benchmark tests for each query were run at each established interval (0, 10, 100, 500, 1,000, 5,000, 10,000 and 100,000) and the results recorded. A timer was then added to the code used to generate the random data for this thesis. The performance of the databases was tested as they loaded data, and recorded the results.

While conducting the experiment, the importance of MySQL’s storage engine and indexing was ascertained. The first time the MySQL database was loaded without the proper storage engine and no secondary indexing applied, it loaded three times slower than PostgreSQL. In executing the queries, MySQL returned performance times of one to three seconds, while PostgreSQL returned performance times of 400–600 milliseconds. The effects of the storage engine and indexing was not examine in this thesis, but the latency associated with not employing the correct storage engine, and timing performance with no indexing applied, was experienced.

Also learned was that PostgreSQL and MySQL use different terms at times. As an example, OVERLAPS, in PostgreSQL means the same as INTERSECTS in MySQL. Not knowing this difference originally caused inconsistent results in this experiment.

This thesis did not do what it was intended to do, which was to prove that one database would outperform the other, but some differences in the two databases, PostgreSQL and MySQL, did occur. Both databases excelled at certain tasks and fell short on others.

This thesis offered a different look at PostgreSQL and MySQL as spatial databases handling multimedia data. Unlike other similar research, this thesis explored the differences between these two databases as they produced larger result sets. Understanding the effects of these databases as they yield larger result sets helps to make more informed decisions for organizations trying to determine which database would work best for them.

As a suggestion for future research, a comparative analysis of the R-tree indexing that MySQL offers, and the GiST indexing that PostgreSQL offers against the quad tree indexing offered by Oracle, is recommended.

APPENDIX A. RANDOM DATA GENERATOR

```
Database db1 = new Database(model);
db1.init();
db1.openDB();
int n = 200; // This number is replaced by the number of X coordinate to create
int m = 200; // This number is replaced by the number of Y coordinate to create
Random randGenerator = new Random();
for( int i = 0; i < n; i++ ) //outer loop for the X coordinate
{
        // generates a random double number
        double incr2 = (randomGenerator.nextInt(99) * 0.0001);
        double centerX = -120.76465; //initialize X coordinate
        for(int j = 0; j < m; j++) //inner loop for the Y coordinate
        {
                double centerY = 35.716847; //initialize Y coordinate
                // generates a random double number
                double incr1 = (randomGenerator.nextInt(99) * 0.0002);
                GeoImage gImage = new GeoImage();
                db1.addGeoImage(gImage, centerX, centerY);

                centerY += incr1; //adds a pseudo-random number to Y coordinate
        }
        centerX += incr2; //adds a pseudo-random number to X coordinate
 }
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. MILESTONE DATA GENERATOR FOR EQUALS QUERY

```
Database db1 = new Database(model);
db1.init();
db1.openDB();
int n = 1; // This number is replaced by the number of X coordinates to create
int m = 10; // This number is replaced by the number of Y coordinates to create
for( int i = 0; i < n; i++ ) //outer loop for the X coordinate
{
        double centerX = -100.6565; //initialize X coordinate
        for(int j = 0; j < m; j++) //inner loop for the Y coordinate
        {
                double centerY = 65.6972; //initialize Y coordinate

                GeoImage gImage = new GeoImage();
                db1.addGeoImage(gImage, centerX, centerY);
        }
 }
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. GENERAL MILESTONE DATA GENERATOR

```
Database db1 = new Database(model);
db1.init();
db1.openDB();
int n = 1; // This number is replaced by the number of longitudes to create
int m = 10; // This number is replaced by the number of latitudes to create
for( int i = 0; i < n; i++ ) //outer loop for the longitude
{
        double centerX = -100.6565; //initialize the X coordinate
        for(int j = 0; j < m; j++) //inner loop for the Y coordinate
        {
                double centerY = 65.6972; //initialize latitude

                GeoImage gImage = new GeoImage();
                db1.addGeoImage(gImage, centerX, centerY);
                centerY += incr1; //increments the X coordinate by 0.00004
        }
        centerX += incr2; // increments the X coordinate by 0.00004
 }
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] S. Ray et al., “A benchmark to evaluate spatial database performance,” *International Conference on Data Engineering—ICDE*, 2011, pp. 1139–1150.

[2] *New data types*. (2012). [Online]. Available: http://technet.microsoft.com/en-us/magazine/2008.04.datatypes.aspx?pr=blog.

[3] *Cross compare SQL server 2008 spatial, postgreSQL/postGIS 1.3-1.4, MySQL 5-6*. (2012). [Online]. Available: http://www.bostongis.com/PrinterFriendly.aspx?content_name=sqlserver2008_postgis_mysql_compare.

[4] *MySQL 5.6 reference manual*. (2012). [Online]. Available: http://dev.mysql.com/doc/refman/5.6/en/index.

[5] *What is b-tree?*. (2012). [Online]. Available: http://encycl.opentopia.com/term/B-Tree.

[6] J. M. Hellerstein et al., “Generalized search trees for database systems,” in *Proc. 21st International Conference on VLDB*, 1995, pp. 562–573.

[7] *PostgreSQL 9.2.0 documentation, 12.9. GiST and GIN index types*. (n.d.). [Online]. Available: http://www.postgresql.org/docs/9.2/static/textsearch-indexes.html.

[8] *Explain extended*. (2010). [Online]. Available: http://explainextended.com/2010/02/02/searching-for-arbitrary-portions-of-a-date/

[9] A. Guttman, “R-trees. A dynamic index structure for spatial searching,” in *Proc. 1984 ACM-SIGMOD Conference on Management of Data*, 1984.

[10] *MySQL vs postgreSQL and MySQL(DDL,DML,DQL,DCL)*. (2012). [Online]. Available: http://www.techie-pro.com/tag/mysql/.

[11] M. Stonebraker et al., “The SEQUOIA 2000 storage benchmark,”in *SIGMOD ’93: Proc. of the 1993 ACM SIGMOD Intl conference on Management of data*, May 1993, pp. 2–11.

[12] R. Power, “Testing geospatial database implementations for water data,” in *Proc. of the 18th IMACS/MODSIM Congress*, July 2009.

[13] J. Myllymaki and J. Kaufman, “Dynamark: A benchmark for dynamic spatial indexing,” in *Proc. of the 4th Intl Conference on Mobile Data Management (MDM)*, January 2003, pp. 92–105.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California