Instructions for PC Users

COMPARATIVE ANALYSIS OF DATABASE SPATIAL TECHNOLOGIES (CADST)

by

Jodi Deprizio

A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Master of Science
Geoinformatics and Geospatial Intelligence

Committee:

______________________________	Dr. Ruixin Yang, Thesis Director
______________________________	Dr. Dieter Pfoser, Committee Member
______________________________	Dr. Andreas Zufle, Committee Member
______________________________	Dr. Dieter Pfoser, Department Chairperson
______________________________	Dr. Donna M. Fox, Associate Dean, Office of Student Affairs & Special Programs, College of Science
______________________________	Dr. Peggy Agouris, Dean, College of Science
Date: ______________________________	Summer Semester 2018 George Mason University Fairfax, VA

Comparative Analysis of Database Spatial Technologies (CADST)

A Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

by

Jodi Deprizio
Bachelor of Science
George Mason University, 2012

Director: Ruixin Yang, Associate Professor
Department of Geography and Geoinformation Science

Summer Semester 2018
George Mason University
Fairfax, VA

DEDICATION

This is dedicated to my loving husband Brad, mother Mary-Beth, and my two wonderful dogs, Chelsea and Ava.

ACKNOWLEDGEMENTS

I would like to thank the many friends, relatives, and supporters who have made this happen. My loving husband, Brad, kept me motivated while my dogs helped me with stress and anxiety. Drs. Yang, Pfoser, and Zufle, members of my committee, were of invaluable help throughout this process. Finally, thanks go out to the Fenwick Library for providing a clean, quiet, and well-equipped repository in which to work.

TABLE OF CONTENTS

Page

List of Tables ... vii

List of Figures ... viii

List of Abbreviations ... x

Abstract ... x

Chapter One: Introduction ... 1

Chapter Two: Literature Review ... 3

Chapter Three: Methodology ... 11

- Installation, Configuration, and Ingestion ... 12
  - MarkLogic ... 12
  - MySQL ... 14
  - Neo4j ... 15
  - MongoDB ... 17
  - PostgreSQL ... 18
- Key Metrics ... 20
  - About the Data ... 22
  - Querying the Data: ... 22

Results ... 29

- Ingestion and Storage ... 29
- Query Performance ... 31
- Accuracy ... 37
- Usability and Complexity ... 39

Conclusion and Future Research ... 41

Appendix ... 60

- Potomac Buffer KML File ... 60
- MySQL Queries ... 61
- Neo4j Queries ... 62

MarkLogic Queries 65
MongoDB Queries 67
PostgreSQL Queries.. 70
Query Runtime Results Table 72
References.. 88

LIST OF TABLES

Table Page
Table 1: Quick reference guide to the analyzed database and its respective model. 11
Table 2: Listing of the version, architecture, and install size of each database into the virtual machine. 11
Table 3: Listing of parameters for Virtual Machine Configurations. 12
Table 4: Description of the evaluation metrics 21
Table 5: Discrete Query Performance Results (time in seconds) Query 2 and 3 are an average of the average cold and warm run times for all 10 geometries queried. This is done for simplicity, but Table 15 in the Appendix section provides an entire detailed list of all query run times. 33
Table 6: Average runtime (seconds) for the overall (cold and warm) execution time for each query per database. Query 2 and 3 are an average of the average cold and warm run times for all 10 geometry queries. 34
Table 7: Count of results returned per query for each database. 38
Table 8: Overall ranking analysis of each system based on predefined metrics 59
Table 9: Example of the contents within the KML file 60
Table 10: MySQL supplemental code and data structure 61
Table 11: Neo4j supplemental code and data structure 62
Table 12: MarkLogic supplemental code and data structure 65
Table 13: MongoDB supplemental code and data structure 67
Table 14: PostgreSQL supplemental code and data structure 70
Table 15: List of all cold and warm query completion times per database and their calculated average 72

LIST OF FIGURES

Figure Page
Figure 1: ArcMap image of the 10 manually defined geospatial boundaries used for queries 2 and 3. 26
Figure 2: ArcMap image showing the 5-mile buffer area of interest used for query 4. ... 27
Figure 3: ArcMap image of the locations of Uranium deposits from the interim output of query 5. 28
Figure 4: Data ingest time (seconds) for each database to load the same dataset. 30
Figure 5: Size (MB) of each database after the same dataset was loaded. 31
Figure 6: Query Time (cold) in blue and Query Time (warm) in yellow for Query 1. Numbers shown are the time needed to process the query in seconds. 34
Figure 7: Query Time (cold) and Query Time (warm) for Query 2. Numbers shown are the time needed to process the query in seconds. 35
Figure 8:Figure 8: Query Time (cold) and Query Time (warm) for Query 3. Numbers shown are the time needed to process the query in seconds. 36
Figure 9: Query Time (cold) and Query Time (warm) for Query 4. Numbers shown are the time needed to process the query in seconds. 36
Figure 10: Query Time (cold) and Query Time (warm) for Query 5. Numbers shown are the time needed to process the query in seconds. 37
Figure 11. Data ingest time and storage footprint.......... 42
Figure 12: The JSON based data structure for MarkLogic and MongoDB. 44
Figure 13: View of a Neo4j node (a node contains keys and values). 45
Figure 14: MySQL (and PostrgeSQL data view (tabular). 45
Figure 15: Differences in the results returned from query 2a for MarkLogic, Mongo, and PostgreSQL and Neo4j and MySQL databases. 48
Figure 16: Differences in the results returned from query 2f for MarkLogic, Mongo, and PostgreSQL and Neo4j and MySQL databases. 49
Figure 17: Differences in the results returned from query 3a for MarkLogic, Mongo, and PostgreSQL and Neo4j and MySQL databases. 50
Figure 18: Differences in the results returned from query 3f for MarkLogic, Mongo, and PostgreSQL and Neo4j and MySQL databases. 51
Figure 19: Query 5 outputs for all 5 databases combined. 53
Figure 20: Query 5 results where points unique to only Neo4j are shown in orange while all else are in yellow. 54
Figure 21: Query 5 results where points unique to only MySQL are shown in purple while all else are in yellow.......... 55

Figure 22: Query 5 results where points unique to only Mongo are shown in cyan while all else are in yellow. 56

LIST OF ABBREVIATIONS

BinaryScript Object Notation .... BSON
Database Management System .... DBMS
Extensible Markup Language .... XML
Geographic Information System .... GIS
JavaScript Object Notation .... JSON
Mineral Resources Data System .... MRDS
My Structured Query Language .... MySQL
No (or Not Only) Structured Query Language .... NoSQL
Relational Database Management System .... RDBMS
Spatial Database Management System .... SDBMS
Structured Query Language .... SQL
United States Geological Survey .... USGS

ABSTRACT

COMPARATIVE ANALYSIS OF DATABASE SPATIAL TECHNOLOGIES (CADST)

Jodi Deprizio, M.S.

George Mason University, 2018

Thesis Director: Dr. Ruixin Yang

Spatial databases are increasingly utilized in, and are a major component of, any Geographic Information System (GIS). There are diverse types of SDBMS available, each with its own advantages and disadvantages, making it difficult to know which one is best suited for a given task. In addition, there is a lack of peer-reviewed literature on this subject specific to using GIS vector datasets that would help guide users into making the proper database choice. The following is a comprehensive comparison of spatial database management systems (SDBMS) for filling the gaps mentioned above. In this thesis five database technologies were analyzed and compared to determine which was more effective for use when storing and querying spatial vector data. Metrics for comparison were ingest performance, storage size, query performance, accuracy, system usability, and complexity. The databases analyzed were MySQL, MongoDB, MarkLogic, Neo4j, and PostgreSQL (with PostGIS). Each database had significant differences in data ingestion time, storage size, system usability, and complexity as well as substantial variations in query execution times.

CHAPTER ONE: INTRODUCTION

When database management systems (DBMS) were first developed, they focused primarily on storing generic tabular data with support for simple data types like text, numbers, and dates. The needs of a DBMS were typically limited to accounting and business data warehousing where data was stored and could be efficiently retrieved using simple queries. As data evolved over time, largely due to advancements in technology and the growing GIS movement, many databases added enhanced support for storing and querying more specific data types. These include objects, as well as semantic and spatial data (Worboys & Duckham, 2004; Guting, 1994; Shekhar & Chawla, 2003). In addition to these extensions, entirely new types of databases were being created to fill gaps left by traditional relational databases where the size and schema rigidity were issues. These limitations were mostly due to the onset of GIS and the copious amounts of geospatial data being collected.

Geospatial data, or spatial data, has geographic positioning information included within it that identifies features and boundaries in relation to their location on Earth. This data is usually stored as coordinates (latitude, longitude) or other spatial objects like lines and polygons, can be mapped, and are often found in large datasets. Non-spatial data is also relationally stored within a spatial dataset and is used to characterize features of objects not related to a spatial location, e.g. mineral name, deposit type, and lithologic and stratigraphic information (Gandhi et al., 2007). GIS is a major technological motivation for spatial databases (Shekhar & Chawla, 2003).

Spatial Database Management Systems, or SDBMS, can work with underlying DBMS and fall under the general category of GIS. They are used to create, store, visualize, process and manipulate geospatial data (Clarke, 2011; Worboys & Duckham, 2004; Guting, 1994; Shekhar & Chawla, 2003). A critical component of any GIS is the database as it is the basis of all decision making. Spatial data requires additional functionalities not readily available in a general-purpose DBMS that facilitates data extraction, storage, and analysis (Worboys & Duckham, 2004; Longley et al., 2001; Singleton & Longley, 2010). Some of these functionalities include spatial indexing, query optimization, and algorithms for processing spatial operations (Guting, 1994; Dolton & Lowe, 2001). There are many SDBMS that offer a wide range of features, many specific to a problem or data type. As a result, this can make choosing the right system challenging. This is especially true for data types specific to GIS because they can influence the resulting analysis.

CHAPTER TWO: LITERATURE REVIEW

There are several types of SDBMS used in GIS but the relational and non-relational models are the most prevalent (Healey, 1991). The relational database management system, or RDBMS, was created by a researcher who worked for IBM in the 1970's named Edgar Codd. His goal was to set up a relational schema that allowed users to easily retrieve and store data without redundancy (Codd, 1970). The relational model uses collections of tables that represent stored objects. Each table has rows and columns where the rows store data for the object and each column represents an attribute. The stored data in these tables are linked by using unique values such as an index or primary key. All associated tables have the unique primary key (per row) but in the linked tables (non-initial) the attribute is called a foreign key. A Relational join is achievable when a primary key in one table matches a foreign key in another table (Healey, 1991). SQL, or Structured Query Language, is used to query and maintain the data within a relational database. SQL, the most widely used database language, was one of the first commercial languages used with Codd's relational model. A RDBMS requires a schema to be defined before adding any records to the database and changes to it can be difficult, requiring transformation and/or re-ingestion of the source data (Worboys & Duckham, 2004; Abdalla & Niall, 2007; Dolton & Lowe, 2001; Longley et al., 2001). Popular examples of RDBMS include MySQL and PostgreSQL.

Non-relational, or NoSQL databases, entered the market place in the late 1990's and have been slowly gaining popularity ever since (Penchikala, 2013; Madison et al., 2015). NoSQL databases do not rely heavily on the use of tables, typically don’t use SQL for data manipulations, and work well with enormous amounts of data (Padhy et al., 2011; Moniruzzaman & Hossain, 2013; Bazar & Sebastian, 2014; Madison et al., 2015). With that said, the most notable difference between a NoSQL database and a relational database is that data is stored without the use of a traditional relational schema. Major types of NoSQL databases include key-value stores, column oriented databases, document based stores, and graph databases (Padhy et al., 2011; Moniruzzaman & Hossain, 2013).

The key-value store model, based from a paper written by Amazon in 2007, puts the data in key pairs that are indexed for retrieval, which can hold structured and unstructured data (Perdue, 2016). This is achieved in part using Hash tables. Hash tables, broadly speaking, are data structures used to create an associative array and use a hash function to compute an index that is stored in a table where specified values can be found (USA Patent No. US 7085911 B2, 2006). Searches using this model can only be performed on the key pairs and are limited to exact matches (Madison et al., 2015). The Oracle NoSQL database is an example of a key-value store (Oracle, 2016).

Column oriented databases were created to store and process very large amounts of data over several machines. Data tables are stored in columns, rather than rows, but are otherwise very similar to the common relational database. Predictive analytics and time stamping are functions of these systems making them ideal for analysis and data versioning (Moniruzzaman & Hossain, 2013; Madison et al., 2015). Cassandra is a type of column oriented database (The Apache Software Foundation, 2016).

Document based stores organize data as a collection of documents encoded in a standard data exchange format like XML (eXtensible Markup Language) or JSON (JavaScript Object Notation). Searches can be conducted on both the keys and the values and each document can contain hundreds of attributes of different data types (Perdue, 2016; Madison et al., 2015). MongoDB and MarkLogic are both document based databases (MongoDB, Inc., 2016; MarkLogic Corporation, 2016).

Graph databases became popular in the 1980's and 90's and were an attempt to overcome the limitations of traditional RDBMS, particularly where GIS is concerned. In general graph databases are a collection of nodes and edges where each node represents a conceptual object and each edge represents a relationship (Angles & Gutierrez, 2008; Padhy et al., 2011; Madison et al., 2015). This relationship is fundamental to the graph database model and is best when storing substantial amounts of interconnected data. Neo4j is an example of a graph database (Neo4j, 2016).

Choosing the right spatial database for the task at hand is extremely important (Shekhar & Chawla, 2003; Guting, 1994). Each system has its own advantages and disadvantages that are dependent upon the type of ingested data and the expected outcome of the analysis (Worboys & Duckham, 2004; Dolton & Lowe, 2001). Making the right choice is becoming increasingly difficult as more and more DBMS are adding spatial modules or extensions for use with geospatial data (Van Oosterom et al., 2002). The following is a review of the available literature for MySQL, MongoDB, MarkLogic, Neo4j, and PostgreSQL (with PostGIS) databases focusing on SDBMS comparative analysis.

MySQL is purported to be the most popular open source RDBMS and uses SQL to maintain and query data within the database. This system was originally developed to manage substantial amounts of information faster than the traditional databases available at the time. The most recent version of MySQL (5.7) offers GIS functions and spatial indexes (R-Tree) out-of-the-box with additional extensions that allow users to perform operations on spatial data, such as determining the distance between two objects. Documentation for GIS features and extensions supported are available on the MySQL website which facilitate the generation, storage, and analysis of geographic information (Oracle Corporation, 2016; Karlsson, 2008).

Nair et al. (2015) did a side by side comparison of MySQL, PostgreSQL (with PostGIS), and SpatialLite, all open source RDBMS, and concluded that MySQL performed best when used with web applications but lacked in stability, raster support, and spatial features (Nair et al., 2015). With that said, the spatial features that MySQL does support have very fast query executing times as was pointed out in an analysis conducted by Zhou et al. (2009). In this study, they compared the query speeds of MySQL to PostgreSQL (with PostGIS), Oracle Spatial, and IBM DB2 Spatial Extender, other popular open-source and commercial databases (Zhou, et al., 2009). When MySQL was compared to SQL Server, a commercially supported RDBMS, to determine which had better query processing times, the results were in favor of SQL Server (Amlanjyoti et al., 2015). The query execution time was measured as a performance metric in both the

Zhou et al., and Amalanjyoti et al., analysis, however, only one of these studies used a geospatial dataset. In addition, the ingestion time and storage and memory footprint were only loosely captured in the future research section of the Amalanjyoti et al. analysis (Amlanjyoti et al., 2015; Nair, Chauhan, & Vats, 2015).

PostgreSQL is another mature open-source RDBMS that utilizes a structured query language. It has no limitations on the size of the database or the number of rows and indexes per table (The PostgreSQL Global Development Group, 2017). It is also highly customizable and can run stored procedures in a plethora of programming languages which include Java, Python, and its own PL/pgSQL. PostGIS is one of the features offered by PostgreSQL which provides support for geographic objects that are used to create a spatial database for GIS like ESRI's Spatial Database Engine (The PostgreSQL Global Development Group, 2017).

Miler et al. (2013) compared the performance of Dijkstra's shortest path calculation using Neo4j and PostGIS to determine if there was any difference in calculation time using road data from OpenStreetMap (Miler, Medak, & Odobasic, 2013). They hypothesized that the graph database (Neo4j) would be the better choice for this type of calculation however that was not the case. They determined that Neo4j was not suitable for the shortest path algorithm because it uses a full graph traversal which takes up substantial amounts of memory (Miler, Medak, & Odobasic, 2013). In this study PostgreSQL (with PostGIS) had both lower peak memory consumption and faster hot and cold query times.

Another open-source option is MongoDB which differs from MySQL and PostgreSQL because it is a NoSQL, document based, database. Rather than store data in tables like relational databases, MongoDB uses collections of fields and values, in a structured BinaryScript Object Notation (BSON) format. Standard SQL is not supported by MongoDB; however, it does support a rich query text of its own as well as JavaScript. Queries can consist of a mix of non-JavaScript and JavaScript code in the same instance. Geospatial indexes and query tools are available to analyze spatial data. Further documentation can be found on their website (MongoDB, Inc., 2016).

A study conducted by Bazar & Sebastian (2014) compared popular open-source, NoSQL, databases to aid readers in transitioning from a traditional RDBMS to a NoSQL solution. One of the databases in this study was MongoDB. The other two databases in this study were Couchbase, similar to MongoDB as it is another document-based database, and Cassandra, a column oriented database. The analysis concluded that MongoDB processed requests faster than Cassandra but slower than Couchbase even though they all showed approximately equal read speeds (Bazar & Sebastian, 2014). In a similar analysis comparing MongoDB to MySQL, Kumar et al. (2015) found that MongoDB had data processing speeds that were much faster than MySQL. In addition, Aghi et al. (2015) found that MongoDB performed better than MySQL when there were complex queries especially when they involved multiple joins. Query execution times, data ingestion, and memory footprints were evaluated in these studies but weren’t specific to geospatial data or spatial queries.

MarkLogic is a commercially supported, document based, NoSQL database that provides storage for many data types including JSON, XML, and geospatial objects. Structured and unstructured data, as well as any pertinent metadata, are stored in the same database (MarkLogic Corporation, 2016). Although MarkLogic was released in 2001, there are no apparent peer reviewed database comparative analysis available. With that said, there are blog posts available that compare the MarkLogic product to other similar databases, such as MongoDB, as well as highlight the overall benefits of using MarkLogic but these are based on opinion and lack unbiased scientific discovery (Fowler, 2013).

Neo4j is a NoSQL graph database that contains a spatial extension library. This library provides spatial indexes that allow users to search their data for objects within a certain distance (proximity) or within a specified area (Bass, 2012; Neo4j, 2016). The database is queried using the Cypher Query Language, a recent addition to the Neo4j platform (Jaiswal & Agrawal, 2013; Batra & Tyagi, 2012).

Batra & Tyagi (2012) conducted a comparative analysis of MySQL and Neo4j to showcase graph databases as a replacement for traditional RDBMS when dealing with large datasets that need a dynamic schema. They found that Neo4j could retrieve data at a much faster rate than MySQL and the schema for Neo4j was more flexible as new relationships could be added without the need for restructuring (Batra & Tyagi, 2012). Jaiswal & Agrawal (2013) also compared Neo4j to MySQL and, similar to Batra & Tyagi (2013), determined that the graph database outperformed the RDBMS in query retrieval time. While these studies looked at query performance and retrieval times they were not specific to geospatial data.

This thesis will assist the GIS community by evaluating the spatial competency of MySQL, MongoDB, MarkLogic, Neo4j and PostgreSQL (with PostGIS) databases when used with a vector dataset. Overall the literature review showed gaps in the lack of comparative analysis available for these databases using geospatial data. Although some literature is available on query performance there was little to none for storage and memory footprint, ingest performance, and the complexity, usability, and accuracy of the database. There was no peer reviewed literature for MarkLogic. In some cases, such as Neo4j, the range or type of database used to conduct the comparative analysis was limited, e.g. Neo4j vs MySQL. Almost all the studies reviewed emphasized the need for future comparative research on other SDBMS largely because there are many to choose from and each has its own pros and cons. The following will evaluate each selected database and provide valuable information to assist users in making the right SDBMS choice for their data.

CHAPTER THREE: METHODOLOGY

Research was performed by initializing the five selected databases and comparing them to one another. The same geospatial (vector) dataset and spatial queries were used for the analysis. Further information on the data used in this analysis is available in the *About the Data* section. The five databases chosen to conduct this comparative analysis were MySQL, MongoDB, MarkLogic, Neo4j, and PostgreSQL (with PostGIS). Table 1 provides a reference guide to each database and its respective model. Table 2 lists the version, architecture, and install/download size.

Table 1: Quick reference guide to the analyzed database and its respective model.

Database	Open Source	Commercially Supported	RDBMS	NoSQL (Non-Relational)	Graph Database
MySQL	X		X		
MongoDB	X			X	
MarkLogic		X		X	
Neo4j				X	X
PostgreSQL	X		X		

Table 2: Listing of the version, architecture, and install size of each database into the virtual machine.

Database	Version	Architecture	Install Size
MySQL	Community Server 5.7.17-1	64 bit	202 MB
MongoDB	3.4.2 for Redhat Enterprise Linux 7	64 bit	257 MB
MarkLogic	For CentOS 7 8.0-6.1	64 bit	193 MB
Neo4j	Community Edition 3.1.2	64 bit	99 MB
PostgreSQL	9.6.3 with PostGIS 2.3.2 r15302	64 bit	104 MB

To provide a controlled environment, a single virtual machine was created and cloned for each database type. The VM hosting platform used was VirtualBox version 5.1.14. The parameters for the virtual machine image are described in Table 3.

Table 3: Listing of parameters for Virtual Machine Configurations.

Parameter	Value
Processor	Dual-Core with VT-x hardware support
RAM	8192MB
Storage	32GB
Network Interface	Bridged to host adapter, 1GB

The Operating System installed on the VM image was CentOS Linux release 7.3.1611. For simplicity, both SELinux and the firewalld process were disabled on the image before cloning. After cloning the image, the database systems were installed, and the tests were performed.

Loading data into a database can typically be done in several ways. For the purposes of this analysis data ingestion was performed using the most common method for each system. These methods are explained in detail below per database.

Installation, Configuration, and Ingestion

MarkLogic

MarkLogic was installed using yum via the RPM package obtained from the MarkLogic website. The command used to install the product was:

```
#yum install MarkLogic-RHEL7-8.0-6.1.x86_64.rpm
```

After installation, the initial configuration was performed automatically. MarkLogic is configured and managed via a web interface. Using this interface, a geospatial element pair index was created on the Documents database prior to loading the data. MarkLogic offers a tool called the MarkLogic Content Pump for ingesting data. This tool was used to parse the CSV file and insert the data into the Documents database. The following command was run to load the data into MarkLogic:

```
#./mlcp.sh import -host localhost -port 8006 -username admin -password ###### \
-input_file_type delimited_text -document_type json -input_file_path /tmp/mrds.csv
```

MarkLogic can execute 2 types of queries: ad-hoc and stored. Stored queries are typically inserted into a modules database within MarkLogic and run via calling a web service or invoked via an ad-hoc query. Ad-hoc queries are run via a web interface that is built into MarkLogic called QConsole.

After the data was ingested, a transformation was run on all the documents in order to extract the latitude and longitude values into a usable format for the range index created previously. This was a three-step process. First, a stored module was created that contained logic to produce a point property from the latitude and longitude properties stored in the documents. This module was then loaded into the modules database for execution. Finally, an ad-hoc query was run to apply the transformation module against every document. This process is detailed below:

1. Stored Transformation Module:

```
declareUpdate();
function createGeoPoint(doc) {
```

```
  if (doc.latitude && doc.longitude) {
    doc.point = {latitude: parseFloat(doc.latitude), longitude:
parseFloat(doc.longitude)};
  }
  return doc;
}
var doc = cts.doc(uri);
var docObject = doc.toObject();
xdmp.nodeReplace(doc, createGeoPoint(docObject));
```

2. Load the transformation module into the modules database (executed from QConsole)

```
// Load the transformation Module
declareUpdate();
xdmp.documentLoad('/tmp/createGeoPoint.sjs', {uri: '/createGeoPoint.sjs', permissions:
xdmp.defaultPermissions()});
```

3. Run the transformation module against every document (Executed from QConsole)

```
for (var uri of cts.uris(null, null, cts.trueQuery())) {
 xdmp.spawn(
   '/createGeoPoint.sjs',
   {uri: uri},
   {transactionMode: 'update-auto-commit'}
 );
}
```

MySQL

MySQL was installed using yum directly from the preconfigured repositories in CentOS:

```
#yum install mysql-community-server
```

To interface with MySQL, the tool MySQL Workbench 6.3 Community Edition was installed on the host machine and configured to connect to the MySQL instance running within the guest VM. After installation and startup, a spatial index was created by running a query in MySQL Workbench. Next, the data was loaded by running a second query. Finally, a transformation was run to synthesize point fields for each row to use with the MySQL spatial index. The process is detailed below:

1. Create spatial index

```
ALTER TABLE mrds.mrds ADD SPATIAL INDEX coords_index (coords);
```

2. Ingest data into MySQL

```
LOAD DATA INFILE '/var/lib/mysql-files/mrds.csv'
INTO TABLE mrds.mrds
FIELDS TERMINATED BY ','
	OPTIONALLY ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 LINES
(dep_id,url,mrds_id,mas_id,site_name,@vlat,@vlon,region,country,state,county,com_type,commod1,commod2,commod3,oper_type,dep_type,prod_size,dev_stat,ore,gangue,other_matl,orebody_fm,work_type,model,alteration,conc_proc,names,ore_ctrl,reporter,hrock_unit,hrock_type,arock_unit,arock_type,structure,tectonic,ref,yfp_ba,yr_fst_prd,ylp_ba,yr_lst_prd,dy_ba,disc_yr,prod_yrs,discr)
	SET latitude = nullif(@vlat,''),
	longitude = nullif(@vlon,'');
```

3. Synthesize point fields

```
UPDATE mrds.mrds SET coords = GeometryFromText( CONCAT( 'POINT(', longitude, ' ', latitude, ')' ) );
```

Neo4j

Neo4j was extracted and run directly from its source package:

```
#tar xf /tmp/neo4j-community-3.1.2-unix.tar.gz
```

In order to utilize spatial capabilities, the Neo4j spatial library (Release 0.24) was installed. The installation process for neo4j-spatial involves building the library from source (via Maven) and then copying the compiled jar file into the Neo4j plugin directory. Maven was installed on the VM via the preconfigured CentOS yum repository, and the spatial plugin was built using the command:

```
#mvn install
```

This produced a jar file that was copied into the Neo4j plugin directory. Neo4j comes with a built-in web interface called Neo4j Browser for running ad-hoc queries against the database. This interface was used for loading the data and running queries. The loading and transformation process for Neo4j consisted of running an initial load query, followed by running a query to produce the geospatial layer necessary for utilizing the Neo4j-spatial plugin. These queries are detailed below:

1. Load the data into Neo4j

```
USING PERIODIC COMMIT 10000
LOAD CSV WITH HEADERS FROM "file:/tmp/mrds.csv" AS row
CREATE (:Resource {dep_id: row.dep_id, url: row.url, mrds_id: row.mrds_id, mas_id:
row.mas_id, site_name: row.site_name, latitude: toFloat(row.latitude), longitude:
toFloat(row.longitude), region: row.region, country: row.country, state: row.state, county:
row.county, com_type: row.com_type, commod1: row.commod1, commod2:
row.commod2, commod3: row.commod3, oper_type: row.oper_type, dep_type:
row.dep_type, prod_size: row.prod_size, dev_stat: row.dev_stat, ore: row.ore, gangue:
row.gangue, other_matl: row.other_matl, orebody_fm: row.orebody_fm, work_type:
row.work_type, model: row.model, alteration: row.alteration, conc_proc: row.conc_proc,
names: row.names, ore_ctrl: row.ore_ctrl, reporter: row.reporter, hrock_unit:
row.hrock_unit, hrock_type: row.hrock_type, arock_unit: row.arock_unit, arock_type:
row.arock_type, structure: row.structure, tectonic: row.tectonic, ref: row.ref, yfp_ba:
```

```
row.yfp_ba, yr_fst_prd: row.yr_fst_prd, ylp_ba: row.ylp_ba, yr_lst_prd: row.yr_lst_prd,
dy_ba: row.dy_ba, disc_yr: row.disc_yr, prod_yrs: row.prod_yrs, discr: row.discr});
```

2. Construct a geospatial layer containing all the records in the dataset.

```
MATCH (r:Resource) WHERE r.latitude is not null and r.longitude is not null
WITH r
CALL spatial.addNode("layer_resources", r) YIELD node as n
RETURN COUNT(*) as cnt;
```

Of significance, this step took over 10 hours to complete.

MongoDB

MongoDB was installed directly in CentOS via the preconfigured yum repository system:

```
#yum install mongodb-org
```

MongoDB provides a tool called mongoimport for ingesting data. This tool was used to parse the CSV file and insert the data into the mrds collection within the local database. The following command was run to load the data into MongoDB:

```
#mongoimport -d local -c mrds --type csv --file /tmp/mrds.csv –headerline
```

Queries in MongoDB were run via a tool called Robo 3T, a GUI interface for managing and querying MongoDB. In order to make use of MongoDB's geospatial indexes, a field was synthesized in each record to hold the geospatial data in the format [longitude, latitude] by running the following query:

```
db.mrds.find().forEach(function(row) { if (row.latitude && row.longitude) {row.point =
[row.longitude, row.latitude]; } db.mrds.save(row); });
```

A text index was created on the com_type field for use in Queries 1 and 3:

```
db.mrds.createIndex( { com_type: “text” }, { sparse: true } );
```

Finally, a geospatial index was created on the point field constructed above:

```
db.mrds.createIndex( { point: "2dsphere" }, { sparse: true } );
```

PostgreSQL

PostgreSQL and PostGIS were both installed directly in CentOS via the preconfigured yum repository system:

```
#yum install postgresql96-server.x86_64
#yum install postgis2_96.x86_64
```

For interacting with PostgreSQL, the open-source tool pgAdmin4 was used. The tool provides mechanisms for configuring and connecting to PostgreSQL databases, as well as executing queries and loading data. The following query was run to create a new table:

```
CREATE TABLE public.mrds
(
	dep_id character varying,
	url character varying,
	mrds_id character varying,
	mas_id character varying,
	site_name character varying,
	latitude character varying,
	longitude character varying,
	region character varying,
	country character varying,
	state character varying,
	county character varying,
	com_type character varying,
```

```
	commod1 character varying,
	commod2 character varying,
	commod3 character varying,
	oper_type character varying,
	dep_type character varying,
	prod_size character varying,
	dev_stat character varying,
	ore character varying,
	gangue character varying,
	other_matl character varying,
	orebody_fm character varying,
	work_type character varying,
	model character varying,
	alteration character varying,
	conc_proc character varying,
	names character varying,
	ore_ctrl character varying,
	reporter character varying,
	hrock_unit character varying,
	hrock_type character varying,
	arock_unit character varying,
	arock_type character varying,
	structure character varying,
	tectonic character varying,
	ref character varying,
	yfp_ba character varying,
	yr_fst_prd character varying,
	ylp_ba character varying,
	yr_lst_prd character varying,
	dy_ba character varying,
	disc_yr character varying,
	prod_yrs character varying,
	discr character varying,
	PRIMARY KEY (dep_id)
)
WITH
(
	OIDS = FALSE
);

ALTER TABLE public.mrds
OWNER
to
postgres;
```

After the table was created, the data from the csv file was loaded into the table by running

```
COPY mrds FROM '/tmp/mrds.csv' WITH DELIMITER ',' CSV HEADER;
```

Once the data was loaded, the latitude/longitude fields needed to be synthesized into a geography data type to take advantage of PostGIS indexes. A new column “pointgeo” with type “geography” was added to the “mrds” table and an index was added on the column via the pgAdmin4 graphical interface. Finally, the latitude/longitude fields were parsed to construct the “pointgeo” geography within the table.

```
update mrds
set pointgeo = st_geogfromtext('SRID=4326;POINT(' || longitude || ' ' || latitude || ')');
```

Key Metrics

The five systems have been analyzed by way of both qualitative and quantitative methods. Ingest performance, query performance, accuracy, and storage and memory footprint have been quantitatively measured while usability and complexity were assessed subjectively. The strategy included:

1. Installing the databases on identical virtual machines.
2. Loading the same dataset into each management system.
3. Running the same predefined set of queries against each database.
4. Analyzing the query outputs for accuracy (it might be possible that differences in query languages and or styles could cause the system to return a different number of results).

Table 4 below further details these metrics and how they have been measured.

Table 4: Description of the evaluation metrics

Metric Name	Measurement Unit	Description of Measurement
Ingest performance	Seconds	How long does it take to load the entire dataset? Are there extra steps to loading (pre or post-processing)?
Storage	Bytes	How much space does the loaded database consume on disk?
Query performance	Seconds	Data retrieval time. How long does each query take to resolve the results? Provide wait analysis and graphs.
Accuracy	Number of records returned	Do all the databases provide the expected query results?
Usability	Qualitative description of user experience	Were there any other factors that made one database easier to use than another?
Complexity	Lines of query, number of processes for each database used, and available documentation	How difficult is it to query for data? Do some databases require more complex queries to achieve the same results (using the same objective and instruction)?

Measuring each of these metrics relied on the instrumentation provided by each individual database and tool. For example, MongoDB provides a tool called mongoimport for loading data, that displays its runtime in its program output. MarkLogic's mlcp tool also displays its runtime as program output, but appears to round the time value to the nearest second. For Neo4j, MySQL, and PostgreSQL, loading was performed by executing ad-hoc queries against each database, and the query runtime was recorded by the Neo4j Browser, MySQL Workbench, and pgAdmin4 respectively.

Likewise, storage size measurement relied on the tools provided. Storage size for MarkLogic was taken from its administration interface. Neo4j storage was recorded from the Neo4j Browser. MySQL, PostgreSQL, and MongoDB storage values were recorded from the operating system measurement of the database directory size on disk.

Usability is a subjective measurement that was derived from the amount of effort required to construct each query or transform the input or output from a system. The more preprocessing and data manipulation required to execute a query or transform a dataset, the less usable a system is considered. Other considerations for usability include toolsets, documentation, and community support (access to online resources for training and reference material).

About the Data

The dataset used was the US Geological Survey's (USGS) Mineral Resource Data System (MRDS). It contains records about mineral resources, such as the type, location, reporter, site name, discovery year, and more. It is available online here https://mrdata.usgs.gov/mrds/. The original publication date for this dataset was 2005 and it was last updated in March of 2016. The dataset contains 304,633 total records with 44 heterogenous fields including text, scalar values, and spatial data (latitude/longitude).

Querying the Data:

A database may have many simultaneous operations occurring at any given time, which can cause minor variations in the performance of a query at a given moment. Likewise, the operating system may have intermittent maintenance and housekeeping tasks that can affect processing performance from one moment to the next. Compounding this variance, most database systems employ a caching mechanism that provides for improved performance of frequently run queries. After a query is executed, the partial results from the execution are maintained in cache to provide faster access for subsequent runs. Queries that are assisted by this cache are generally referred to as "warm" queries, and queries that occur with no cache assistance are referred to as “cold” queries. To take these variables into account, each query was executed a total of 10 times, 5 immediately after a database restart (to measure performance with an empty cache), and 5 executed in immediate succession. The results of these trials were averaged for the conclusive results detailed below. This methodology was followed to remove any minor variances in performance across trials due to external influences.

The spatial queries used for performance measurements are defined below and will be notated throughout this thesis by the corresponding number (e.g. Query 1):

1. Find all records with the attribute type of “non-metal.”
2. Find all records within a specified geometry. This was manually conducted for10 different regions (Refer to Figure 1 for an illustration of the geometries queried).
3. Find all records of type “non-metal” within a defined geometry. This was manually conducted for 10 different regions (Refer to Figure 1 for an illustration of the geometries queried).
4. Find all records within 5 miles of the Potomac River in Washington DC. (Refer to Figure 2 for a detailed map view of the defined space).
5. Find all records within 1 mile of Uranium deposits. (For a detailed view of the Uranium deposit locations refer to Figure 3).

These five queries were formulated to test different properties of each DBMS, ranging from basic, non-spatial information retrieval, to more complex geospatial queries. Query 1 is a basic attribute query, without any geospatial properties. Query 2 is a simple geospatial geometry query. Query 3 is a combination of queries 1 and 2. Query 4 is a complex polygon geospatial query. Lastly, Query 5 is a 2-part query, using the output of the initial attribute query to dynamically construct a geospatial query.

Because each of the databases use a different query language, the methods for querying data differed substantially. MarkLogic and MongoDB both use JavaScript as their query language, but each provides a separate set of extensions and support functions for executing queries. MySQL and PostgreSQL use SQL as their querying languages, with some geospatial-specific language extensions and features for querying spatial data. Neo4j uses Cypher as its query language, which is similar to SQL but with some features that enhance the ability to query multi-level relationships within a connected graph.

Query 1 was the simplest query of the set, and therefore the most logically consistent query across all the databases. This query serves as a baseline for simple data retrieval within the DBMS.

The geometries for Queries 2 and 3 were produced by drawing 10 bounded areas, 5 rectangles and 5 polygons, each randomly chosen in separate geographic regions within the United States using Google Earth. These latitude and longitudes were recorded, and the resulting geometries were used in the queries for all 5 databases using their respective languages.

The geometries for Query 4 were constructed by producing a KML file using Google Earth. A line was drawn along the center of the Potomac river within Washington D.C. and a 5-mile buffer was applied to the line, and the output was saved into a KML file. MarkLogic and PostgreSQL with the PostGIS extension could automatically load the geometry within the KML file and use it as part of the query. All other databases required extracting the KML file as text, and then constructing the appropriate geometries as strings that the candidate database would understand. This process required a significant amount of time and effort and is typical of the workflow of a geospatial analyst.

The first part of Query 5 returned a result set that contained all the mineral deposits with a primary commodity type of Uranium. The query then used the resulting latitudes and longitudes from this set to dynamically construct a geospatial query of a 1-mile radius circle around every item. Because each database represents distances differently and expects different geometries to represent a point buffer (circle), this query had the most inconsistent logic across all the databases. Figure 3 illustrates the first part of this query highlighting the locations of all the Uranium deposits within the Continental United States.

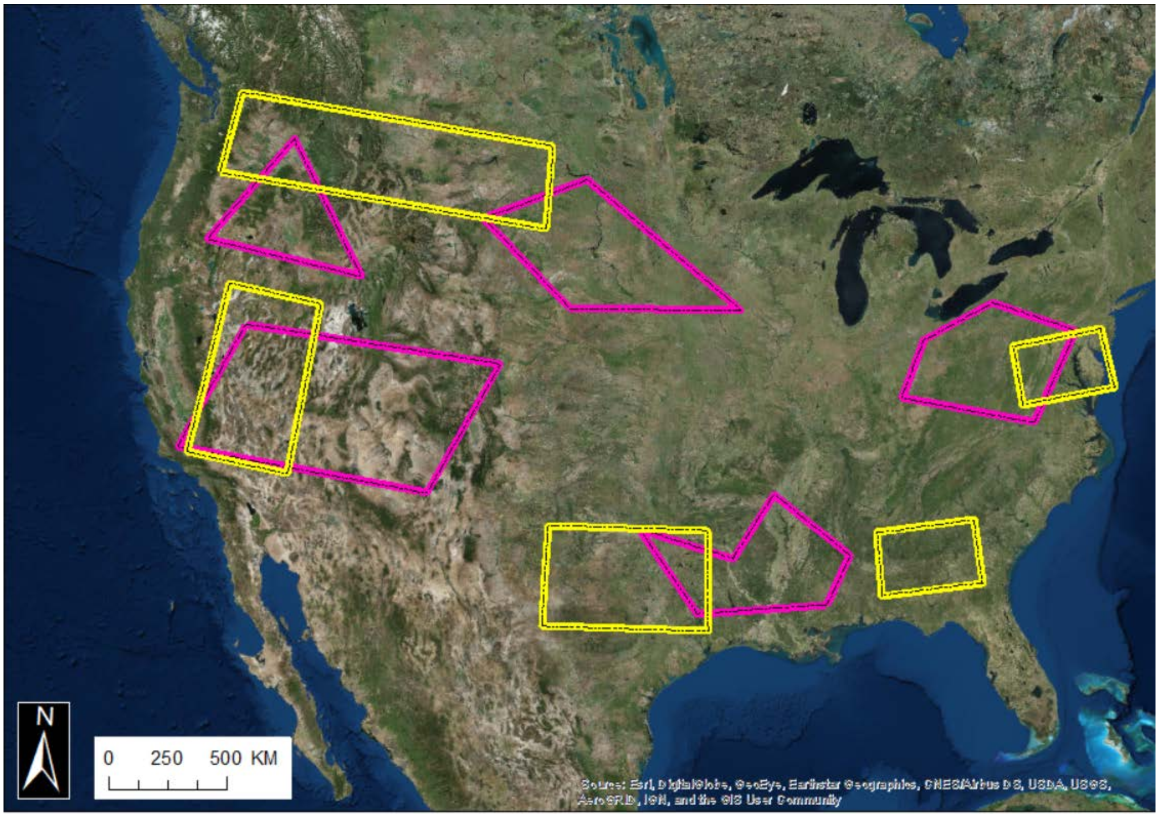

Figure 1: ArcMap image of the 10 manually defined geospatial boundaries used for queries 2 and 3.

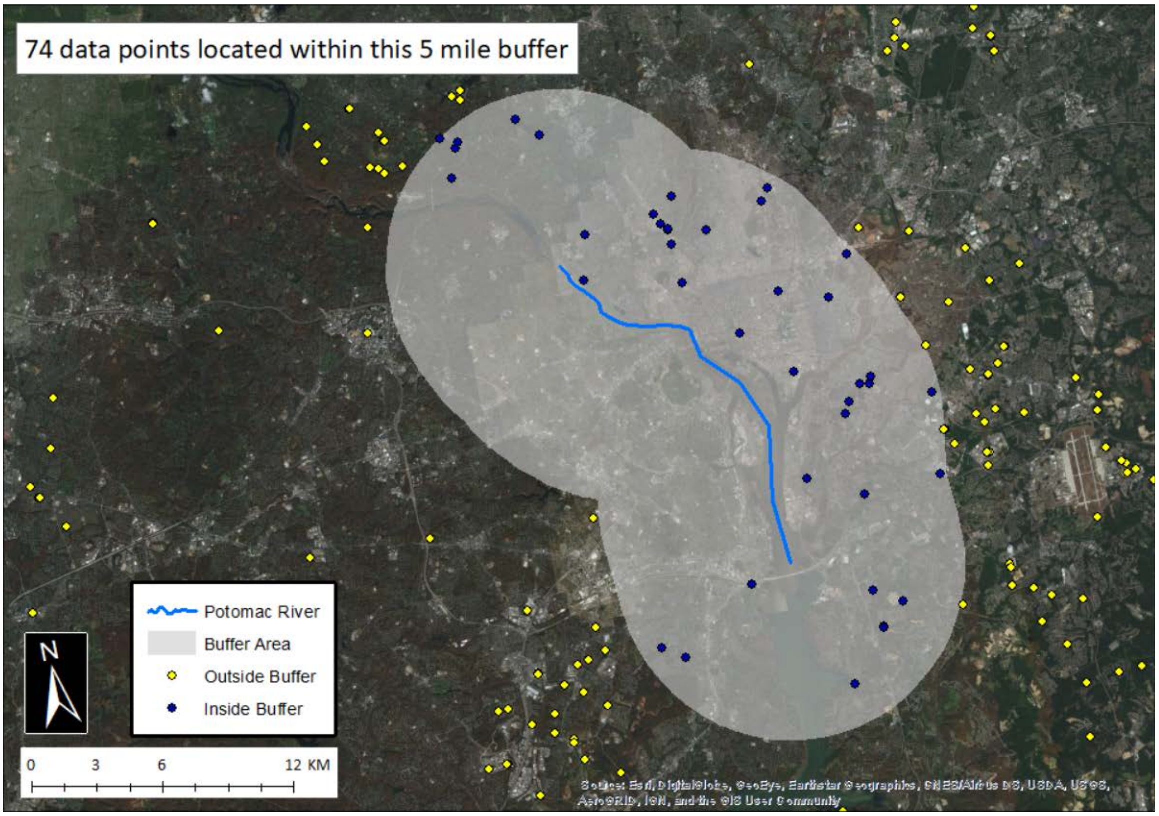

Figure 2: ArcMap image showing the 5-mile buffer area of interest used for query 4.

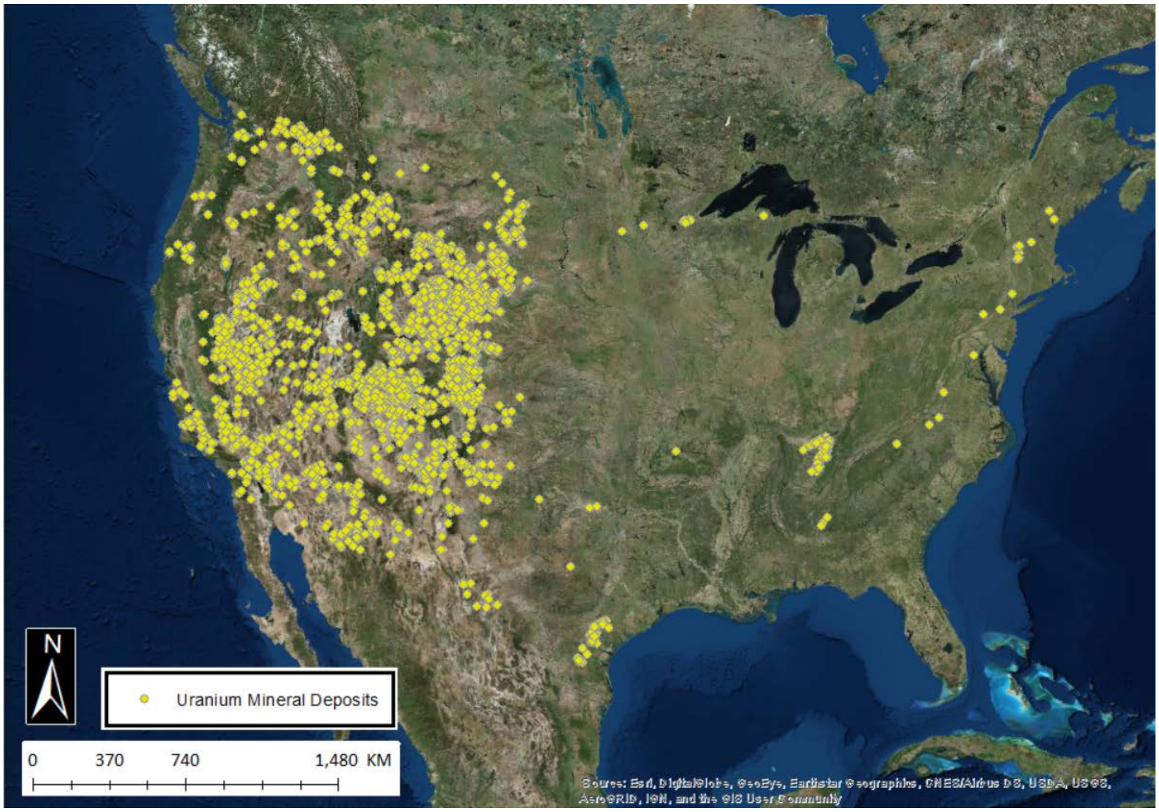

Figure 3: ArcMap image of the locations of Uranium deposits from the interim output of query 5.

RESULTS

Ingestion and Storage

Based on the reviewed literature, it was deemed likely that there would be significant differences in the data ingestion time in each of the different database systems. It was also expected that the data within each of the spatial databases would have different storage and memory footprints after ingesting the same dataset. This anticipated difference would occur because all five databases employ vastly different data structures for storing information. These different data structures influence the size of the stored data, as well as the performance of data retrieval.

The first stage of this comparative analysis consisted of loading the preprocessed geospatial data into the respective databases to measure ingest performance and the overall size of the database (storage and memory footprint). As predicted, there were significant differences in the amount of time each database took to load the same dataset with a maximum ingest time of 108s with MarkLogic and a minimum time of 3s with PostgreSQL. Figure 4 further details these differences in data ingestion time per database. Likewise, there were large variations in the resulting storage size for each system with a maximum of 1901MBs for MarkLogic and a minimum of 177MBs for MySQL, as detailed in Figure 5. The data loading times tended to correlate with the resulting database size, with larger database sizes linked to longer ingestion times. This distinction will be further discussed in the *Conclusion and Future Research* section of this Thesis.

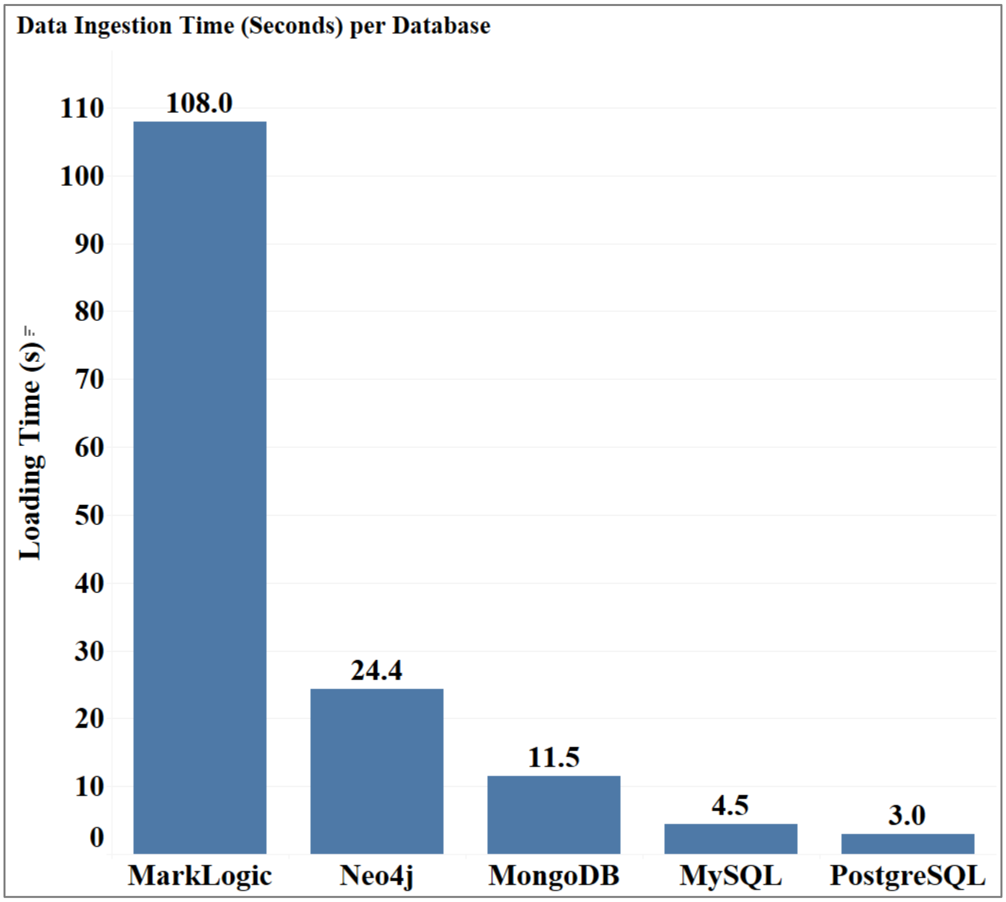

Figure 4: Data ingest time (seconds) for each database to load the same dataset.

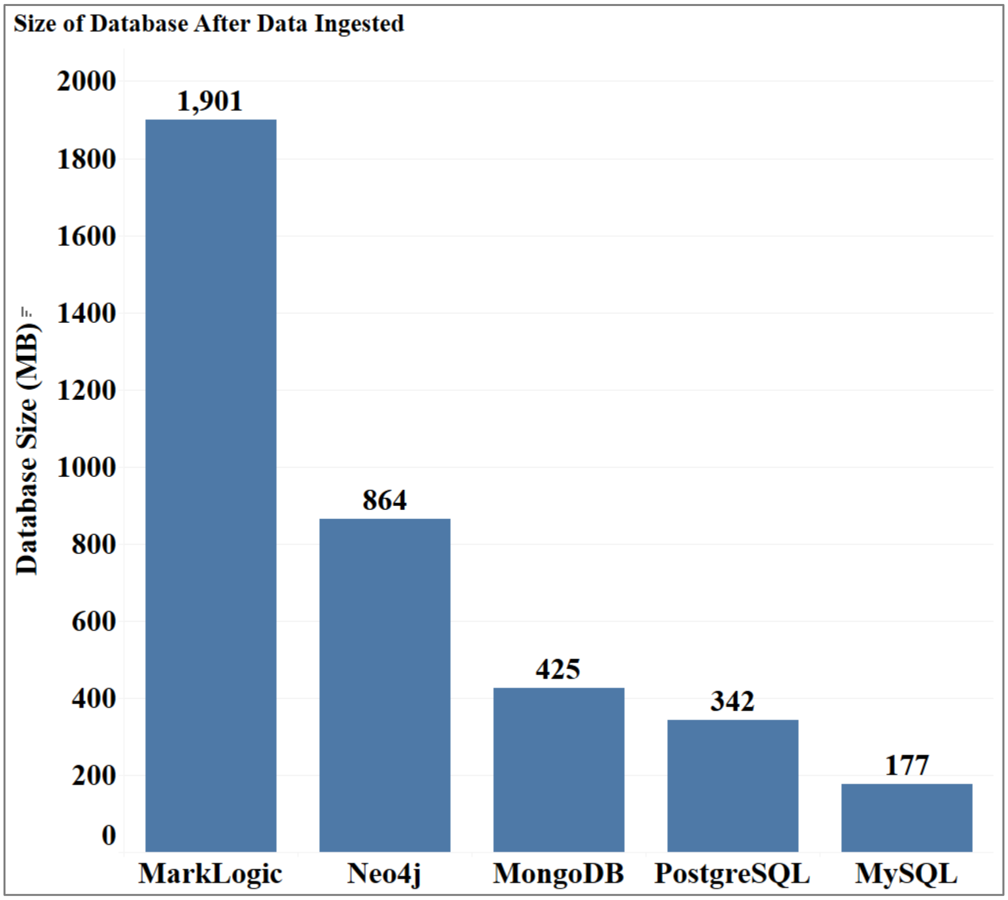

Figure 5: Size (MB) of each database after the same dataset was loaded.

Query Performance

It was anticipated that there would be significant differences in query performance, measured by their execution times, between each of the spatial databases compared. As noted in the *Ingestion and Storage* section, the data structures a database uses affects the query and retrieval performance of a DBMS. Since each one of the candidate databases utilized different indexing mechanisms, it was estimated that they would perform differently under different scenarios, with some being better suited at certain types of queries than others.

As anticipated, all five systems demonstrated substantial variations in query execution times. Table 5 shows the discrete results of the cold and warm queries run against each database apart from queries 2 and 3 where the computed average of the average cold and warm performance times for all the 10 bounding geometries are documented. Table 6 aggregates these values into the average overall runtimes per query per database. A full list of discrete query runtimes is available in the Appendix section of this thesis.

These results show that MarkLogic was the fastest performing database among the group across all 5 queries. MySQL had the second fastest retrieval performance for queries 2, 3, and 4 while PostgreSQL and MongoDB came in second for query 1 and query 5 respectively. MongoDB had the third fastest performance for queries 1, 4, and 2 along with MySQL for query 5 and PostgreSQL for query 3. For queries 2, 4, and 5 PostgreSQL had the fourth fastest times along with MySQL for query 1 and MongoDB for query 3. Neo4j consistently required longer query processing times for all 5 queries executed.

All five databases were able to complete all five of the defined queries although performance times varied significantly between databases. Query 5 had the largest variance across all the databases observed with a minimum runtime of 0.055s with MarkLogic and a maximum of 1585.9s, approximately 26 minutes, with Neo4j. These and other outcomes are further illustrated in Figures 6-10. Each query was defined previously under the *Methodology* section and will be noted using the same numerical key. The full query text for each query performed can be found in the *Appendix* section of this thesis.

Table 5: Discrete Query Performance Results (time in seconds) Query 2 and 3 are an average of the average cold and warm run times for all 10 geometries queried. This is done for simplicity, but Table 15 in the Appendix section provides an entire detailed list of all query run times.

Database	Query #	Query Time (cold)	Query Time (warm)
MarkLogic	1	0.001954	0.0012574
	2	0.001585	0.000712
	3	0.002306	0.001224
	4	0.0121788	0.0104978
	5	0.06227	0.0474812
MongoDB	1	0.026	0.0184
	2	0.10742	0.04246
	3	0.35222	0.15742
	4	0.071	0.057
	5	740.961	769.7186
MySQL	1	0.024875	0.024839
	2	0.05692351	0.039764865
	3	0.043668845	0.027020345
	4	0.036066	0.032606
	5	783.5784	786.5094
Neo4j	1	0.9546	0.363
	2	1.27264	0.46942
	3	1.31064	0.51334
	4	8.9392	6.5932
	5	1666.435	1505.4304
PostgreSQL	1	0.010747	0.008265
	2	0.11455298	0.11076532
	3	0.2778676	0.08241374
	4	3.39914	3.333564
	5	1303.978	1462.89

Table 6: Average runtime (seconds) for the overall (cold and warm) execution time for each query per database. Query 2 and 3 are an average of the average cold and warm run times for all 10 geometry queries.

Query #					
Database	**1**	**2**	**3**	**4**	**5**
MarkLogic	0.0016	0.0011	0.0018	0.0113	0.0549
MongoDB	0.0222	0.0749	0.2548	0.0640	755.3398
MySQL	0.0249	0.0483	0.0353	0.0343	785.0439
Neo4j	0.6588	0.8710	0.9120	7.7662	1,585.9327
PostgreSQL	0.0095	0.1127	0.1801	3.3664	1,383.4340

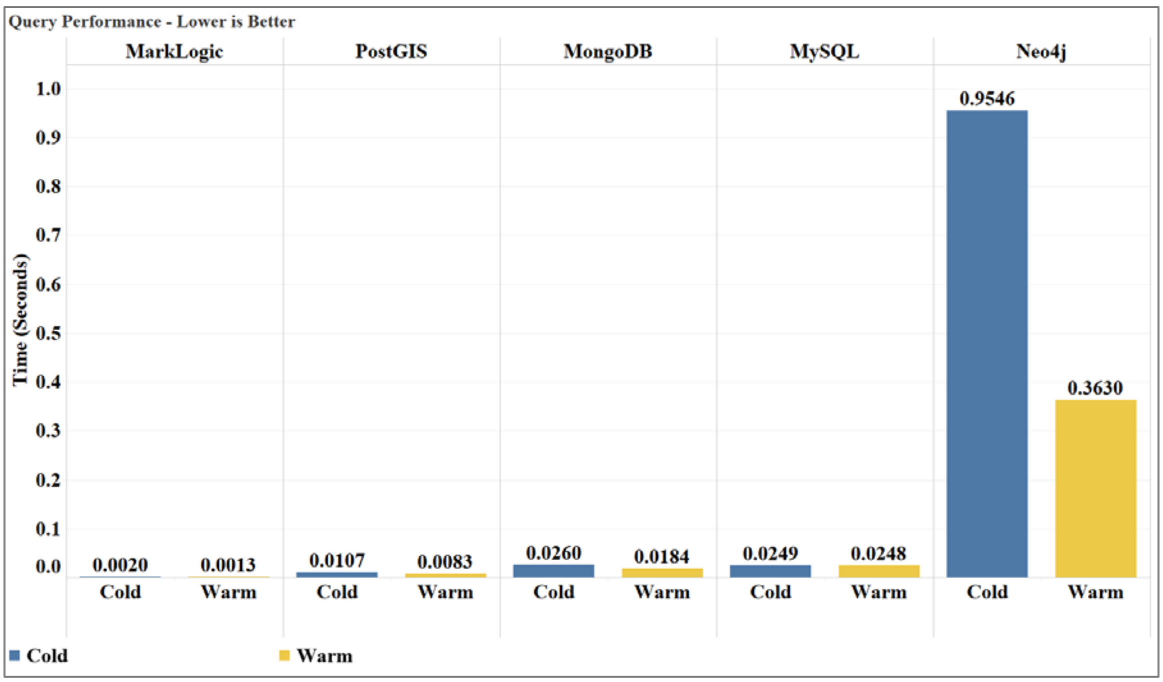

Figure 6: Query Time (cold) in blue and Query Time (warm) in yellow for Query 1. Numbers shown are the time needed to process the query in seconds.

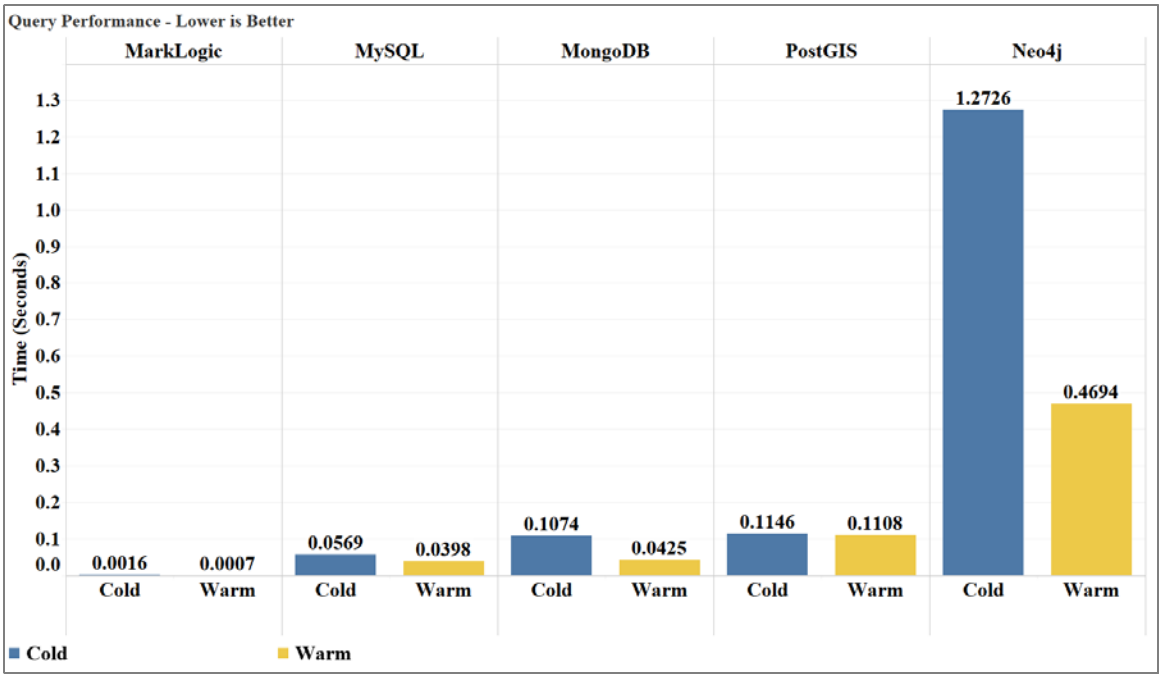

Figure 7: Query Time (cold) and Query Time (warm) for Query 2. Numbers shown are the time needed to process the query in seconds.

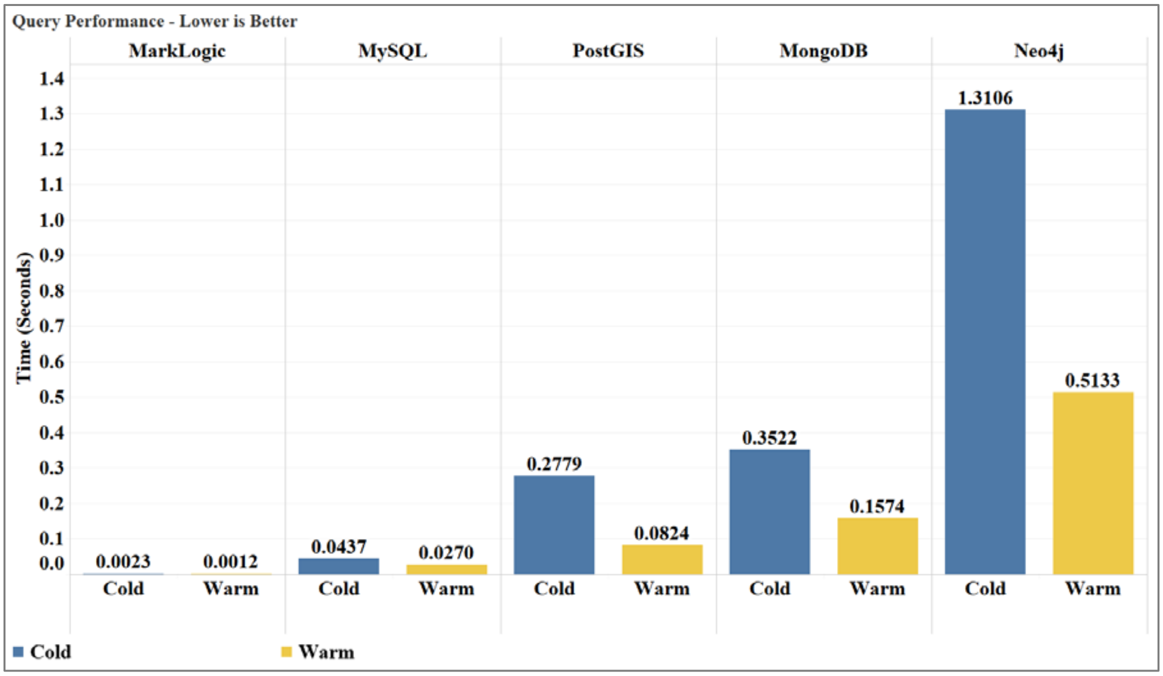

Figure 8:Figure 8: Query Time (cold) and Query Time (warm) for Query 3. Numbers shown are the time needed to process the query in seconds.

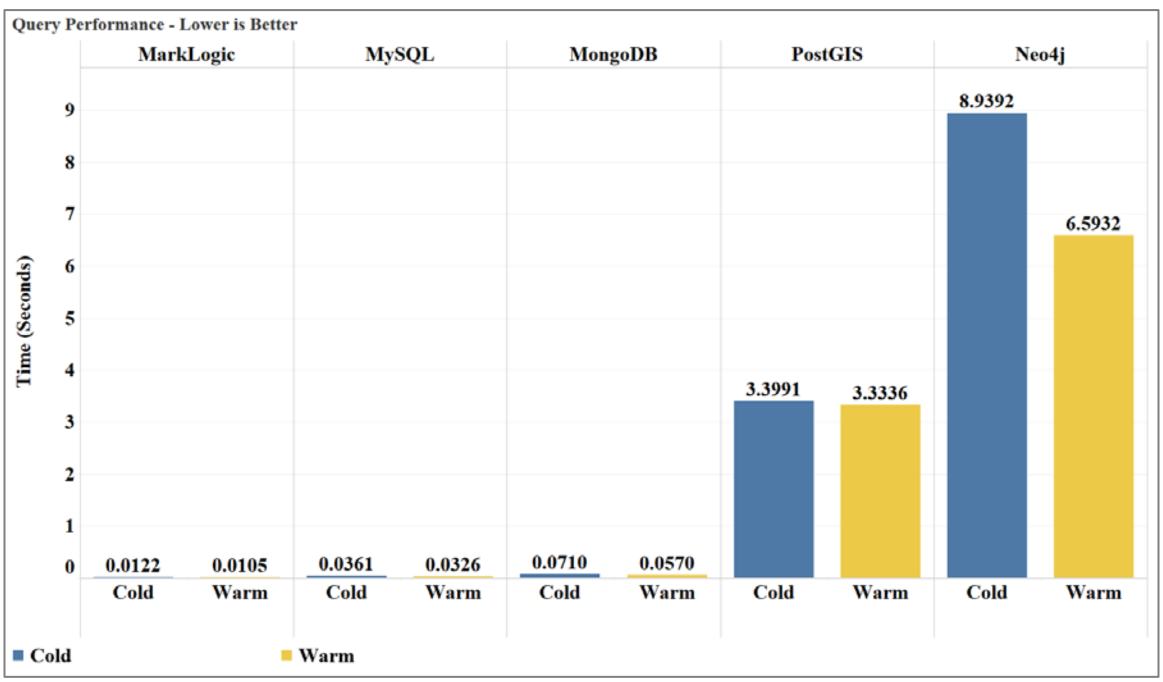

Figure 9: Query Time (cold) and Query Time (warm) for Query 4. Numbers shown are the time needed to process the query in seconds.

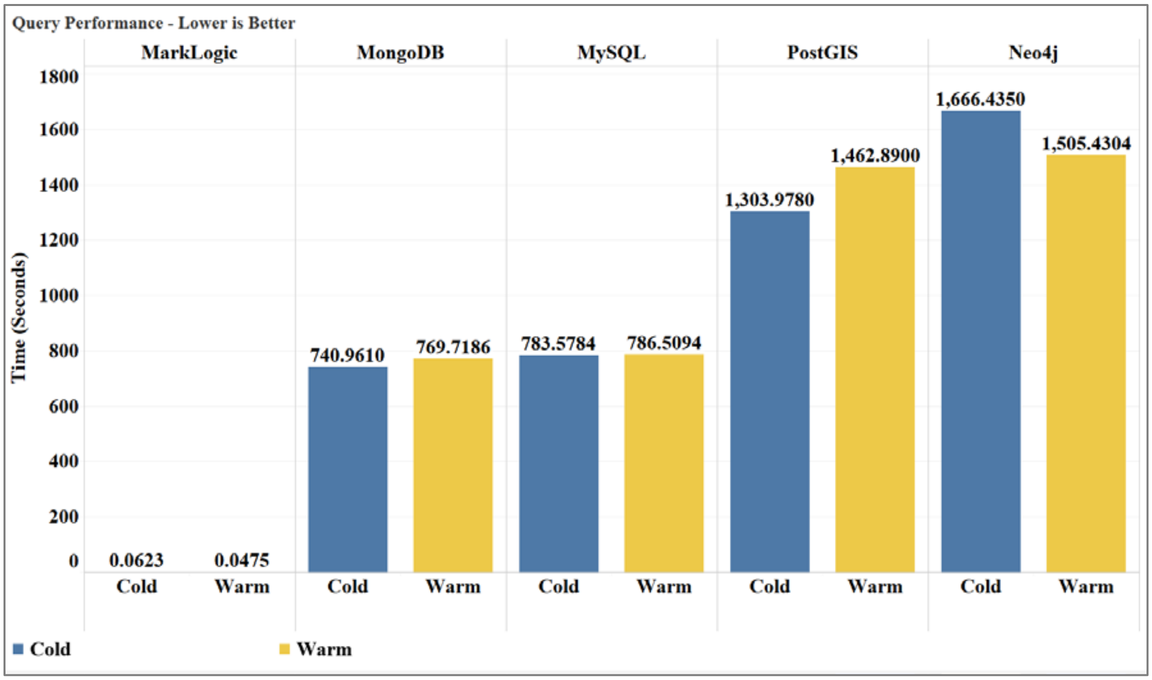

Figure 10: Query Time (cold) and Query Time (warm) for Query 5. Numbers shown are the time needed to process the query in seconds.

Accuracy

As the primary purpose of a database is accurate data storage and retrieval, it was expected that each database would produce the exact same results for the same high-level queries issued. There were not expected to be any variances in the number of results returned. In addition, this metric was used to ensure that the functions used to query each database were in fact the proper ones to use as each database used different query languages.

Somewhat unexpected, not all databases returned the same number of results for every query. Queries 1 and 4 were the only queries that returned the same number of results for all five databases tested. The remainder of the results returned for each of the databases per query had only slight variances from database to database however there were a few noteworthy deviations. For queries 2 and 3, MarkLogic, MongoDB, and PostgreSQL all agreed on the number of results returned per executed query, while MySQL and Neo4j agreed on a different number of result matches. None of the databases agreed on query 5, and they each returned a slightly different number of results with the minimum returned result of 17,032 from MongoDB and a maximum of 17,059 from MySQL, a difference of 24 data points. Table 7 illustrates these commonalities and differences in further detail including the query number, database, and the number of results returned. These deviations will be further discussed in the *Conclusions* section of this thesis.

Table 7: Count of results returned per query for each database.

	Database				
Query #	**MarkLogic**	**MongoDB**	**MySQL**	**Neo4j**	**PostgreSQL**
1	111061	111061	111061	111061	111061
2a	3254	3254	3236	3236	3254
2b	3763	3763	3758	3758	3763
2c	19020	19020	19073	19074	19020
2d	15130	15130	16217	16217	15130
2e	1342	1342	1290	1290	1342
2f	4701	4701	4793	4793	4701
2g	1669	1669	1642	1642	1669
2h	3493	3493	3631	3631	3493
2i	9813	9813	9581	9581	9813
2j	37323	37323	37636	37636	37323
3a	1730	1730	1736	1736	1730
3b	1209	1209	1203	1203	1209
3c	3050	3050	3078	3078	3050
3d	4748	4748	4886	4886	4748
3e	1085	1085	1045	1045	1085

3f	2850	2850	2887	2887	2850
3g	1331	1331	1330	1330	1331
3h	2230	2230	2312	2312	2230
3i	2582	2582	2549	2549	2582
3j	7385	7385	7259	7259	7385
4	74	74	74	74	74
5	17039	17032	17059	17040	17038

Usability and Complexity

Because each of the tested databases were initially built with a specific intention, it was predicted that there would likely be differences in usability and complexity between them. In some cases, a geospatial capability was not built directly into the platform, but rather added as an extension after the product was released. In other cases, the database was built for more general-purpose data storage with geospatial as a small subset of the overall platform.

As expected, there were significant differences in the usability and complexity of each of the database systems tested. All the databases required some amount of initial preprocessing to produce the proper format for optimal indexing within each database system. This effort was mostly equivalent across all the databases. Essentially, the initial ingested data needed to be supplemented to convert its scalar-based data into a geospatial format. Of note, the data preprocessing step for Neo4j was significant in that although the syntax was relatively trivial, the processing itself took over 10 hours to complete.

Out of the databases surveyed, the databases that required the least overall query preprocessing and data manipulation were MarkLogic and PostgreSQL (with PostGIS). As mentioned in the *Methodology* section, the complex geometry for Query 4 required a significant amount of preprocessing for all tested databases except MarkLogic and PostgreSQL (with PostGIS), due to both databases having native support for KML.

Support-wise, Neo4j tended to have the fewest information resources available online. MarkLogic tended to not have much community-provided information but had very comprehensive documentation that made query construction relatively straight-forward. MongoDB tended to have very broad community support and relatively useful product documentation. MySQL had broad community support, but had some vagueness in its documentation, particularly surrounding the units used for geospatial buffers. Both PostgreSQL and PostGIS had an extensive online community with comprehensive documentation which made query construction considerably easier.

Subjectively speaking, the order of usability from best to worst was MarkLogic, MongoDB, PostgreSQL (with PostGIS), MySQL, and Neo4j.

CONCLUSION AND FUTURE RESEARCH

Given an identical input dataset, there were significant differences in the data ingestion time and the resulting storage footprint of the databases. The ingestion time tended to strongly correlate with the resulting size of the database. This relationship is illustrated in Figure 11 which shows the subsequent database size per system as well as the data ingest time. MarkLogic took the longest time to load the data (108 seconds) and had the largest resulting database size (876MB). MongoDB, a NoSQL document-based database, like MarkLogic, had a storage footprint of 425MB, a full 451MB less than MarkLogic, and took 11.5 seconds to load the data. In comparison, MySQL had the second shortest loading time at 4.5 seconds, and the smallest resulting database size (177MB) while PostgreSQL, also an RDBMS, ingested the dataset the fastest (3 seconds) with a resulting database size almost double that of MySQL (342MB). Neo4j, the only graph database of the group, had a loading time of 24.4 seconds, and a resulting database size of 632.3MB.

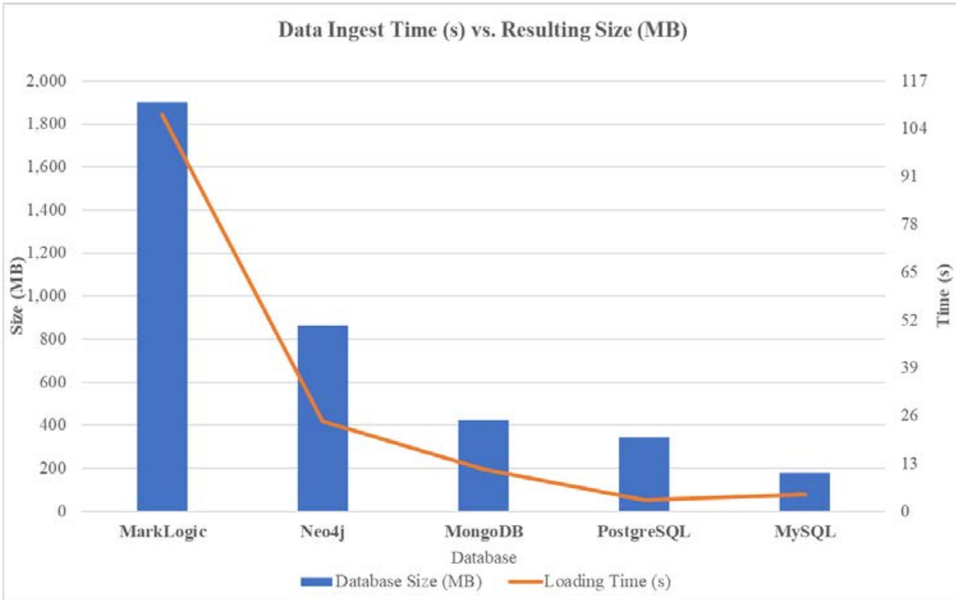

Figure 11. Data ingest time and storage footprint.

The reason for the large variations in storage size and ingestion time is due in part to the difference in data structures used by each database to store the dataset. Figures 12, 13, and 14 show how each database stored the same dataset differently. MarkLogic and MongoDB store their data as JSON documents. MySQL and PostgreSQL store their data in tabular format (relational), and Neo4j stores its data as Nodes, which contain keys and values (much like a document). Additionally, the databases have different default indexing strategies. For example, upon ingestion into the MarkLogic database every field from each record is added to its universal index, which is MarkLogic's mechanism for querying data by value. This universal index provides capabilities more aligned with a search engine, such as term-frequency/inverse-document-frequency relevance scoring for results. As a result, MarkLogic had the longest data ingest time and largest storage footprint. The other analyzed databases don't build a general-purpose index by default, and instead rely on a complete database scan when running queries on non-indexed fields. The lack of these indexes by default results in smaller on-disk sizes, at the expense of general-purpose query performance. To more accurately compare the databases in this regard, it would be necessary to add a text index on every field in each record and compare resulting data size.

```
{
    "dep_id": "10183980",
    "url": "https://mrdata.usgs.gov/mrds/show-mrds.php?dep_id=10183980",
    "mrds_id": "",
    "mas_id": "0320030367",
    "site_name": "Yellow Jacket Group",
    "latitude": "35.7003",
    "longitude": "-115.30164",
    "region": "NA",
    "country": "United States",
    "state": "Nevada",
    "county": "Clark",
    "com_type": "M",
    "commod1": "Uranium",
    "commod2": "",
    "commod3": "",
    "oper_type": "Surface",
    "dep_type": "",
    "prod_size": "",
    "dev_stat": "Prospect",
    "ore": "",
    "gangue": "",
    "other_matl": "",
    "orebody_fm": "",
    "work_type": "",
    "model": "",
    "alteration": "",
    "conc_proc": "",
    "names": "Yellow Jacket Grp",
    "ore_ctrl": "",
    "reporter": "Ridenour, James",
    "hrock_unit": "",
    "hrock_type": "",
    "arock_unit": "",
    "arock_type": "",
    "structure": "",
    "tectonic": "",
    "ref": "NEV BUR MINES BULL.81,1973,P38",
    "yfp_ba": "",
    "yr_fst_prd": "",
    "ylp_ba": "",
    "yr_lst_prd": "",
    "dy_ba": "",
    "disc_yr": "",
    "prod_yrs": "",
    "discr": "",
    "point": {
        "latitude": 35.7003,
        "longitude": -115.30164
        }
    }
```

Figure 12: The JSON based data structure for MarkLogic and MongoDB.

mrds_id	A010000
country	United States
com_type	M
gtype	1
dep_id	10000001
ore	Chalcopyrite, Covellite, Pyrite
bbox	[-132.14344, 55.05612, -132.14344, 55.05612]
latitude	55.05612
dev_stat	Occurrence
reporter	Hirschmann, M. M. (Elliott, R. L.)
structure	Schist Strikes N65w, Dips 70sw
hrock_type	Schist
gangue	Quartz, Sericite
url	https://mrdata.usgs.gov/mrds/show-mrds.php?dep_id=10000001
site_name	Lookout Prospect
ref	USGS PROFESSIONAL PAPER 1, P. 75-77.USGS BULL 347, P. 131.USGS BULL 1246, P. 174USGS MF 433USGS OF 78-869, P. 117
names	Conundrum, Mammoth, Wakefield Minerals Co.
oper_type	Unknown
commod1	Copper
prod_size	N
commod2	Gold, Silver
state	Alaska
region	NA
longitude	-132.14344

Figure 13: View of a Neo4j node (a node contains keys and values).

dep_id	url	mrds_id	mas_id	site_name	latitude	longitude	region	country	state	county	com_type	commod1	commod2
10000001	https://mrdata.usgs.gov/mrds/show-mrds.php?dep_id=10000001	A010000	(null)	Lookout Prospect	55.05612	-132.14344	NA	United States	Alaska	(null)	M	Copper	Gold, Silv
10000002	https://mrdata.usgs.gov/mrds/show-mrds.php?dep_id=10000002	A010001	(null)	Lucky Find Prospect	55.52751	-132.68514	NA	United States	Alaska	(null)	M	Copper	Gold
10000003	https://mrdata.usgs.gov/mrds/show-mrds.php?dep_id=10000003	A010002	(null)	Mccullough Prospect	55.97751	-132.99906	NA	United States	Alaska	(null)	M	Copper	(null)
10000004	https://mrdata.usgs.gov/mrds/show-mrds.php?dep_id=10000004	A010003	(null)	Lucky Jim Claim	55.52195	-132.68653	NA	United States	Alaska	(null)	M	Gold	(null)
10000005	https://mrdata.usgs.gov/mrds/show-mrds.php?dep_id=10000005	A010004	(null)	Matilda Occurrence	55.14556	-132.05233	NA	United States	Alaska	(null)	M	Gold	(null)
10000006	https://mrdata.usgs.gov/mrds/show-mrds.php?dep_id=10000006	A010005	(null)	Marion Prospect	55.14695	-132.48512	NA	United States	Alaska	(null)	M	Copper	(null)
10000007	https://mrdata.usgs.gov/mrds/show-mrds.php?dep_id=10000007	A010006	(null)	Marble Heart Prospect	55.3289	-132.76013	NA	United States	Alaska	(null)	M	Lead	(null)
10000008	https://mrdata.usgs.gov/mrds/show-mrds.php?dep_id=10000008	A010007	(null)	Morning Star Prospect	55.56362	-132.45042	NA	United States	Alaska	(null)	M	Gold	Copper
10000009	https://mrdata.usgs.gov/mrds/show-mrds.php?dep_id=10000009	A010008	(null)	Monday Prospect	55.50529	-132.63237	NA	United States	Alaska	(null)	M	Silver	Gold

Figure 14: MySQL (and PostrgeSQL data view (tabular).

Variations in query time and database performance were also prevalent among the five systems analyzed with query 5 resulting in the longest execution time for all systems. MarkLogic had the fastest query time for all 5 queries with an overall average resolution time of 0.014 seconds. MongoDB and MySQL had similar overall average query times of 151 and 157 seconds respectively with query 1 being the fastest and query 5 taking the longest to resolve for both databases. This similarity occurred even though MongoDB and MySQL store and retrieve data in very different ways. In comparison, the variance that resulted between MarkLogic and MongoDB was unexpected because, on paper, these two databases seem to be most similar in that they are both NoSQL document-based databases.

Neo4j had the longest runtime out of the five systems for every query performed including Query 1, which was the simplest of all the defined queries. For query 5 Neo4j took an additional 202 seconds longer than PostgreSQL to complete and finished Query 4 in 7.76 seconds while this same query took MySQL a mere 0.034 seconds, a difference of 7.72 seconds. The overall lackluster performance of Neo4j compared to MySQL, was unexpected because it has been reported that this system is roughly 1000 times faster than relational systems (Nixon, 2015).

MySQL and PostgreSQL both outperformed MongoDB in executing Query 3 where it had a faster runtime by 0.22 and .07 seconds, respectively. In contrast, PostgreSQL had the second longest runtimes for queries 2, 4, and 5. It took PostgreSQL 1,383 seconds or 23 minutes to complete query 5 while MySQL executed in 785 seconds, coming in third fastest. It is important to note that the reason MySQL didn’t process the query faster is likely an effect of not using an index to calculate this result, as this result was orders of magnitude slower than the previous complex geometry, Query 4, conducted using MySQL. An “explain plan” on Query 5 against MySQL showed that it would use an index, but the astronomical result indicates otherwise. Multiple attempts were made to force MySQL to use the index, but the results were similar.

With respect to accuracy, each database agreed on the returned results for both queries 1 and 4. Query 1 was a simple attribute query and therefore left little room for ambiguity. Query 4 was a complex geospatial buffer query confined to a small region and thus not heavily influenced by the projections employed by each database tested. Queries 2, 3, and 5 showed variations in the number of results returned among all the databases tested, with some observable groupings present in the outputs.

For queries 2 and 3, MarkLogic, MongoDB, and PostgreSQL output the same number of results, which differed from the number of results output by Neo4j and MySQL, which both agreed with each other. Figures 15, 16, 17, and 18 below illustrate the differences observed in queries 2a, 2f, 3a, where the red points represent outputs unique to MarkLogic, MongoDB, and PostgreSQL while the light green points represent those outputs unique to Neo4j and MySQL. What is noteworthy is that these discrepancies occurred on or near the borders of the predefined geographic regions only with no extreme outliers.

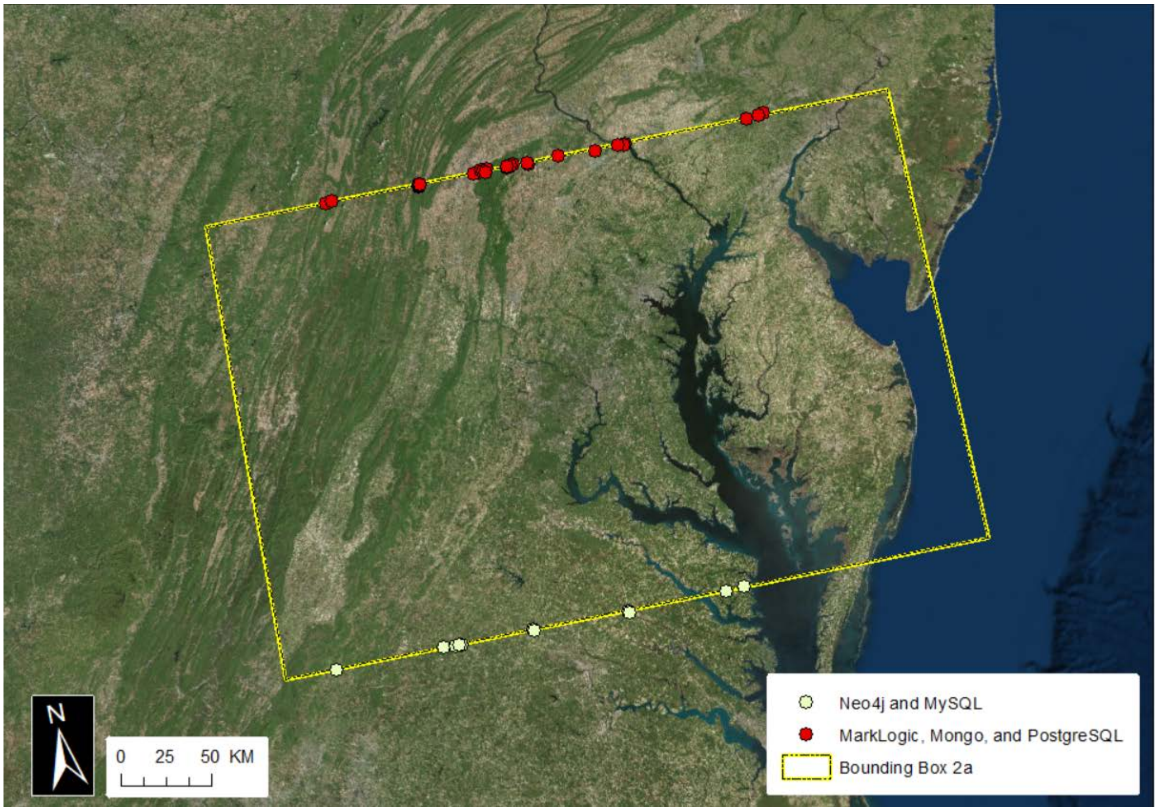

Figure 15: Differences in the results returned from query 2a for MarkLogic, Mongo, and PostgreSQL and Neo4j and MySQL databases.

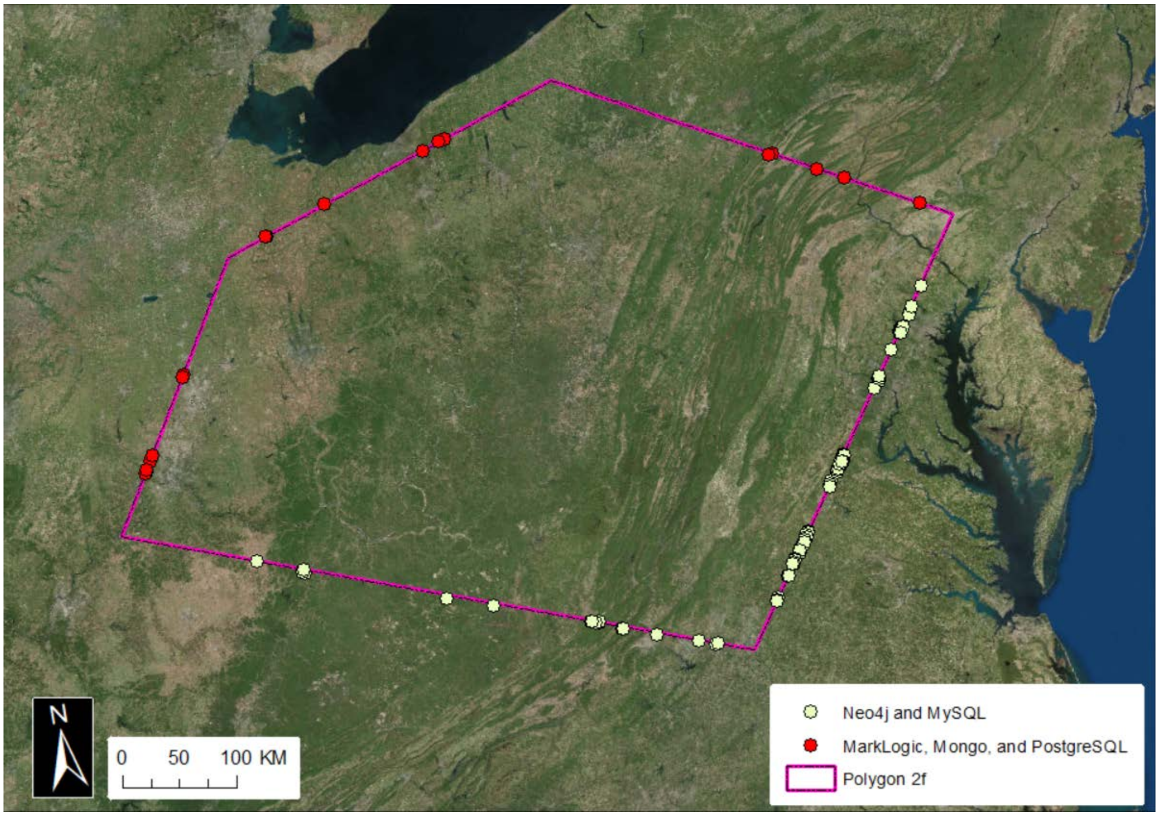

Figure 16: Differences in the results returned from query 2f for MarkLogic, Mongo, and PostgreSQL and Neo4j and MySQL databases.

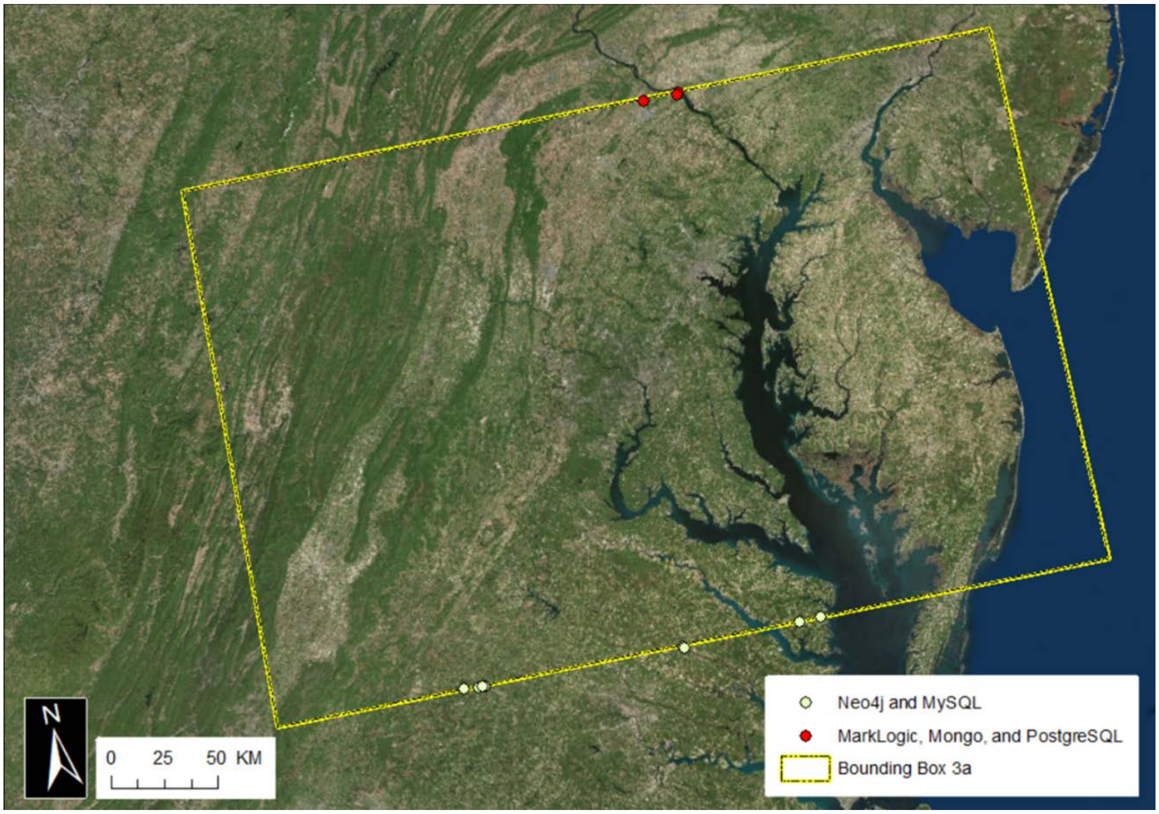

Figure 17: Differences in the results returned from query 3a for MarkLogic, Mongo, and PostgreSQL and Neo4j and MySQL databases.

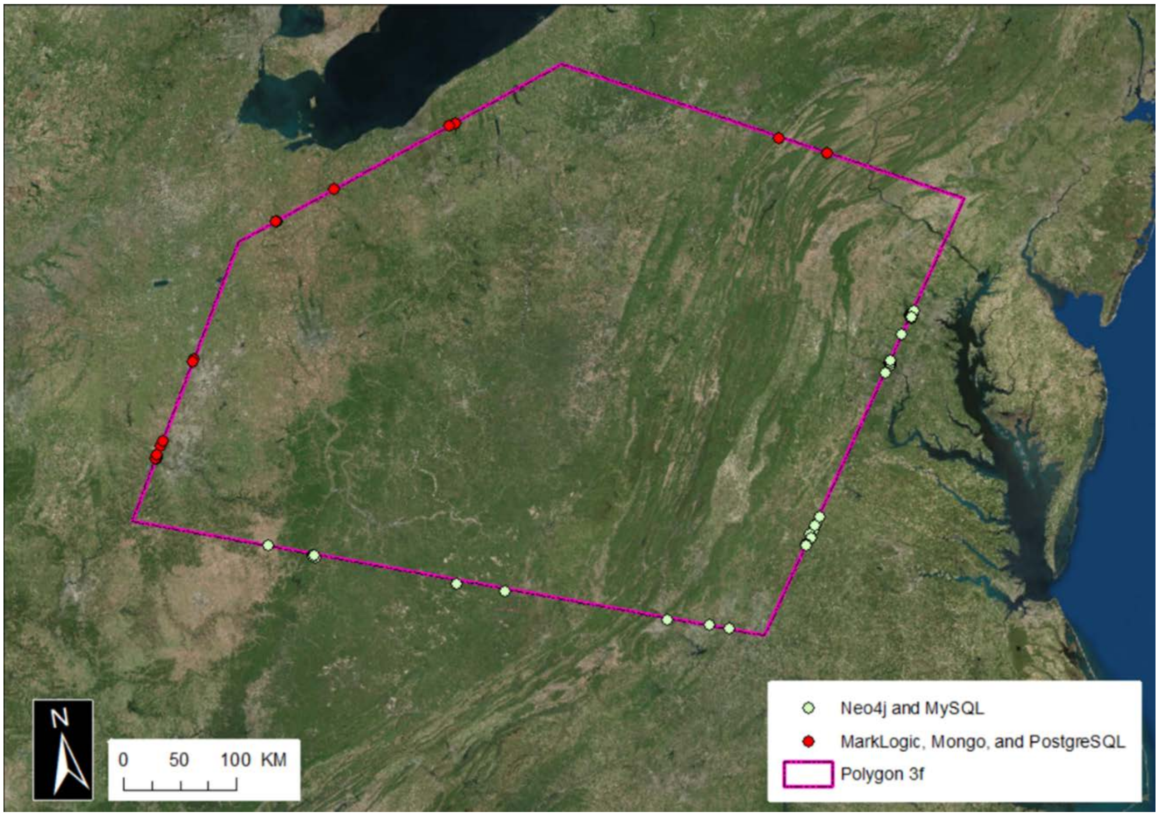

Figure 18: Differences in the results returned from query 3f for MarkLogic, Mongo, and PostgreSQL and Neo4j and MySQL databases.

The variation observed in these queries between the two groups is likely because the regions queried were relatively large, and thus heavily influenced by the curvature of the earth. These two groupings expose a difference of projection by the query engines in these two groups of databases. MarkLogic, MongoDB, and PostgreSQL all execute geodesic calculations when resolving these polygon queries, while MySQL and Neo4j do not appear to have a way to run their calculations geodesically (considering the curvature of the Earth). Interestingly, MarkLogic, MongoDB, and PostgreSQL do provide settings to perform their calculations non-geodesically and return the same result values as MySQL and Neo4j. This problem didn't appear to affect query 4, which was also a polygon query, likely because the polygons for the buffer were contained to a much smaller area, and therefore less susceptible to the influence of the curvature of the Earth.

Query 5 further highlights some differences in the geospatial query techniques between these databases, as every database tested returned a slightly different number of results. Figure 19 below illustrates the total output for all 5 databases combined for query 5. Neo4j, MySQL, and Mongo output points that were unique among the full set while MarkLogic and PostgreSQL with PostGIS had identical outputs. Figure 20 shows the 3 unique values for Neo4j. Figure 21 illustrates the 635 unique records ouput by MySQL. Figure 22 shows the lone unique record output by Mongo. These variances are due to assumptions that each database makes regarding distance when calculated with respect to their query projection and the location of the queried region on the earth. The reason MySQL had so many unique values was because it does not natively support a geospatial buffer query using miles as the unit of measure instead it uses decimal degrees. Therefore, the conversion from decimal degrees to miles was an approximation based on a singular point on the globe.

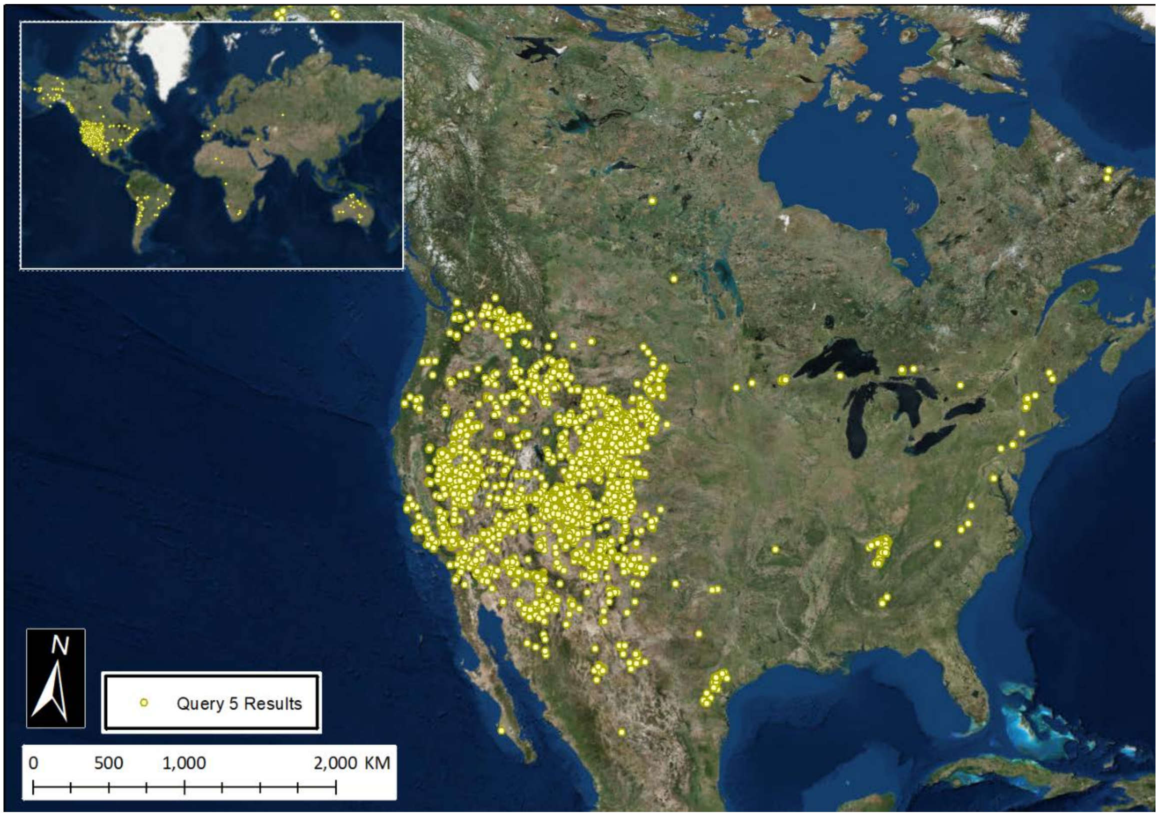

Figure 19: Query 5 outputs for all 5 databases combined.

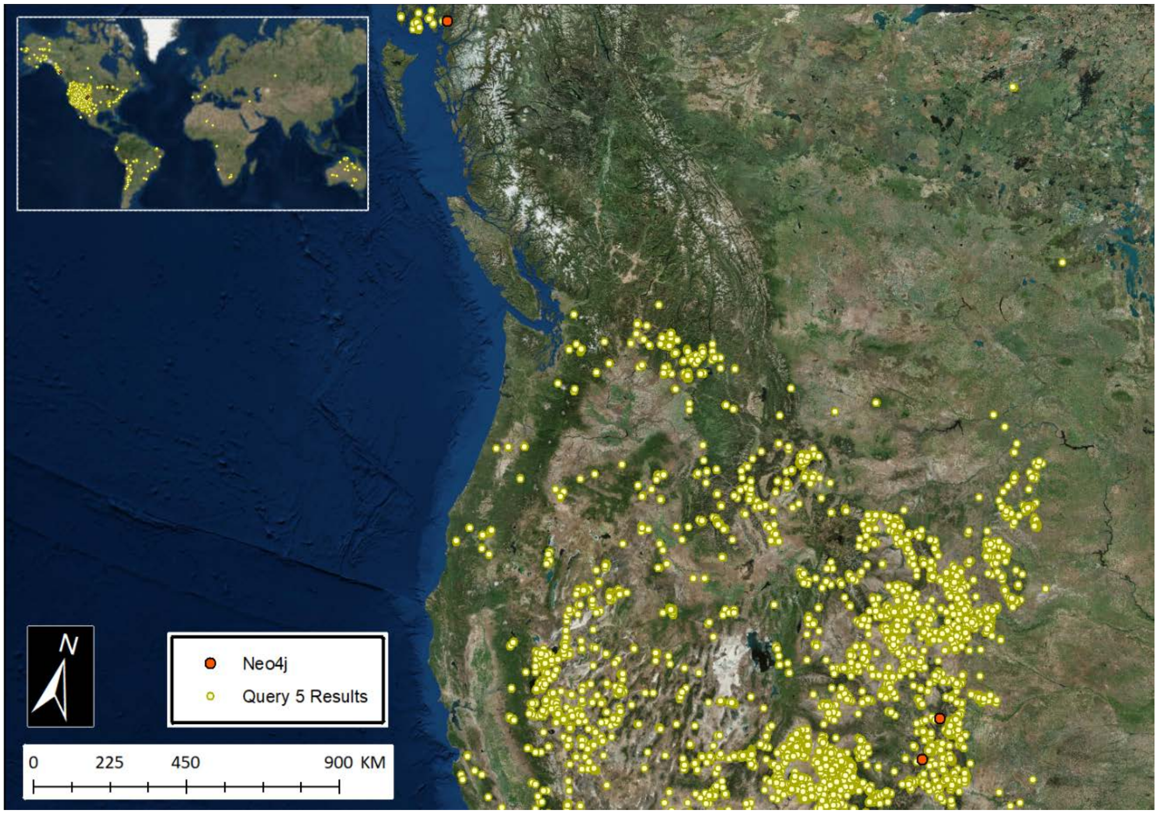

Figure 20: Query 5 results where points unique to only Neo4j are shown in orange while all else are in yellow.

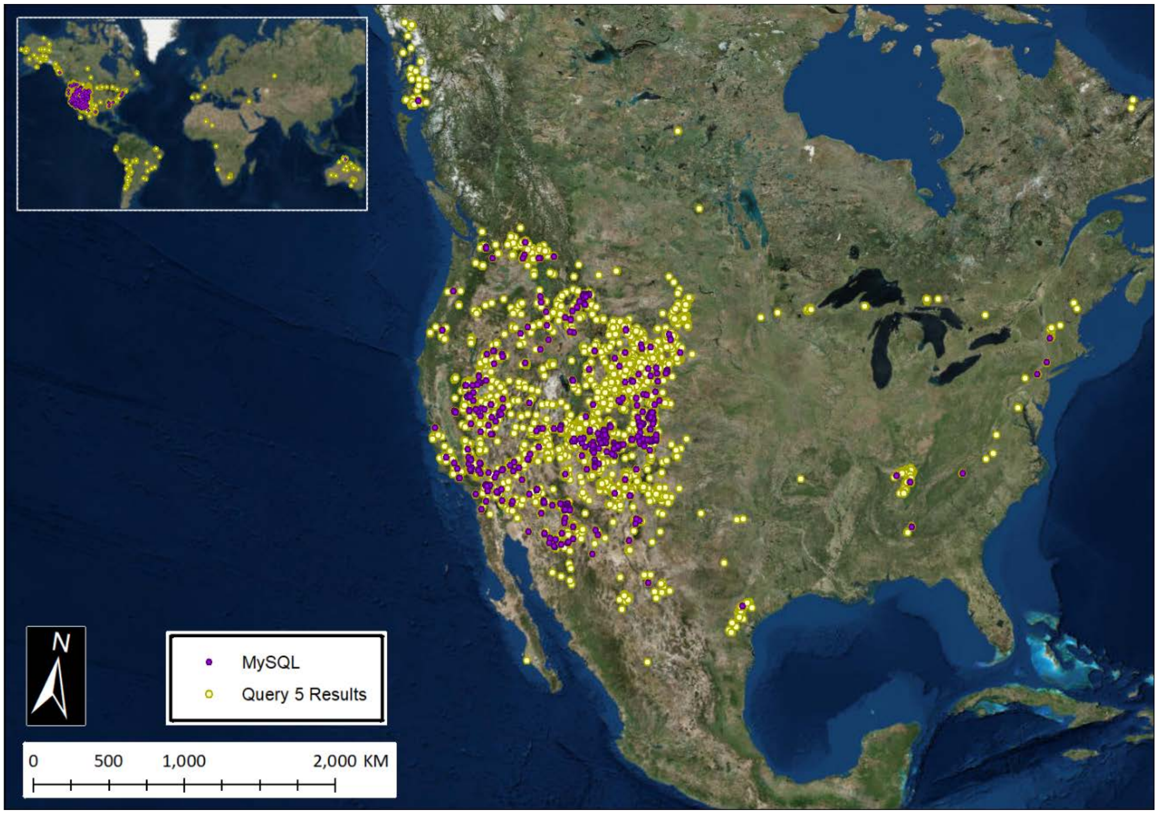

Figure 21: Query 5 results where points unique to only MySQL are shown in purple while all else are in yellow.

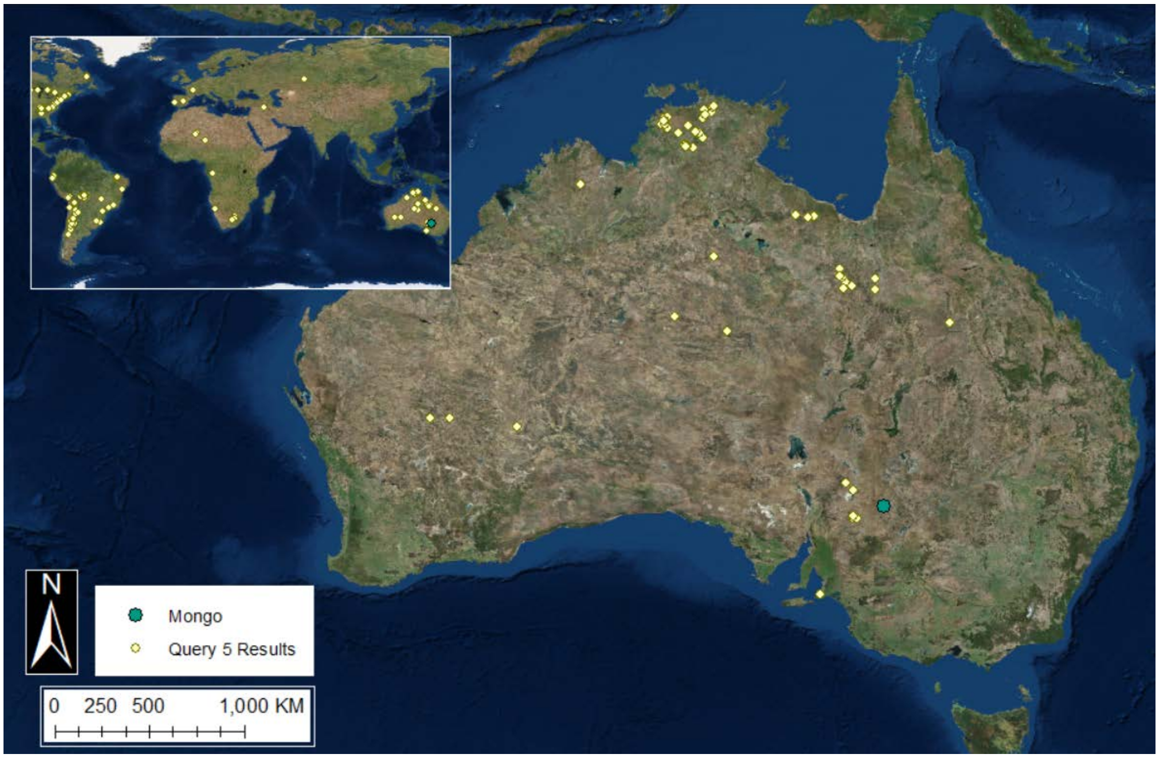

Figure 22: Query 5 results where points unique to only Mongo are shown in cyan while all else are in yellow.

As also predicted, there were noticeable differences in system usability and complexity between each of the database systems analyzed. Based on the ingestion process, data preprocessing, and queries executed in this thesis, the database that seems best suited for geospatial queries and analysis is MarkLogic. MarkLogic required the least amount of query preprocessing. This is because MarkLogic has built-in support for building geometries directly from KML files and using them in queries, which eliminated the need for any preprocessing for Query 4, and saved a significant amount of time and effort. With JavaScript being its primary query language, it has a vast vocabulary of structures for performing a large array of complex tasks. Additionally, MarkLogic provides the built-in QConsole web interface for executing ad-hoc queries, which greatly enhances its usability by providing syntax highlighting, database browsing, and result viewing.

MongoDB was similarly noteworthy in this regards but fell short in its ability to natively handle KML, which is a very common format used in geospatial analysis, and substantially increased the complexity of Query 4. Additionally, its performance in executing Query 5 was several orders of magnitude slower than MarkLogic. There are GUI's available, like Robo 3T, that allow for the execution of queries against MongoDB that decrease the overall complexity of formulating queries and processing data. In Addition, its simple and powerful query syntax also makes it very well suited for running geospatial queries and analysis.

MySQL's rigid language syntax was frustrating for constructing geospatial queries. Due to its lack of native KML support, and SQL's inherent shortcomings in expressiveness, building the geometries for the geospatial queries required a great deal of complexity. Its performance was mostly good, with the notable exception of Query 5. More analysis should be done to determine why the database didn't appear to use the provided index for this query. On the positive side, MySQL has a vast userbase and broad community support, and the available tools for interfacing with it, namely MySQL Workbench, enhance its overall usability.

Neo4j uses a third-party library for executing geospatial queries, and its geospatial capabilities feel likewise an afterthought. Constructing geospatial queries in the Cypher language seemed unintuitive and needlessly complicated. Like MarkLogic, Neo4j provides a built-in web interface for running ad-hoc queries, loading data, and visualizing results. This did enhance its usability considerably, but ultimately didn’t compensate for its other shortcomings in performance and usability.

PostgreSQL is purpose-built for geospatial queries and therefore has native KML support. It is considered in the community as the predominant database for geospatial data storage and retrieval. As a result, there is a plethora of community online documentation and support as well as many available query tools, such as pgAdmin4. SQL is the language used to query the database which does impose some limitations however it also lowers the barrier of entry due to the pervasiveness of SQL. Based on the results of this thesis, what is surprising is that PostgreSQL was not the overall fastest or best geospatial database solution for this dataset.

Table 8 below provides an overall ranking of each of the database systems analyzed in this thesis. Each database was scored for one of seven metrics enumerated and from that the overall system was ranked. For the accuracy component, a score of 1 was given to the databases that were able to correctly query using the geodetic geometries and a score of 2 was given to those which were not. This tabulation mostly agrees with the subjective analysis above, but doesn’t consider the scale of the differences between the databases. For instance, the two-minute data load time for MarkLogic pales in comparison to the 26-minute query time for PostgreSQL when executing Query 5, or the ten-hour processing time of creating the geospatial layer for Neo4j.

Table 8: Overall ranking analysis of each system based on predefined metrics

Database					
Metric	**MarkLogic**	**MongoDB**	**MySQL**	**Neo4j**	**PostgreSQL**
Ingest Time	5	3	2	4	1
Storage & Memory Footprint	5	3	1	4	2
Query Performance Rank Avg.	1	3	2	5	4
Accuracy	1	1	2	2	1
Complexity	1	2	4	5	3
Usability	1	2	4	5	3

Future research should focus on more in-depth analysis of the index types used by each database system, and the strengths and weaknesses of each. More exploration of third-party tools may also result in enhanced usability and increases in query and data ingestion performance for each database examined here. Additionally, each of these database technologies is still being developed and enhanced, so revisiting the same queries in the future is warranted and may yield different results.

APPENDIX

The following tables contain supplemental data mentioned within this thesis.

Table 9: Example of the contents within the KML file

Potomac Buffer KML File

```
<?xml version="1.0" encoding="UTF-8"?>
  <kml xmlns="http://www.opengis.net/kml/2.2">
    <Document>
      <LookAt>
        <longitude>-77.0861321636</longitude>
        <latitude>38.9022958677</latitude>
        <range>3000</range>
        <tilt>0</tilt>
        <heading>0</heading>
      </LookAt>
      <Style id="examplePolyStyle">
        <PolyStyle>
          <color>ff0000cc</color>
          <colorMode>random</colorMode>
          <fill>1</fill>
          <outline>0</outline>
        </PolyStyle>
      </Style>
      <Placemark>
        <name>Potomac</name>
        <description> Buffer: 5 miles</description>
        <styleUrl>#examplePolyStyle</styleUrl>
        <MultiGeometry>
          <Polygon>
            <outerBoundaryIs>
              <LinearRing>
                <coordinates>-77.1232869978,38.7908060665,0 -77.1258730736,38.8171074753,0 -
76.9409902771,38.8313033005,0 -76.9384042013,38.8050018916,0 -
77.1232869978,38.7908060665,0</coordinates>
              </LinearRing>
            </outerBoundaryIs>
          </Polygon>

          [MORE POLYGON COORDINATE DATA HERE]

        </MultiGeometry>
      </Placemark>
    </Document>
  </kml>
```

MySQL Queries

Table 10: MySQL supplemental code and data structure

#	Code
1	SELECT count(*) FROM mrds.mrds WHERE com_type = "N"
2	SELECT count(*) from mrds.mrds WHERE st_contains(geomfromtext('POLYGON(([Coordinates for specific subquery]))', 4326), coords);

3	SELECT count(*) from mrds.mrds
	WHERE st_contains(
	  geomfromtext('POLYGON((*[Coordinates for specific subquery]*))'),
	  mrds.coords
	)
	AND mrds.com_type = "N"

4	SELECT count(*) FROM mrds.mrds WHERE ST_CONTAINS(GeomFromText('MULTIPOLYGON(((*[Coordinates]*)), coords)

5	set session group_concat_max_len = 100000000;
	set @str := '';
	SELECT @str := group_concat(astext(buffer(coords, .018))) from mrds.mrds
	WHERE mrds.commod1 = 'uranium';
	set @str := cast(@str as CHAR);
	set @str := replace(@str, 'POLYGON', '');
	set @str := concat('MULTIPOLYGON(', @str, ')');
	
	SELECT count(*) from mrds.mrds force index (coords_index)
	WHERE st_contains(st_geomfromtext(@str, 4326), coords);

Neo4j Queries

Table 11: Neo4j supplemental code and data structure

#	Code
1	MATCH (r:Resource) WHERE r.com_type = "N" RETURN count(*)
2	WITH "POLYGON((*[Coordinates for specific subquery]*))" AS polygon CALL spatial.intersects('layer_resources', polygon) YIELD node RETURN count(*)
3	WITH "POLYGON((*[Coordinates for specific subquery]*))" AS polygon CALL spatial.intersects('layer_resources', polygon) YIELD node WHERE node.com_type = "N" RETURN count(*)
4	WITH [] as depIds, "POLYGON((*[Coordinates]*))" as polygon CALL spatial.intersects('layer_resources', polygon) YIELD node WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds, "POLYGON((*[Coordinates]*))" as polygon CALL spatial.intersects('layer_resources', polygon) YIELD node WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds, "POLYGON((*[Coordinates]*))" as polygon CALL spatial.intersects('layer_resources', polygon) YIELD node WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds, "POLYGON((*[Coordinates]*))" as polygon CALL spatial.intersects('layer_resources', polygon) YIELD node WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds, "POLYGON((*[Coordinates]*))" as polygon CALL spatial.intersects('layer_resources', polygon) YIELD node WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds, "POLYGON((*[Coordinates]*))" as polygon CALL spatial.intersects('layer_resources', polygon) YIELD node WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds, "POLYGON((*[Coordinates]*))" as polygon CALL spatial.intersects('layer_resources', polygon) YIELD node WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds, "POLYGON((*[Coordinates]*))" as polygon CALL spatial.intersects('layer_resources', polygon) YIELD node WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds, "POLYGON((*[Coordinates]*))" as polygon CALL spatial.intersects('layer_resources', polygon) YIELD node WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds, "POLYGON((*[Coordinates]*))" as polygon CALL spatial.intersects('layer_resources', polygon) YIELD node WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds, "POLYGON((*[Coordinates]*))" as polygon CALL spatial.intersects('layer_resources', polygon) YIELD node WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds, "POLYGON((*[Coordinates]*))" as polygon CALL spatial.intersects('layer_resources', polygon) YIELD node WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds, "POLYGON((*[Coordinates]*))" as polygon CALL spatial.intersects('layer_resources', polygon) YIELD node WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds, "POLYGON((*[Coordinates]*))" as polygon

```
CALL spatial.intersects('layer_resources', polygon) YIELD node
WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds,
"POLYGON(([Coordinates]))" as polygon
CALL spatial.intersects('layer_resources', polygon) YIELD node
WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds,
"POLYGON(([Coordinates]))" as polygon
CALL spatial.intersects('layer_resources', polygon) YIELD node
WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds,
"POLYGON(([Coordinates]))" as polygon
CALL spatial.intersects('layer_resources', polygon) YIELD node
WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds,
"POLYGON(([Coordinates]))" as polygon
CALL spatial.intersects('layer_resources', polygon) YIELD node
WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds,
"POLYGON(([Coordinates]))" as polygon
CALL spatial.intersects('layer_resources', polygon) YIELD node
WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds,
"POLYGON(([Coordinates]))" as polygon
CALL spatial.intersects('layer_resources', polygon) YIELD node
WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds,
"POLYGON(([Coordinates]))" as polygon
CALL spatial.intersects('layer_resources', polygon) YIELD node
WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds,
"POLYGON(([Coordinates]))" as polygon
CALL spatial.intersects('layer_resources', polygon) YIELD node
WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds,
"POLYGON(([Coordinates]))" as polygon
CALL spatial.intersects('layer_resources', polygon) YIELD node
WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds,
"POLYGON(([Coordinates]))" as polygon
CALL spatial.intersects('layer_resources', polygon) YIELD node
WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds,
"POLYGON(([Coordinates]))" as polygon
CALL spatial.intersects('layer_resources', polygon) YIELD node
WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds,
"POLYGON(([Coordinates]))" as polygon
CALL spatial.intersects('layer_resources', polygon) YIELD node
WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds,
"POLYGON(([Coordinates]))" as polygon
CALL spatial.intersects('layer_resources', polygon) YIELD node
WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds,
"POLYGON(([Coordinates]))" as polygon
CALL spatial.intersects('layer_resources', polygon) YIELD node
WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds,
"POLYGON(([Coordinates]))" as polygon
CALL spatial.intersects('layer_resources', polygon) YIELD node
WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds,
"POLYGON(([Coordinates]))" as polygon
CALL spatial.intersects('layer_resources', polygon) YIELD node
WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds,
"POLYGON(([Coordinates]))" as polygon
CALL spatial.intersects('layer_resources', polygon) YIELD node
WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds,
"POLYGON(([Coordinates]))" as polygon
CALL spatial.intersects('layer_resources', polygon) YIELD node
WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds,
"POLYGON(([Coordinates]))" as polygon
CALL spatial.intersects('layer_resources', polygon) YIELD node
WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds,
```

```
"POLYGON(([Coordinates]))" as polygon
CALL spatial.intersects('layer_resources', polygon) YIELD node
WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds,
"POLYGON(([Coordinates]))" as polygon
CALL spatial.intersects('layer_resources', polygon) YIELD node
WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds,
"POLYGON(([Coordinates]))" as polygon
CALL spatial.intersects('layer_resources', polygon) YIELD node
WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds,
"POLYGON(([Coordinates]))" as polygon
CALL spatial.intersects('layer_resources', polygon) YIELD node
WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds,
"POLYGON(([Coordinates]))" as polygon
CALL spatial.intersects('layer_resources', polygon) YIELD node
WITH filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds as depIds,
"POLYGON(([Coordinates]))" as polygon
CALL spatial.intersects('layer_resources', polygon) YIELD node
RETURN size(filter(x IN collect(node.dep_id) WHERE NOT x IN depIds) + depIds)
```

5

```
MATCH (r:Resource)
WHERE r.commod1 = "Uranium" AND r.latitude <> "" AND r.longitude <> ""
WITH {latitude: r.latitude, longitude: r.longitude} as coordinate
CALL spatial.withinDistance('layer_resources', coordinate, 1.60934) YIELD node
RETURN count(DISTINCT node.dep_id)
```

MarkLogic Queries

Table 12: MarkLogic supplemental code and data structure

#	Code
1	var q = cts.elementValueQuery("com_type", "N"); [cts.estimate(q), xdmp.elapsedTime()];

2	var boxes = [
	  cts.polygon([cts.point(37.7544, -79.3124), cts.point(40.0182, -79.3124), cts.point(40.0182, -74.7763), cts.point(37.7544, -74.7763)]),
	  cts.polygon([cts.point(31.33, -86.99), cts.point(33.78, -86.99), cts.point(33.78, -82.46), cts.point(31.33, -82.46)]),
	  cts.polygon([cts.point(34.88, -119.27), cts.point(41.41, -119.27), cts.point(41.41, -114.54), cts.point(34.88, -114.54)]),
	  cts.polygon([cts.point(45.41, -121.2), cts.point(48.56, -121.2), cts.point(48.56, -103.27), cts.point(45.41, -103.27)]),
	  cts.polygon([cts.point(30.37, -102.13), cts.point(34.17, -102.13), cts.point(34.17, -94.75), cts.point(30.37, -94.75)])
	];
	
	var shapes = [
	  cts.polygon("41.77131,-79.98047 40.14529,-76.06934 37.09024,-78.92578 38.69938,-85.08694 40.77448,-83.63219"),
	  cts.polygon("31.16581,-89.5166 30.9797,-95.29633 34.17735,-97.89283 33.06392,-93.60352 35.45702,-91.58515 32.99024,-88.28613"),
	  cts.polygon("42.45589,-101.77734 45.59973,-106.7638 47.36859,-101.20605 42.45589,-92.90039"),
	  cts.polygon("42.87596,-120.9375 42.74701,-112.67578 47.36533,-117.73573"),
	  cts.polygon("40.04444,-117.94922 40.11169,-105.11719 35.03,-107.92969 34.95836,-119.75142")
	];
	
	var q = cts.jsonPropertyPairGeospatialQuery(
	    "point",
	    "latitude",
	    "longitude",
	    boxes[4] //Update this variable depending on the shape being queried
	);
	
	[cts.estimate(q), xdmp.elapsedTime()];

3	var boxes = [
	  cts.polygon([cts.point(37.7544, -79.3124), cts.point(40.0182, -79.3124), cts.point(40.0182, -74.7763), cts.point(37.7544, -74.7763)]),
	  cts.polygon([cts.point(31.33, -86.99), cts.point(33.78, -86.99), cts.point(33.78, -82.46), cts.point(31.33, -82.46)]),
	  cts.polygon([cts.point(34.88, -119.27), cts.point(41.41, -119.27), cts.point(41.41, -114.54), cts.point(34.88, -114.54)]),
	  cts.polygon([cts.point(45.41, -121.2), cts.point(48.56, -121.2), cts.point(48.56, -103.27), cts.point(45.41, -103.27)]),
	  cts.polygon([cts.point(30.37, -102.13), cts.point(34.17, -102.13), cts.point(34.17, -94.75), cts.point(30.37, -94.75)])
	];
	
	var shapes = [
	  cts.polygon("41.77131,-79.98047 40.14529,-76.06934 37.09024,-78.92578 38.69938,-85.08694 40.77448,-83.63219"),
	  cts.polygon("31.16581,-89.5166 30.9797,-95.29633 34.17735,-97.89283 33.06392,-93.60352 35.45702,-91.58515 32.99024,-88.28613"),

```
  cts.polygon("42.45589,-101.77734 45.59973,-106.7638 47.36859,-101.20605 42.45589,-92.90039"),
  cts.polygon("42.87596,-120.9375 42.74701,-112.67578 47.36533,-117.73573"),
  cts.polygon("40.04444,-117.94922 40.11169,-105.11719 35.03,-107.92969 34.95836,-119.75142")
];

var q1 = cts.jsonPropertyPairGeospatialQuery(
    "point",
    "latitude",
    "longitude",
    boxes[4]  //Update this variable depending on the shape being queried
);

var q2 = cts.jsonPropertyValueQuery("com_type", "N");

var q = cts.andQuery([q1, q2]);

[cts.estimate(q), xdmp.elapsedTime()];
```

4

```
var geokml = require('/MarkLogic/geospatial/kml.xqy');

var kmlText = xdmp.filesystemFile('/tmp/potomac_buffer_5_miles.kml');
var kml = fn.head(fn.head(xdmp.unquote(kmlText)).root.xpath('.//*:Placemark[1]//*:MultiGeometry'));
var geometry = geokml.parseKml(kml);
var query = cts.jsonPropertyPairGeospatialQuery(
    "point",
    "latitude",
    "longitude",
    Geometry
);
[cts.estimate(query), xdmp.elapsedTime()]
```

5

```
// Find all records within 1 mile of another record with its primary commodity being uranium
var q1 = cts.jsonPropertyValueQuery("commod1", "uranium");

var uraniumPoints = cts.elementPairGeospatialValues("point", "latitude", "longitude", null, null, q1);

var circleBuffers = [];
for (point of uraniumPoints) {
  circleBuffers.push(cts.circle(1, point));
}

var q2 =
    cts.jsonPropertyPairGeospatialQuery(
    "point",
    "latitude",
    "longitude",
    circleBuffers
  );

[xdmp.estimate(q2), xdmp.elapsedTime()]
```

MongoDB Queries

Table 13: MongoDB supplemental code and data structure

#	Code

1

```
function propertyQuery() {
    var a = new Date();
    var results = db.mrds.find({com_type: "N"}).hint("com_type_1").count();
    var b = new Date();
    var time = b - a;
    return [results, time];
}
```

2

```
function polygonQuery(idx) {
    var points = [
        [[-79.3124, 37.7544], [-79.3124, 40.0182], [-74.7763, 40.0182], [-74.7763, 37.7544], [-79.3124, 37.7544]], //2a
        [[-86.99, 31.33], [-86.99, 33.78], [-82.46, 33.78], [-82.46, 31.33], [-86.99, 31.33]], //2b
        [[-119.27, 34.88], [-119.27, 41.41], [-114.54, 41.41], [-114.54, 34.88], [-119.27, 34.88]], //2c
        [[-121.2, 45.41], [-121.2, 48.56], [-103.27, 48.56], [-103.27, 45.41], [-121.2, 45.41]], //2d
        [[-102.13, 30.37], [-102.13, 34.17], [-94.75, 34.17], [-94.75, 30.37], [-102.13, 30.37]], //2e
        [[-79.98047, 41.77131], [-76.06934, 40.14529], [-78.92578, 37.09024], [-85.08694, 38.69938], [-83.63219, 40.77448], [-79.98047, 41.77131]], //2f
        [[-89.5166, 31.16581], [-95.29633, 30.9797], [-97.89283, 34.17735], [-93.60352, 33.06392], [-91.58515, 35.45702], [-88.28613, 32.99024], [-89.5166, 31.16581]], //2g
        [[-101.77734, 42.45589], [-106.7638, 45.59973], [-101.20605, 47.36859], [-92.90039, 42.45589], [-101.77734, 42.45589]], //2h
        [[-120.9375, 42.87596], [-112.67578, 42.74701], [-117.73573, 47.36533], [-120.9375, 42.87596]], //2i
        [[-117.94922, 40.04444], [-105.11719, 40.11169], [-107.92969, 35.03], [-119.75142, 34.95836], [-117.94922, 40.04444]] //2j
    ];
    var a = new Date();
    var results = db.mrds.find({
        point: {
            $geoWithin: {
                $geometry: {
                    type: "Polygon",
                    coordinates: [points[idx]]
                }
            }
        }
    }).count();
    var b = new Date();
    var time = b - a;
    return [results, time];
}

polygonQuery(0); // Change the input value here depending on the query
```

3

```
function polygonQuery(idx) {
        var points = [
                [[-79.3124, 37.7544], [-79.3124, 40.0182], [-74.7763, 40.0182], [-74.7763, 37.7544], [-79.3124, 37.7544]], //2a
                [[-86.99, 31.33], [-86.99, 33.78], [-82.46, 33.78], [-82.46, 31.33], [-86.99, 31.33]], //2b
                [[-119.27, 34.88], [-119.27, 41.41], [-114.54, 41.41], [-114.54, 34.88], [-119.27, 34.88]], //2c
                [[-121.2, 45.41], [-121.2, 48.56], [-103.27, 48.56], [-103.27, 45.41], [-121.2, 45.41]], //2d
                [[-102.13, 30.37], [-102.13, 34.17], [-94.75, 34.17], [-94.75, 30.37], [-102.13, 30.37]], //2e
```

```
            [[-79.98047, 41.77131], [-76.06934, 40.14529], [-78.92578, 37.09024], [-85.08694, 38.69938], [-
83.63219, 40.77448], [-79.98047, 41.77131]], //2f
            [[-89.5166, 31.16581], [-95.29633, 30.9797], [-97.89283, 34.17735], [-93.60352, 33.06392], [-
91.58515, 35.45702], [-88.28613, 32.99024], [-89.5166, 31.16581]], //2g
            [[-101.77734, 42.45589], [-106.7638, 45.59973], [-101.20605, 47.36859], [-92.90039, 42.45589], [-
101.77734, 42.45589]], //2h
            [[-120.9375, 42.87596], [-112.67578, 42.74701], [-117.73573, 47.36533], [-120.9375, 42.87596]],
//2i
            [[-117.94922, 40.04444], [-105.11719, 40.11169], [-107.92969, 35.03], [-119.75142, 34.95836], [-
117.94922, 40.04444]] //2j
    ];
    var a = new Date();
    var results = db.mrds.find(
    {
      $and: [
        {
          point: {
            $geoWithin: {
              $geometry: {
                type: "Polygon",
                coordinates: [points[idx]]
              }
            }
          }
        },
        {
          com_type: "N"
        }
      ]
    }).count();
    var b = new Date();
    var time = b - a;
    return [results, time];
}

polygonQuery(0); // Change the input value here depending on the query
```

4

```
function bufferQuery() {
  var geoQuery = {
    "$or":[
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
```

```
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}},
        {"point":{"$geoWithin":{"$geometry":{"type":"Polygon", "coordinates":[[Coordinates]]}}}}
    ]
};
    var a = new Date();
    var results = db.mrds.find(geoQuery).count();
    var b = new Date();
    var time = b - a;
    return [results, time];
}
bufferQuery();
5 function geoPointQuery() {
    var a = new Date();
    var circleQueries = db.mrds.find({commod1: "Uranium"}, {point: 1, _id:
0}).toArray().filter(function(point){return point.point != null}).map(
        function(point) {
            return {
                point: {
                    $geoWithin: {
                        $centerSphere: [point.point.coordinates, 1/3963.2]
                    }
                }
            }
        }
    );
    var geoQuery = {
        $or: circleQueries
    };
    var results = db.mrds.find(geoQuery).count();
    var b = new Date();
    var time = b - a;
    return [results, time];
}

geoPointQuery();
```

PostgreSQL Queries

Table 14: PostgreSQL supplemental code and data structure

#	Code
1	SELECT count(*) from mrds WHERE com_type = 'N'
2	SELECT count(*) from mrds WHERE st_covers(st_geogfromtext('SRID=4326;POLYGON((*[Coordinates from specific subquery]*))'), pointgeo)
3	SELECT count(*) from mrds WHERE st_covers(st_geogfromtext('SRID=4326;POLYGON((*[Coordinates from specific subquery]*))'), pointgeo) AND com_type = 'N'
4	SELECT count(*) FROM mrds WHERE st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo) OR st_covers(st_geogfromtext('POLYGON((*[Coordinates]*))'), pointgeo)

```
OR st_covers(st_geogfromtext('POLYGON(([Coordinates]))'), pointgeo)
OR st_covers(st_geogfromtext('POLYGON(([Coordinates]))'), pointgeo)
OR st_covers(st_geogfromtext('POLYGON(([Coordinates]))'), pointgeo)
OR st_covers(st_geogfromtext('POLYGON(([Coordinates]))'), pointgeo)
OR st_covers(st_geogfromtext('POLYGON(([Coordinates]))'), pointgeo)
OR st_covers(st_geogfromtext('POLYGON(([Coordinates]))'), pointgeo)
OR st_covers(st_geogfromtext('POLYGON(([Coordinates]))'), pointgeo)
OR st_covers(st_geogfromtext('POLYGON(([Coordinates]))'), pointgeo)
OR st_covers(st_geogfromtext('POLYGON(([Coordinates]))'), pointgeo)
OR st_covers(st_geogfromtext('POLYGON(([Coordinates]))'), pointgeo)
OR st_covers(st_geogfromtext('POLYGON(([Coordinates]))'), pointgeo)
OR st_covers(st_geogfromtext('POLYGON(([Coordinates]))'), pointgeo)
OR st_covers(st_geogfromtext('POLYGON(([Coordinates]))'), pointgeo)
OR st_covers(st_geogfromtext('POLYGON(([Coordinates]))'), pointgeo)
OR st_covers(st_geogfromtext('POLYGON(([Coordinates]))'), pointgeo)
```

5

```
SELECT COUNT(distinct "b".dep_id) FROM (SELECT * FROM mrds WHERE commod1 = 'Uranium') "a"
INNER JOIN mrds "b" ON st_dwithin("a".pointgeo, "b".pointgeo, 1609.34);
```

Query Runtime Results Table

Table 15: List of all cold and warm query completion times per database and their calculated average

Database	Query #	Cold Duration (s)	Cold Duration Average (s)	Warm Duration (s)	Warm Duration Average (s)
MarkLogic	**1**	0.002023	0.001954	0.001069	0.0012574
MarkLogic		0.002006		0.001241	
MarkLogic		0.001919		0.001408	
MarkLogic		0.001937		0.001056	
MarkLogic		0.001885		0.001513	
MarkLogic	**2a**	0.00123	0.0012946	0.000905	0.0008108
MarkLogic		0.001339		0.000783	
MarkLogic		0.001303		0.000802	
MarkLogic		0.001335		0.000825	
MarkLogic		0.001266		0.000739	
MarkLogic	**2b**	0.001427	0.0014914	0.000864	0.0006768
MarkLogic		0.001526		0.00063	
MarkLogic		0.001657		0.000713	
MarkLogic		0.001434		0.000599	
MarkLogic		0.001413		0.000578	
MarkLogic	**2c**	0.001809	0.001642	0.000583	0.0006826
MarkLogic		0.001637		0.000563	
MarkLogic		0.001715		0.001097	
MarkLogic		0.001534		0.00057	
MarkLogic		0.001515		0.0006	
MarkLogic	**2d**	0.001504	0.0015252	0.000663	0.0006722
MarkLogic		0.001568		0.000614	
MarkLogic		0.001454		0.000757	
MarkLogic		0.001564		0.000635	
MarkLogic		0.001536		0.000692	
MarkLogic	**2e**	0.001902	0.0015868	0.000817	0.0006534
MarkLogic		0.001441		0.000607	
MarkLogic		0.001476		0.000634	
MarkLogic		0.001589		0.000602	

MarkLogic		0.001526		0.000607	
MarkLogic	**2f**	0.002038	0.0017004	0.000964	0.0008022
MarkLogic		0.001534		0.000629	
MarkLogic		0.001625		0.001043	
MarkLogic		0.001614		0.000637	
MarkLogic		0.001691		0.000738	
MarkLogic	**2g**	0.001527	0.0015996	0.000645	0.0006752
MarkLogic		0.001598		0.000659	
MarkLogic		0.001632		0.000719	
MarkLogic		0.00156		0.000678	
MarkLogic		0.001681		0.000675	
MarkLogic	**2h**	0.001735	0.001688	0.000815	0.0006288
MarkLogic		0.00156		0.000162	
MarkLogic		0.00163		0.000691	
MarkLogic		0.001739		0.000692	
MarkLogic		0.001776		0.000784	
MarkLogic	**2i**	0.00179	0.0016724	0.000654	0.000735
MarkLogic		0.001569		0.000769	
MarkLogic		0.001641		0.000682	
MarkLogic		0.001703		0.000901	
MarkLogic		0.001659		0.000669	
MarkLogic	**2j**	0.001596	0.0016494	0.000601	0.0007792
MarkLogic		0.001582		0.000776	
MarkLogic		0.001689		0.000714	
MarkLogic		0.001557		0.000775	
MarkLogic		0.001823		0.00103	
MarkLogic	**3a**	0.002078	0.0019682	0.001122	0.0011516
MarkLogic		0.002029		0.001091	
MarkLogic		0.001951		0.001175	
MarkLogic		0.001846		0.001296	
MarkLogic		0.001937		0.001074	
MarkLogic	**3b**	0.002195	0.0022102	0.001335	0.0010912
MarkLogic		0.002233		0.001205	
MarkLogic		0.00217		0.001162	
MarkLogic		0.002226		0.000675	
MarkLogic		0.002227		0.001079	
MarkLogic	**3c**	0.001908	0.0022858	0.001281	0.0012224

MarkLogic		0.002296		0.001133	
MarkLogic		0.002579		0.001176	
MarkLogic		0.002247		0.001029	
MarkLogic		0.002399		0.001493	
MarkLogic	**3d**	0.003089	0.002508	0.001605	0.001206
MarkLogic		0.002127		0.001107	
MarkLogic		0.00233		0.001512	
MarkLogic		0.002774		0.0011	
MarkLogic		0.00222		0.000706	
MarkLogic	**3e**	0.001596	0.002065	0.001418	0.0012
MarkLogic		0.002252		0.001138	
MarkLogic		0.002059		0.001127	
MarkLogic		0.002226		0.001129	
MarkLogic		0.002192		0.001188	
MarkLogic	**3f**	0.001942	0.002285	0.001274	0.0012758
MarkLogic		0.002525		0.001203	
MarkLogic		0.002288		0.001446	
MarkLogic		0.002277		0.00123	
MarkLogic		0.002393		0.001226	
MarkLogic	**3g**	0.003547	0.0027116	0.00144	0.0012098
MarkLogic		0.002967		0.001239	
MarkLogic		0.002491		0.001145	
MarkLogic		0.002176		0.000664	
MarkLogic		0.002377		0.001561	
MarkLogic	**3h**	0.001905	0.0022582	0.001527	0.0014268
MarkLogic		0.00229		0.001629	
MarkLogic		0.002458		0.001344	
MarkLogic		0.002298		0.00154	
MarkLogic		0.00234		0.001094	
MarkLogic	**3i**	0.002514	0.0024084	0.001186	0.001238
MarkLogic		0.002412		0.00123	
MarkLogic		0.002297		0.00112	
MarkLogic		0.002385		0.00161	
MarkLogic		0.002434		0.001044	
MarkLogic	**3j**	0.001816	0.0023626	0.001225	0.0012186
MarkLogic		0.002588		0.001106	
MarkLogic		0.002775		0.001219	

MarkLogic		0.002412		0.001479	
MarkLogic		0.002222		0.001064	
MarkLogic	**4**	0.01201	0.0121788	0.010643	0.0104978
MarkLogic		0.012078		0.010474	
MarkLogic		0.012714		0.010288	
MarkLogic		0.012029		0.010882	
MarkLogic		0.012063		0.010202	
MarkLogic	**5**	0.059287	0.06227	0.047869	0.0474812
MarkLogic		0.064712		0.047973	
MarkLogic		0.062569		0.047875	
MarkLogic		0.063208		0.047063	
MarkLogic		0.061574		0.046626	
MongoDB	**1**	0.025	0.026	0.019	0.0184
MongoDB		0.026		0.018	
MongoDB		0.028		0.018	
MongoDB		0.025		0.018	
MongoDB		0.026		0.019	
MongoDB	**2a**	0.046	0.0444	0.012	0.013
MongoDB		0.044		0.013	
MongoDB		0.045		0.013	
MongoDB		0.043		0.015	
MongoDB		0.044		0.012	
MongoDB	**2b**	0.055	0.05	0.017	0.0178
MongoDB		0.049		0.017	
MongoDB		0.049		0.016	
MongoDB		0.049		0.023	
MongoDB		0.048		0.016	
MongoDB	**2c**	0.175	0.1726	0.069	0.0684
MongoDB		0.163		0.068	
MongoDB		0.184		0.068	
MongoDB		0.176		0.068	
MongoDB		0.165		0.069	
MongoDB	**2d**	0.173	0.1702	0.075	0.0752
MongoDB		0.175		0.075	
MongoDB		0.167		0.077	
MongoDB		0.172		0.075	
MongoDB		0.164		0.074	

MongoDB	**2e**	0.047	0.0434	0.009	0.0094
MongoDB		0.048		0.009	
MongoDB		0.037		0.01	
MongoDB		0.049		0.01	
MongoDB		0.036		0.009	
MongoDB	**2f**	0.079	0.0744	0.027	0.0276
MongoDB		0.07		0.028	
MongoDB		0.081		0.028	
MongoDB		0.072		0.027	
MongoDB		0.07		0.028	
MongoDB	**2g**	0.065	0.0626	0.017	0.018
MongoDB		0.055		0.018	
MongoDB		0.063		0.018	
MongoDB		0.064		0.018	
MongoDB		0.066		0.019	
MongoDB	**2h**	0.059	0.0546	0.015	0.0152
MongoDB		0.048		0.015	
MongoDB		0.059		0.016	
MongoDB		0.059		0.014	
MongoDB		0.048		0.016	
MongoDB	**2i**	0.093	0.087	0.035	0.0346
MongoDB		0.084		0.034	
MongoDB		0.082		0.035	
MongoDB		0.093		0.034	
MongoDB		0.083		0.035	
MongoDB	**2j**	0.316	0.315	0.143	0.1454
MongoDB		0.313		0.145	
MongoDB		0.302		0.145	
MongoDB		0.317		0.147	
MongoDB		0.327		0.147	
MongoDB	**3a**	0.061	0.0612	0.014	0.0134
MongoDB		0.061		0.013	
MongoDB		0.061		0.013	
MongoDB		0.062		0.013	
MongoDB		0.061		0.014	
MongoDB	**3b**	0.059	0.0588	0.018	0.0182
MongoDB		0.059		0.018	

MongoDB		0.058		0.018	
MongoDB		0.059		0.019	
MongoDB		0.059		0.018	
MongoDB	**3c**	0.166	0.1734	0.074	0.074
MongoDB		0.179		0.073	
MongoDB		0.177		0.075	
MongoDB		0.176		0.073	
MongoDB		0.169		0.075	
MongoDB	**3d**	0.166	0.1706	0.081	0.0804
MongoDB		0.175		0.079	
MongoDB		0.175		0.08	
MongoDB		0.171		0.08	
MongoDB		0.166		0.082	
MongoDB	**3e**	0.053	0.0528	0.01	0.01
MongoDB		0.052		0.01	
MongoDB		0.053		0.01	
MongoDB		0.053		0.01	
MongoDB		0.053		0.01	
MongoDB	**3f**	0.085	0.0822	0.03	0.0296
MongoDB		0.075		0.03	
MongoDB		0.084		0.029	
MongoDB		0.085		0.029	
MongoDB		0.082		0.03	
MongoDB	**3g**	0.059	0.0654	0.019	0.0192
MongoDB		0.071		0.02	
MongoDB		0.059		0.019	
MongoDB		0.07		0.019	
MongoDB		0.068		0.019	
MongoDB	**3h**	0.064	0.0604	0.016	0.016
MongoDB		0.055		0.016	
MongoDB		0.054		0.016	
MongoDB		0.065		0.016	
MongoDB		0.064		0.016	
MongoDB	**3i**	0.101	0.1002	0.039	0.0388
MongoDB		0.101		0.039	
MongoDB		0.091		0.038	
MongoDB		0.105		0.039	

MongoDB		0.103		0.039	
MongoDB	**3j**	1.684	2.1488	0.221	0.2218
MongoDB		1.842		0.221	
MongoDB		2.232		0.222	
MongoDB		2.442		0.222	
MongoDB		2.544		0.223	
MongoDB	**4**	0.076	0.071	0.058	0.057
MongoDB		0.075		0.056	
MongoDB		0.065		0.057	
MongoDB		0.074		0.057	
MongoDB		0.065		0.057	
MongoDB	**5**	711.039	740.961	724.445	769.7186
MongoDB		781.032		775.201	
MongoDB		728.428		838.035	
MongoDB		717.286		736.225	
MongoDB		767.02		774.687	
PostgreSQL	**1**	0.012244	0.0107466	0.008024	0.0082652
PostgreSQL		0.01145		0.008661	
PostgreSQL		0.009759		0.0081	
PostgreSQL		0.011152		0.00805	
PostgreSQL		0.009128		0.008491	
PostgreSQL	**2a**	0.129094	0.108353	0.097041	0.097687
PostgreSQL		0.101668		0.102763	
PostgreSQL		0.108412		0.103519	
PostgreSQL		0.105354		0.092618	
PostgreSQL		0.097237		0.092494	
PostgreSQL	**2b**	0.099354	0.0995096	0.095681	0.100943
PostgreSQL		0.10388		0.09901	
PostgreSQL		0.097711		0.098417	
PostgreSQL		0.099058		0.116239	
PostgreSQL		0.097545		0.095368	
PostgreSQL	**2c**	0.135068	0.140339	0.13295	0.1359644
PostgreSQL		0.136782		0.131826	
PostgreSQL		0.141565		0.151737	
PostgreSQL		0.137947		0.1323	
PostgreSQL		0.150333		0.131009	
PostgreSQL	**2d**	0.115272	0.1174038	0.12789	0.1214968

PostgreSQL		0.116201		0.119979	
PostgreSQL		0.117706		0.111208	
PostgreSQL		0.121547		0.119398	
PostgreSQL		0.116293		0.129009	
PostgreSQL	**2e**	0.096518	0.098741	0.090741	0.0921078
PostgreSQL		0.099538		0.090726	
PostgreSQL		0.096794		0.096247	
PostgreSQL		0.095655		0.090447	
PostgreSQL		0.1052		0.092378	
PostgreSQL	**2f**	0.104489	0.1066626	0.096285	0.0989164
PostgreSQL		0.114789		0.104451	
PostgreSQL		0.104783		0.09602	
PostgreSQL		0.103699		0.096932	
PostgreSQL		0.105553		0.100894	
PostgreSQL	**2g**	0.100946	0.103942	0.095145	0.1019834
PostgreSQL		0.10306		0.113774	
PostgreSQL		0.101316		0.097008	
PostgreSQL		0.106119		0.094291	
PostgreSQL		0.108269		0.109699	
PostgreSQL	**2h**	0.101955	0.1057526	0.094552	0.1004204
PostgreSQL		0.11471		0.101705	
PostgreSQL		0.101784		0.10957	
PostgreSQL		0.101488		0.094735	
PostgreSQL		0.108826		0.10154	
PostgreSQL	**2i**	0.11261	0.1103262	0.103392	0.1080274
PostgreSQL		0.110404		0.10334	
PostgreSQL		0.108643		0.116534	
PostgreSQL		0.110925		0.104328	
PostgreSQL		0.109049		0.112543	
PostgreSQL	**2j**	0.155091	0.1545	0.146954	0.1501066
PostgreSQL		0.151675		0.147875	
PostgreSQL		0.152379		0.143874	
PostgreSQL		0.160909		0.148863	
PostgreSQL		0.152446		0.162967	
PostgreSQL	**3a**	0.185597	0.1915766	0.076251	0.077634
PostgreSQL		0.180146		0.081913	
PostgreSQL		0.195119		0.073577	

PostgreSQL		0.193725		0.083405	
PostgreSQL		0.203296		0.073024	
PostgreSQL	**3b**	0.293427	0.3285902	0.079521	0.073425
PostgreSQL		0.300274		0.069943	
PostgreSQL		0.285929		0.070851	
PostgreSQL		0.360579		0.070227	
PostgreSQL		0.402742		0.076583	
PostgreSQL	**3c**	0.362699	0.4312846	0.091222	0.0824508
PostgreSQL		0.494764		0.090421	
PostgreSQL		0.502965		0.080669	
PostgreSQL		0.465643		0.07543	
PostgreSQL		0.330352		0.074512	
PostgreSQL	**3d**	0.329671	0.333806	0.077243	0.0878634
PostgreSQL		0.395672		0.099652	
PostgreSQL		0.3356		0.101867	
PostgreSQL		0.304615		0.085078	
PostgreSQL		0.303472		0.075477	
PostgreSQL	**3e**	0.313314	0.351766	0.073072	0.0790392
PostgreSQL		0.276604		0.090791	
PostgreSQL		0.436541		0.071411	
PostgreSQL		0.439728		0.088525	
PostgreSQL		0.292643		0.071397	
PostgreSQL	**3f**	0.339362	0.3836084	0.084776	0.0838524
PostgreSQL		0.451154		0.076153	
PostgreSQL		0.421306		0.076887	
PostgreSQL		0.357272		0.083512	
PostgreSQL		0.348948		0.097934	
PostgreSQL	**3g**	0.134353	0.2686088	0.076854	0.0849066
PostgreSQL		0.39837		0.085205	
PostgreSQL		0.244928		0.107742	
PostgreSQL		0.241992		0.079566	
PostgreSQL		0.323401		0.075166	
PostgreSQL	**3h**	0.281429	0.1823866	0.073224	0.086149
PostgreSQL		0.171118		0.074177	
PostgreSQL		0.119878		0.073967	
PostgreSQL		0.186826		0.104618	
PostgreSQL		0.152682		0.104759	

PostgreSQL	**3i**	0.194407	0.1531976	0.077072	0.0789438
PostgreSQL		0.138303		0.074346	
PostgreSQL		0.120643		0.082007	
PostgreSQL		0.143108		0.074768	
PostgreSQL		0.169527		0.086526	
PostgreSQL	**3j**	0.148842	0.1538512	0.092119	0.0898732
PostgreSQL		0.196334		0.085196	
PostgreSQL		0.126435		0.08284	
PostgreSQL		0.147359		0.100643	
PostgreSQL		0.150286		0.088568	
PostgreSQL	**4**	3.41959	3.3991404	3.345733	3.3335642
PostgreSQL		3.426663		3.329342	
PostgreSQL		3.406785		3.338428	
PostgreSQL		3.357049		3.327378	
PostgreSQL		3.385615		3.32694	
PostgreSQL	**5**	1299.979374	1303.977861	1467.178303	1462.889549
PostgreSQL		1329.012883		1456.45122	
PostgreSQL		1290.551296		1490.61865	
PostgreSQL		1301.038063		1450.57571	
PostgreSQL		1299.307687		1449.623861	
MySQL	**1**	0.0250045	0.0248753	0.02388725	0.02483925
MySQL		0.02477325		0.02353325	
MySQL		0.02492425		0.02482625	
MySQL		0.02484175		0.02532725	
MySQL		0.02483275		0.02662225	
MySQL	**2a**	0.020427	0.02000455	0.01391025	0.01139035
MySQL		0.019128		0.01064275	
MySQL		0.0194845		0.010588	
MySQL		0.02077175		0.011277	
MySQL		0.0202115		0.01053375	
MySQL	**2b**	0.01776175	0.0194645	0.013762	0.01320935
MySQL		0.018215		0.01325	
MySQL		0.01806625		0.012788	
MySQL		0.0248995		0.01236325	
MySQL		0.01838		0.0138835	
MySQL	**2c**	0.0824365	0.0822352	0.06268825	0.05995515
MySQL		0.0807585		0.05919325	

MySQL		0.08247025		0.05967	
MySQL		0.084169		0.0592835	
MySQL		0.08134175		0.05894075	
MySQL	**2d**	0.07167075	0.0731389	0.054087	0.05121995
MySQL		0.0727985		0.0502725	
MySQL		0.07558575		0.050472	
MySQL		0.07301675		0.05109325	
MySQL		0.07262275		0.050175	
MySQL	**2e**	0.0105	0.0094997	0.00579375	0.0051088
MySQL		0.00862475		0.00481775	
MySQL		0.0097595		0.00485775	
MySQL		0.00923825		0.00524075	
MySQL		0.009376		0.004834	
MySQL	**2f**	0.04194075	0.03976385	0.0283595	0.02585385
MySQL		0.04009		0.02487575	
MySQL		0.038883		0.02517375	
MySQL		0.03874675		0.02532025	
MySQL		0.03915875		0.02554	
MySQL	**2g**	0.0329225	0.03225445	0.02151	0.02030085
MySQL		0.03193675		0.02081675	
MySQL		0.031804		0.019708	
MySQL		0.0326435		0.019886	
MySQL		0.0319655		0.0195835	
MySQL	**2h**	0.03596875	0.03479905	0.024112	0.0221698
MySQL		0.03495475		0.02099925	
MySQL		0.03412925		0.0221815	
MySQL		0.03439625		0.02172825	
MySQL		0.03454625		0.021828	
MySQL	**2i**	0.07665325	0.074301	0.05450125	0.0520951
MySQL		0.07394425		0.052812	
MySQL		0.072917		0.04950525	
MySQL		0.07464375		0.05039125	
MySQL		0.07334675		0.05326575	
MySQL	**2j**	0.191123	0.1837739	0.13695225	0.13634545
MySQL		0.18070175		0.135557	
MySQL		0.17951525		0.135768	
MySQL		0.18309575		0.136286	

MySQL		0.18443375		0.137164	
MySQL	**3a**	0.0184645	0.01696375	0.01059425	0.0090177
MySQL		0.01512575		0.008472	
MySQL		0.016968		0.0089375	
MySQL		0.0171625		0.0088145	
MySQL		0.017098		0.00827025	
MySQL	**3b**	0.013969	0.01442865	0.00989525	0.0089291
MySQL		0.013863		0.008384	
MySQL		0.01465825		0.0087365	
MySQL		0.0151645		0.00885125	
MySQL		0.0144885		0.0087785	
MySQL	**3c**	0.059653	0.0577175	0.03618275	0.03569315
MySQL		0.055749		0.0371995	
MySQL		0.057273		0.03466575	
MySQL		0.05619325		0.03521975	
MySQL		0.05971925		0.035198	
MySQL	**3d**	0.05823475	0.05543885	0.03632775	0.03354725
MySQL		0.0552085		0.03249325	
MySQL		0.0552385		0.033222	
MySQL		0.0535715		0.03275925	
MySQL		0.054941		0.032934	
MySQL	**3e**	0.008218	0.00908605	0.00528025	0.0048373
MySQL		0.00983425		0.00510175	
MySQL		0.0091115		0.004666	
MySQL		0.0090485		0.0045655	
MySQL		0.009218		0.004573	
MySQL	**3f**	0.03681075	0.0358545	0.0235245	0.02216085
MySQL		0.03452925		0.02194475	
MySQL		0.03578075		0.02095475	
MySQL		0.0347995		0.02119375	
MySQL		0.03735225		0.0231865	
MySQL	**3g**	0.03185025	0.0308635	0.021196	0.01916135
MySQL		0.0296725		0.0189365	
MySQL		0.031208		0.019097	
MySQL		0.03140875		0.0186505	
MySQL		0.030178		0.01792675	
MySQL	**3h**	0.03194675	0.0310507	0.019318	0.0175506

MySQL		0.03055825		0.01777125	
MySQL		0.0311115		0.016914	
MySQL		0.030256		0.01668875	
MySQL		0.031381		0.017061	
MySQL	**3i**	0.05862725	0.0571284	0.035218	0.03533485
MySQL		0.05776525		0.036236	
MySQL		0.056396		0.03589175	
MySQL		0.05675075		0.03544025	
MySQL		0.05610275		0.03388825	
MySQL	**3j**	0.12845725	0.12815655	0.084478	0.0839713
MySQL		0.1267		0.0858295	
MySQL		0.12618275		0.08290075	
MySQL		0.1299495		0.08310825	
MySQL		0.12949325		0.08354	
MySQL	**4**	0.03439625	0.03606555	0.0320135	0.0326055
MySQL		0.03358025		0.032234	
MySQL		0.03584725		0.03284175	
MySQL		0.04096975		0.0337455	
MySQL		0.03553425		0.03219275	
MySQL	**5**	781.203	783.5784	784.781	786.5094
MySQL		784.11		792.953	
MySQL		780.719		793.047	
MySQL		783.297		781.578	
MySQL		788.563		780.188	
Neo4j	**1**	0.979	0.9546	0.405	0.363
Neo4j		0.932		0.347	
Neo4j		1.026		0.356	
Neo4j		0.901		0.349	
Neo4j		0.935		0.358	
Neo4j	**2a**	0.801	0.8506	0.282	0.249
Neo4j		0.863		0.239	
Neo4j		0.956		0.239	
Neo4j		0.829		0.247	
Neo4j		0.804		0.238	
Neo4j	**2b**	0.865	0.8552	0.312	0.2762
Neo4j		0.899		0.263	
Neo4j		0.867		0.268	

Neo4j		0.913		0.255	
Neo4j		0.732		0.283	
Neo4j	**2c**	1.38	1.3326	0.59	0.5546
Neo4j		1.385		0.55	
Neo4j		1.324		0.56	
Neo4j		1.237		0.534	
Neo4j		1.337		0.539	
Neo4j	**2d**	1.366	1.3326	0.569	0.5366
Neo4j		1.303		0.517	
Neo4j		1.1		0.532	
Neo4j		1.357		0.527	
Neo4j		1.537		0.538	
Neo4j	**2e**	0.875	0.8762	0.227	0.229
Neo4j		0.782		0.233	
Neo4j		0.777		0.231	
Neo4j		1.038		0.222	
Neo4j		0.909		0.232	
Neo4j	**2f**	1.3	1.2138	0.483	0.4132
Neo4j		1.271		0.44	
Neo4j		1.09		0.406	
Neo4j		1.231		0.376	
Neo4j		1.177		0.361	
Neo4j	**2g**	1.167	1.1342	0.323	0.3228
Neo4j		1.319		0.329	
Neo4j		1.02		0.309	
Neo4j		1.014		0.316	
Neo4j		1.151		0.337	
Neo4j	**2h**	1.366	1.1466	0.441	0.4252
Neo4j		1.177		0.415	
Neo4j		1.119		0.521	
Neo4j		1.101		0.399	
Neo4j		0.97		0.35	
Neo4j	**2i**	1.669	1.49	0.67	0.599
Neo4j		1.458		0.612	
Neo4j		1.309		0.571	
Neo4j		1.519		0.569	
Neo4j		1.495		0.573	

Neo4j	**2j**	2.388	2.4946	1.12	1.0886
Neo4j		2.509		1.064	
Neo4j		2.892		1.07	
Neo4j		2.273		1.111	
Neo4j		2.411		1.078	
Neo4j	**3a**	0.827	0.886	0.282	0.2686
Neo4j		0.879		0.265	
Neo4j		0.931		0.27	
Neo4j		0.86		0.266	
Neo4j		0.933		0.26	
Neo4j	**3b**	0.887	0.899	0.322	0.3078
Neo4j		0.899		0.338	
Neo4j		0.885		0.329	
Neo4j		0.974		0.28	
Neo4j		0.85		0.27	
Neo4j	**3c**	1.296	1.3652	0.802	0.7714
Neo4j		1.292		0.765	
Neo4j		1.598		0.761	
Neo4j		1.267		0.766	
Neo4j		1.373		0.763	
Neo4j	**3d**	1.347	1.3484	0.584	0.5528
Neo4j		1.304		0.547	
Neo4j		1.36		0.536	
Neo4j		1.381		0.543	
Neo4j		1.35		0.554	
Neo4j	**3e**	0.957	0.8734	0.241	0.2236
Neo4j		0.744		0.219	
Neo4j		0.872		0.214	
Neo4j		0.881		0.225	
Neo4j		0.913		0.219	
Neo4j	**3f**	1.152	1.2424	0.502	0.4482
Neo4j		1.323		0.464	
Neo4j		1.308		0.465	
Neo4j		1.265		0.419	
Neo4j		1.164		0.391	
Neo4j	**3g**	1.236	1.1802	0.379	0.362
Neo4j		1.274		0.366	

Neo4j		1.172		0.365	
Neo4j		1.098		0.34	
Neo4j		1.121		0.36	
Neo4j	**3h**	1.16	1.1496	0.416	0.3866
Neo4j		1.044		0.386	
Neo4j		1.331		0.386	
Neo4j		1.236		0.386	
Neo4j		0.977		0.359	
Neo4j	**3i**	1.571	1.6332	0.75	0.5932
Neo4j		1.76		0.606	
Neo4j		1.581		0.542	
Neo4j		1.641		0.532	
Neo4j		1.613		0.536	
Neo4j	**3j**	2.51	2.529	1.254	1.2192
Neo4j		2.601		1.218	
Neo4j		2.578		1.183	
Neo4j		2.421		1.227	
Neo4j		2.535		1.214	
Neo4j	**4**	8.352	8.9392	6.608	6.5932
Neo4j		8.628		6.536	
Neo4j		10.745		6.586	
Neo4j		8.761		6.627	
Neo4j		8.21		6.609	
Neo4j	**5**	1649.513	1666.435	1502.476	1505.4304
Neo4j		1668.13		1501.586	
Neo4j		1682.654		1494.196	
Neo4j		1665.882		1511.465	
Neo4j		1665.996		1517.429	

REFERENCES

Abdalla, R. M., & Niall, K. K. (2007). *Review of spatial-database system usability: Recommendations for the ADDNS Project.* Toronto: Defense R&D Canada.

Aghi, R., Mehta, S., Chauhan, R., Chaudhary, S., & Bohra, N. (2015). A Comprehensive Comparison of SQL and MongoDB Databases. *International Journal of Scienctific and Research Publications*, *5*(2).

Amlanjyoti, S., Sherin, J., Dhondup, D., & Roseline, M. R. (2015). Comparative Performance Analysis of MySQL and SQL Server Relational Database SYstems in WIndows Enivironment. *International Journal of Advanced Research in Computer and Communication Engineering, 4*(3), 160-164.

Angles, R., & Gutierrez, C. (2008). Survey of Graph Database Models. *ACM Computing Surveys*, *40*(1), 39. doi:10.1145/1322432.1322433

Bass, B. (2012). NoSQL spatial - Neo4j versus PostGIS. *Geographical Information Management and Applications*.

Batra, S., & Tyagi, C. (2012). Comparative Analysis of Relational and Graph Databases. *International Journal of Soft Computing and Engineering (IJSCE), 2*(2), 509-512.

Bazar, C., & Sebastian, C. (2014). The Transition from RDBMS to NoSQL. A Comparative Analysis of Three Popular Non-Relational Solutions: Cassandra, MongoDB and Couchbase. *Database Systems Journal*.

Clarke, K. C. (2011). *Getting Started with Geographic Information Systems.* Pearson.

Codd, E. (1970, June). A Relational Model of Data for Large Shared Data Banks. (P. Baxendale, Ed.) *Communications of the ACM, 13*(6), 377-387.

Dolton, L. M., & Lowe, J. W. (2001). Prospecting Spatial Database Offerings. *Geospatial Solutions*.

Fowler, A. (2013, Nov). *MarkLogic, huh, what is it good for?…*. Retrieved from NoSQL, Sales Engineering, And Arduino Blog: https://adamfowler.org/2013/11/25/marklogic-huh-what-is-it-good-for/

Gandhi, V., Kang, J., & Shekhar, S. (2007). *Spatial Databases - Technical Report.* Minneapolis: Department of Computer Science and Engineering University of Minnesota.

Guting, R. H. (1994). An Introduction to Spatial Database Systems. *VLDB*.

Healey, R. (1991). *Database Management Systems* (Vol. 1). (D. J. Maguire, M. F. Goodchild, & D. W. Rhind, Eds.) New York: Longman Scientific & Technical.

Jaiswal, G., & Agrawal, A. P. (2013). Comparative Analysis of Relational and Graph Databases. *IOSR Journal of Engineering (IOSRJEN)*, 25-27.

Karlsson, A. (2008). My SQL for GIS Applications. *Geo:Connexion*.

Kumar, L., Rajawat, S., & Joshi, K. (2015). Comparative Analysis of NoSQL (MongoDB) with MySQL Database. *International Journal of Modern Trends in Engineering and Research, 2*(5), 120-127.

Longley, P. A., Goodchild, M. F., Maguire, D. J., & Rhind, D. W. (2001). *Geographic Information Systems and Science.* New York: John Wiley & Sons LTD.

Madison, M., Barnhill, M., Napier, C., & Godin, J. (2015). NoSQL Database Technologies. *International Information Management Association, Inc.*

MarkLogic Corporation. (2016). *Flexible Model Datasheet*. Retrieved from MarkLogic: http://cdn.marklogic.com/wp-content/uploads/2016/09/Flexible-Data-Model-Datasheet.pdf

Miler, M., Medak, D., & Odobasic, D. (2013). The Shortest Path Algorithm Performance Comparison in Graph and Relational Database on A Transportation Network. *Information and Communication Technology - Preliminary Communication*, 75-82.

MongoDB. (2016). *Documents*. Retrieved from Introduction to MongoDB: https://docs.mongodb.com/manual/core/document/#bson-document-format

MongoDB, Inc. (2016). *Calculate Distance Using Spherical Geometry*. Retrieved from MongoDB: https://docs.mongodb.com/manual/tutorial/calculate-distances-using-spherical-geometry-with2d-geospatial-indexes/

MongoDB, Inc. (2016). *FAQ - MongoDB Fundamentals*. Retrieved from MongoDB: https://docs.mongodb.com/manual/faq/fundamentals/

Moniruzzaman, A., & Hossain, S. A. (2013). NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison. *International Journal of Database Theory and Applicaion*.

Nair, R., Chauhan, R., & Vats, M. (2015). Comparitive Analysis of Open Source Spatial Database Systems. *International Journal of Innocative Computer Science & Engineering, 2*(6), 1-3.

Neo4j. (2016). *Neo4j - Product*. Retrieved from Neo4j Spatial: https://neo4j.com/product/

Nixon, K. (2015, February). Sustainable Competitive Advantage: Creating Business Value through Data Relationships. *White Paper*. neo4j.

Oracle. (2016). *Oracle NoSQL Database*. Retrieved from Integrated Cloud Applications & Platform Services: https://www.oracle.com/database/nosql/index.html

Oracle Corporation. (2016). *13.15.9 Functions That Test Spatial Relations Between Geometry Objects*. Retrieved from MySQL - Spatial Relation Functions: http://dev.mysql.com/doc/refman/5.7/en/spatial-relation-functions.html

Oracle Corporation. (2016). *What is MySQL*. Retrieved from MySQL: https://dev.mysql.com/doc/refman/5.7/en/what-is-mysql.html

Padhy, R. P., Patra, M. R., & Satapathy, S. C. (2011). RDBMS to NoSQL: Reviewing Some Next-Genergation Non-Relational Database's. *International Journal of Advanced Engineering Sciences and Technologies*, 15-30.

Penchikala, S. (2013, July 23). *NoSQL Database Adoption Trends*. Retrieved 2016, from InfoQ: https://www.infoq.com/research/nosql-databases

Perdue, T. (2016, April 03). *NoSQL: An Overview of NoSQL Databases*. Retrieved 2016, from Lifewire: https://www.lifewire.com/nosql-an-overview-of-nosql-databases-2495393

Sachedina, A., Huras, M. A., & Romanufa, K. K. (2006, Aug). *USA Patent No. US 7085911 B2*.

Shekhar, S., & Chawla, S. (2003). *Spatial Databases: A Tour*. Upper Saddle River, New Jersey: Prentice Hall.

Singleton, A. M., & Longley, P. (2010). *Developing Efficient We-based GIS Applications*. London: UCL Centre for Advanced Spatial Analysis.

The Apache Software Foundation. (2016). *What is Cassandra*. Retrieved from Apache Cassandra: http://cassandra.apache.org/

The PostgreSQL Global Development Group. (2017). *About*. Retrieved from PostgreSQL: https://www.postgresql.org/about/

Van Oosterom, P., Quak, W., & Tijssen, T. (2002). Testing Current DBMS Products with Real Spatial Data. *Management of Urban and Rural Information, 7*.

Worboys, M., & Duckham, M. (2004). *GIS: A Computing Perspectinve Second Edition.* CRC Press.

Zhou, Z., Zhou, B., Li, W., Griglak, B., Caiseda, C., & Huang, Q. (2009). Evaluating Query Performance on Object-Relational Spatial Databases. *Computer Science and Information Technology* (pp. 489-492). Beijing: 2nd IEEE International Conference on, Beijing. doi:10.1109/ICCSIT.2009.5234509

BIOGRAPHY

Jodi Deprizio received her Bachelor of Science from George Mason University in 2012. She is employed as a Senior Data Analyst in Fairfax County.