Examensarbete

JÄMFÖRANDE ANALYS AV FRÅGOR FÖR ENSKILDA OCH FLERA GEOMETRITYPER FÖR HÄMTNING AV GEOSPATIALA DATA I MYSQL OCH MONGODB

Bedömning av frågeprestanda för platsbaserad information i MySQL och MongoDB

COMPARATIVE ANALYSIS OF SINGLE AND MULTIPLE GEOMETRIC TYPE QUERIES FOR GEOSPATIAL DATA RETRIEVAL IN MYSQL AND MONGODB

Assessing Fetch Query Performance for Location-Based Information in MySQL and MongoDB

Examensarbete inom huvudområdet Informationsteknologi
Grundnivå 30 högskolepoäng
Vårtermin 2023

William Larsson

Summary

The use of databases for managing spatial data is widespread due to the efficiency of traditional SQL databases like *Azure SQL*. However, the exponential growth of data from sources like social media has led to the popularity of NoSQL databases such as *MongoDB* that handle large volumes of data effectively. NoSQL databases, including *MongoDB*, have built-in support for geospatial queries, making them suitable for managing geospatial data. Geospatial data combines geometric and geographic information and is represented by spatial datatypes like Point, LineString, and Polygon. *MySQL* and *MongoDB* both support geospatial data, but limited studies are comparing their performance in geospatial queries. An experiment was conducted to compare the fetch speed of geospatial data in these databases. The results were analyzed using graphs and related studies to draw conclusions, which showed that *MongoDB* performed slower fetch requests than *MySQL*. Future studies can use more data points and different queries.

Keywords: MongoDB, MySQL, Geospatial data, Single-Geometry types, Multiple-Geometry types

Table of Contents

1 Introduction 1

2 Background 2
2.1 SQL and NoSQL 2
2.1.1 SQL 2
2.1.2 NoSQL 3
2.2 Geographical, Geometric, and Geospatial Data 4
2.2.1 Geographic Data 4
2.2.2 Geometric Data 4
2.2.3 Geospatial Data 4
2.3 Spatial data types 5
2.3.1 Point 5
2.3.2 LineString 6
2.3.3 Polygon 7
2.3.4 LinearRing vs Polygon 8

3 Problem 9
3.1 Problem Statement 9
3.2 Hypothesis 10

4 Method 11
4.1 Independent Variables 11
4.2 Dependent Variables 11
4.3 Dataset 11
4.4 Measurements 12
4.5 Network and Hardware 13
4.6 Alternative Methodologies 14
4.6.1 Case Study 14
4.6.2 Simulation 14
4.7 Ethics 14

5 Implementation 16
5.1 Previous Research and Literature Study 16
5.2 Implementation and Progression 17
5.2.1 Preparation 17
5.2.2 Project Structure 18
5.2.3 Table and Collection creation 19
5.2.4 Command Line Parser 20
5.2.5 Data Generation 21
5.2.6 NodeJS 24
5.2.7 Front end 26
5.2.8 Time Measurements 27
5.3 Pilot Study 27
5.3.1 Pilot Study Results 30
5.3.2 Pilot Study Discussion 31

6 Evaluation 32
6.1 Presentation of study 32
6.2 Analysis 33
6.2.1 How does the amount of data affect the query speeds? 33
6.2.2 Which database performed the best overall? 36
6.3 Conclusions 37

7 Final Thoughts 39
7.1 Summary 39

7.2 Discussion 39
7.2.1 Related research 41
7.2.2 Ethical Aspects 41
7.2.3 Social benefit 42
7.3 Future Work 42
References **44**

1 Introduction

The utilization of databases for managing spatial data has become a common practice due to the efficiency and performance of traditional SQL databases such as *Azure SQL* and *PostGIS* in handling and processing spatial data (Laksono, 2018). However, with the exponential growth of spatial data generated by various sources like social media and smartphones, NoSQL databases have gained popularity due to their ability to handle spatial data quickly and efficiently (Laksono, 2018). Several NoSQL databases, including *MongoDB*, have built-in functionality to support geospatial queries (Sharma et al., 2018), thus facilitating the usage of geospatial data within them.

Geospatial data refers to information about objects, events, or other features with a position on Earth (Khalilizangelani & Ghaffarian, n.d.) and can be seen as a combination of both geometric data (shape and size) and geographic data (physical and cultural features). Certain spatial datatypes support geospatial data, which are Point, MultiPoint, LineString, MultiLineString, Polygon, and MultiPolygon. Each serves a different purpose within a map, where Point is a single location, LineString is a wall or border, and Polygon is an area. The Multiple-Geometries meanwhile represent multiple instances of these types.

MySQL, a widely used traditional SQL database for web applications (Swardika & Santiary, 2017), and *MongoDB*, a NoSQL database, both have built-in support for geospatial data (Butler et al., 2016; MySQL :: MySQL 8.0 Reference Manual :: 11.4.2 The OpenGIS Geometry Model, 2023). However, although NoSQL databases are commonly used over traditional SQL, there have been limited studies comparing the performance of geospatial queries between these two databases, making the choice between them difficult. This study was therefore conducted to answer which one yields the fastest query speed using geospatial data and queries.

An experimental approach was used as it allows for a systematic and controlled comparison of different options and was used in similar studies (Baralis et al., 2017; Filip & Cegan, 2020; Laksono, 2018). The experiment compared the databases by inserting generated data and then querying it using a backend server which in turn is connected to a frontend web artifact that sends GET requests. Each data type was inserted and queried one at a time to get the fetch time of each one individually. The data was then shown in a simple tabular format within the artifact and downloaded in a CSV format. This yielded a dataset consisting of response times which could then be analyzed.

The results from the data were finally discussed and conclusions could be drawn based on the times examined, such as what datatypes were the fastest and how both databases overall compared when considering all datatypes. This was done using graphs and other studies to support the discussion and to answer both the hypothesis and the problem statement questions.

2 Background

2.1 SQL and NoSQL

There are two primary database models which are commonly used – SQL and NoSQL. Both have the purpose of handling and querying data for applications, but the way they handle said data differ. Mainly, while the former uses a relational data model, the latter offers a flexible schema and one of many different models such as a key, column, document, and graph-based one (Bhogal & Choksi, 2015; Laksono, 2018; Schmid et al., 2015). In Figure 1 an overview of the two models is described, where the left side shows which model a typical *MySQL* database supports and the right side shows what a typical NoSQL database supports.

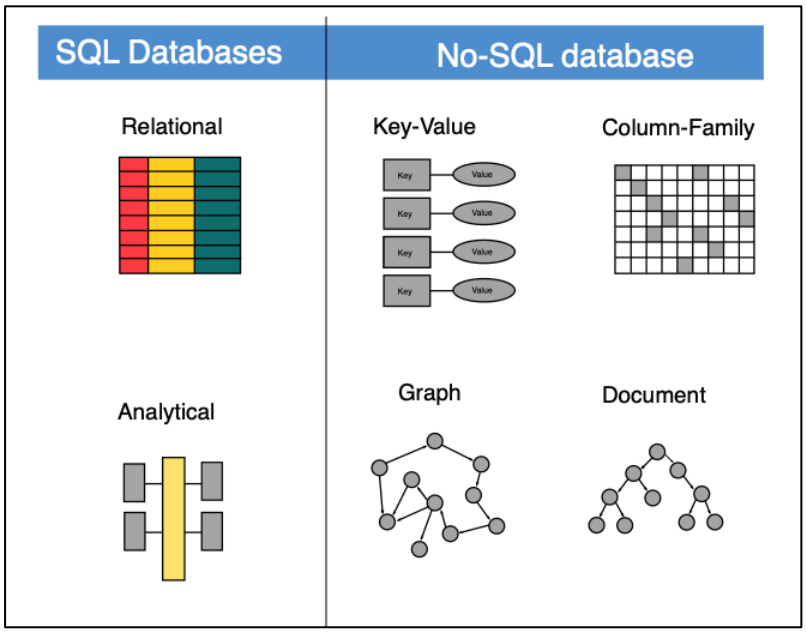


Figure 1 SQL and NoSQL database comparison

2.1.1 SQL

SQL uses a relational approach to handle data as opposed to NoSQL. It is often referred to as "traditional" SQL as it has been around the longest in the industry (Schmid et al., 2015). Rather than scaling horizontally, a traditional SQL database scale vertically (Li & Manoharan, 2013), and instead of adding more instances, the processing power of the server hosting the database instance is increased.

SQL is less commonly used to handle geospatial data types due to its rigid schemas, but it has also been proven that SQL databases sometimes outperform NoSQL databases and that the data model has little correlation with the actual performance (Bhogal & Choksi, 2015). Furthermore, many SQL databases have support for geospatial data types. The database *MYSQL* for example is a relational database that supports many different types of spatial data types, including Geometry, Point, LineString, Polygon, etc. (MySQL :: MySQL 8.0 Reference Manual :: 11.4.1 Spatial Data Types, 2023; Swardika & Santiary, 2017). In Figures 2 and 3, examples of an insert and select statement within *MySQL* can be seen.

```
INSERT INTO table_name (column1, column2, column3)
VALUES (value1, value2, value3)
```

Figure 2 Example of an insert statement in the SQL database *MySQL*

```
SELECT column1, column2
FROM table_name
```

Figure 3 Example of a select statement in the SQL database *MySQL*

2.1.2 NoSQL

NoSQL is used due to its ability to handle semi-structured or unstructured data such as that of spatial data types (Laksono, 2018) as it scales horizontally (spread over many instances) and uses a decentralized architecture. It is often utilized to manage "Big Data", meaning the data is of such magnitude and intricacy that conventional methods of data management become difficult or unfeasible to use for processing (Li & Manoharan, 2013). NoSQL works well for such cases as it provides “speed, flexible schemas, and distributed (i.e., non-centralized) databases” (Li & Manoharan, 2013), which handle big data effectively. In conjunction with geospatial data, a NoSQL database is appropriate as geospatial data tend to be unstructured (Laksono, 2018).

An example of a NoSQL database is *MongoDB* which is a document-oriented database that is written in C++. An entry in the database is stored as an object which in turn is serialized as *JSON*, *XML*, or *BSON*. The objects do not need to have the same structure, fields, or types of fields, which makes the database flexible (Li & Manoharan, 2013). Multiple data types are supported, including `strings`, `ints` (numbers), `booleans`, and most relevantly geospatial data in the form of *GeoJSON*. However, depending on the model, geospatial data queries might differ in structure, but for document-based ones, *GeoJSON* is commonly used. In Figures 4 and 5, examples of an insert and select statement within the NoSQL database *MongoDB* can be seen.

```
db.runCommand(
   {
      insert: "users",
      documents: "[{column1: value1, column2: value2, column3: value3}]"
   }
)
```

Figure 4 Example of an insert statement in the NoSQL database *MongoDB*

```
db.table_name.findOne({column1: value1})
```

Figure 5 Example of a select statement in the NoSQL database *MongoDB*

2.2 Geographical, Geometric, and Geospatial Data

2.2.1 Geographic Data

Geographic data refers to data that describes the physical and cultural features of a location mapped on a sphere such as the Earth (Khalilizangelani & Ghaffarian, n.d.). This data may consist of a latitude and longitude which is used to highlight the position of a physical object on a map, along with other metadata corresponding to that position. For example, modern phones have built-in GPS which creates geolocated or geotagged data (Laksono, 2018; Salharma et al., 2018). An example of this data can be seen in Figure 6, where coordinates are represented by a GeoJSON object. It is possible to add more metadata to these objects than what the example below presents.

```
{
  "_id": ObjectId("5a934e000102030405000000"),
  "name": "The Sizzling Kitchen",
  "location": {
    "type": "Point",
    "coordinates": [
      -73.985428,
      40.748817
    ]
  }
```

Figure 6 Example of geographic data represented by a GeoJSON object

2.2.2 Geometric Data

Geometric data refers to data that describes the shape and size of objects, including information such as points, lines, and polygons. An example of such an object can be seen in Figure 7.

```
{
  "_id": ObjectId("5a934e000102030405000000"),
  "width": 10,
  "height": 5,
  "topLeft": {
    "x": 2,
    "y": 3
  }
}
```

Figure 7 Example of geometric data represented by a GeoJSON object

2.2.3 Geospatial Data

Geospatial data refers to information about objects, events, or other features with a position on Earth (Khalilizangelani & Ghaffarian, n.d.). In a sense, it's a combination of both geometric

and geographic data, as spatial data can be both vector-based and position based, and in essence, any data related to the geographic location of features and boundaries on earth. In databases, a spatial object is often used (see 2.3 which goes into more detail on these objects), which can represent both geographic and geospatial data through points, lines, coordinates, etc. (Baralis et al., 2017). In summary, geographic data is location-based data, geometric data is shape-based data, and geospatial data is a combination of both. Its data structure is very similar to geographic and geometric data, although instead of only choosing between one of these types it may combine them.

(example here)

Many databases do not inherently possess the capability to handle geospatial or geolocated information, but these limitations can be overcome by enhancing or upgrading the database to a spatial database. This involves expanding the database by adding data both geometric and geographic data types. For instance, while *MySQL* has integrated support, other SQL databases, such as *Postgres*, have an extended version known as PostGIS that offers geospatial capabilities (Laksono, 2018).

2.3 Spatial data types

Spatial data types or spatial objects are used in databases to represent different spatial features, such as a position or an area (Gao et al., 2017). Depending on the database these data types may be represented by different formats, such as *GeoJSON* for *MongoDB* and the *Open GIS Geometry Model* for *MySQL* (Butler et al., 2016; MySQL :: MySQL 8.0 Reference Manual :: 11.4.1 Spatial Data Types, 2023). These data types can be separated into two different categories, which are Single-Geometry and Multiple-Geometry types. The primary difference between them is that while the former handles a single instance of said object, the latter represents multiple instances, i.e., an array of objects. Also, while *MongoDB* and *MySQL* use different formats, the data types are the same, although functionality is implemented differently (Butler et al., 2016; MySQL :: MySQL 8.0 Reference Manual :: 11.4.2 The OpenGIS Geometry Model, 2023).

2.3.1 Point

A *Point* is a data type that represents a position of an object (Butler et al., 2016). For example, it can be the position of a car traveling on the road or the position of a café. Figure 8 shows an example in *MySQL* and Figure 9 shows an example in *MongoDB*. In Figure 10, an example of Google Maps using point data to display a position can also be seen.

```
SET @g = 'POINT(1 1)';
```

Figure 8 *MySQL* – Point

```
{ type: "Point", coordinates: [40, 5] }
```

Figure 9 *MongoDB* – Point

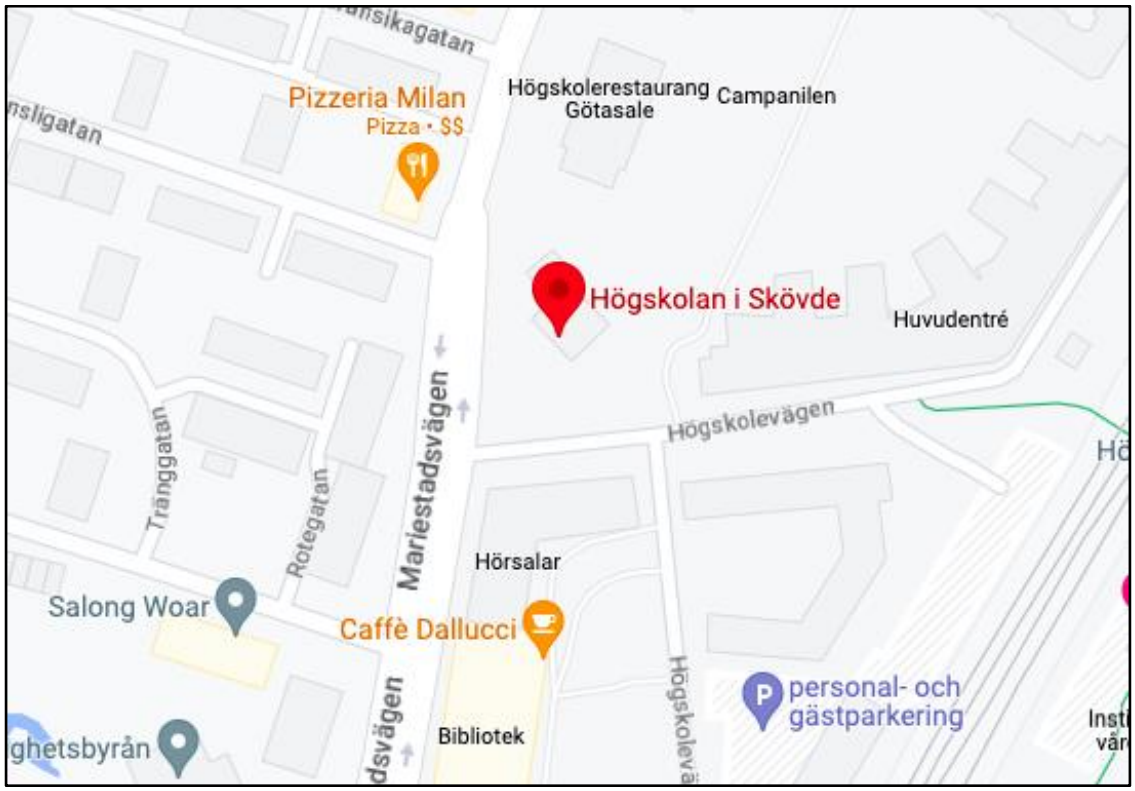


Figure 10 A point displaying the position of *Högskolan I Skövde*

2.3.2 LineString

A *LineString* is a data type that represents linear futures, for example, roads or rivers (Butler et al., 2016). A LineString that is closed, meaning it ends at the same place it starts, is called a *LinearRing*. Figure 11 shows an example in *MySQL* and Figure 12 shows an example in *MongoDB*. In Figure 13, an example of Google Maps using line data to display borders and roads can also be seen.

```
SET @ls 'LineString(1 1, 2 2, 3 3)';
```

Figure 11 *MySQL* – LineString

```
{ type: "LineString", coordinates: [ [ 40, 5 ], [ 41, 6 ] ] }
```

Figure 12 *MongoDB* – LineString

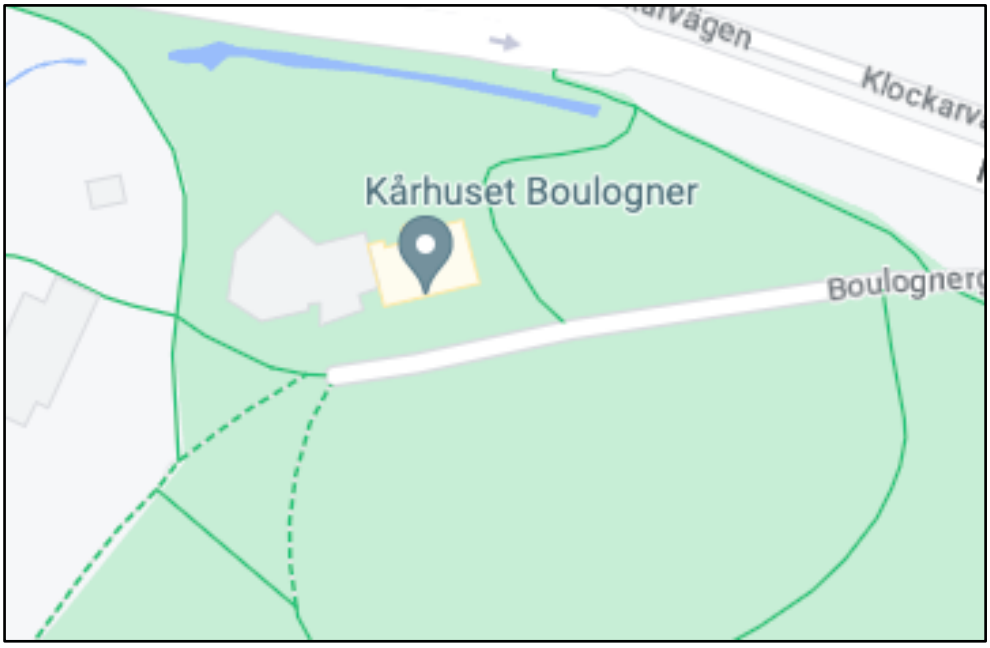


Figure 13 Borders and roads are represented by lines

2.3.3 Polygon

A *Polygon* is a data type that represents a two-dimensional shape that is defined by a set of coordinates/points that encloses an area (Butler et al., 2016). For example, countries, neighborhoods, etc. Figure 14 shows an example in *MySQL* and Figure 15 shows an example in *MongoDB*. In Figure 16, an example of Google Maps using polygon data to display the layout of a shopping center can also be seen.

```
SET @poly = 'Polygon((0 0,0 3,3 0,0 0), (1 1,1 2,2 1,1 1))';
```

Figure 14 *MySQL* – Polygon

```
{
   type: "Polygon",
   coordinates: [
      [
         [[0, 0], [3, 6], [6, 1], [0, 0]],
         [[2, 2], [3, 3], [4, 2], [1, 2]]
      ]
   ]
}
```

Figure 15 *MongoDB* – Polygon

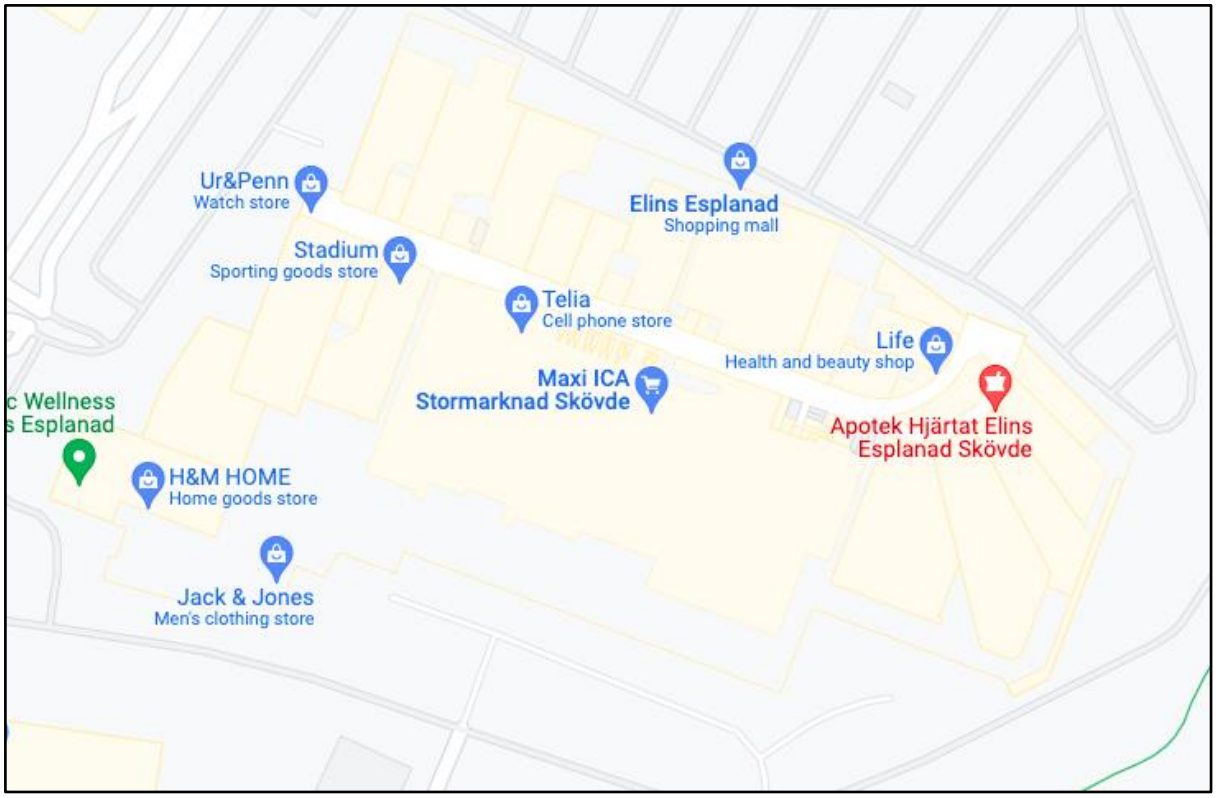


Figure 16 Polygons showing the layout of a shopping center

2.3.4 LinearRing vs Polygon

A LinearRing and a Polygon are similar in that they both define a closed shape, but they are considered to be different geometries (MySQL :: MySQL 8.0 Reference Manual :: 11.4.2 The OpenGIS Geometry Model, 2023). A LinearRing is a set of connected line segments that form a closed loop, with the first and last points being the same, but it does not define an area and is instead used to represent a linear feature, such as a border or a circumference.

A Polygon, on the other hand, is a two-dimensional shape that is defined by a set of coordinates that encloses an area. A Polygon is used to represent regions, such as countries, states, or neighborhoods.

3 Problem

Geolocated data have increased in availability and size over the years, and as a result, more online services have started exploiting the increasing amount of data (Baralis et al., 2017; Zhou et al., 2009). New data is being generated constantly through social media platforms such as Twitter or Facebook in conjunction with the wide support within smart devices and online platforms. As such geospatial data has also become an important part of many citizens-centric services in society like hospitals, banks, disaster management, etc. (Sharma et al., 2018). Chicago City, for example, uses such data to reduce crime by analyzing data in real-time from 30 different departments (Bhogal & Choksi, 2015). However, while the amount of geospatial data has increased, consumers simultaneously expect a low-latency response time, and thus the choice of database to handle geospatial data becomes paramount, as it highly impacts the performance of a deployed application (Baralis et al., 2017).

Today there are many databases to pick from, both traditional SQL and NoSQL. Geospatial data is commonly handled by NoSQL databases due to their ability to handle big and unstructured data, but SQL databases have also been shown to perform the same or even better in some cases (Gyorodi et al., 2015). The opposite has also been proven in studies like "Testing Spatial Data Deliverance in SQL and NoSQL Database Using *NodeJS* Fullstack Web App" by Laksono (2018), which compared *MongoDB* with *PostGIS*. The choice of database model is therefore not obvious.

3.1 Problem Statement

Looking for papers that compare *MySQL* and *MongoDB* within the context of geospatial data querying returns almost no relevant results regardless of the platform used (Google Scholar, IEEE, WOC, etc.). There is a lack of research when it comes to comparing these two databases within this subject area, and thus the reason they've been selected for this thesis.

Some papers such as "Comparison of MySQL and MongoDB with focus on performance" by Filip and Cegan (2020) have compared *MySQL* with *MongoDB* by running queries on geospatial information such as longitude and latitude, but this information was not represented by geospatial-specific objects, but rather generic `string` or `int` types. Other papers have also done comparisons of the two databases in general but omitted any geospatial data types, such as in the study "A comparative study: MongoDB vs. MySQL" by Gyorodi et al. (2015). Although research has been conducted on *MySQL* and *MongoDB*, geospatial data performance comparisons between them are diminutive.

However, prior research has investigated the performance comparison between *MongoDB* and other SQL databases apart from *MySQL*. In the study "Performance Analysis of RDBMS and No SQL Databases: PostgreSQL, MongoDB and Neo4j" (Sharma et al., 2018), the authors conducted a study where they compared two NoSQL databases (*MongoDB* and *Neo4j*) and a SQL database (*PostgreSQL*) in terms of search query speed using *NodeJS* as the framework and *Express.js* as middleware. Sharma et al. (2018) explain that because of the nature of geotagged data, there is no control over it, and thus a traditional database that strives with structured data is not efficient enough. The study concluded that *MongoDB* performed the best out of the two NoSQL databases, especially in handling a high amount of data. It also performed slightly better than *PostgreSQL*.

In another study "Testing Spatial Data Deliverance in SQL and NoSQL Database Using NodeJS Full- stack Web App" (Laksono, 2018), the author compared data deliverance in SQL (*PostGIS*) and NoSQL (*MongoDB*) also using *NodeJS* as the framework. The author explains that the abundance of geospatial big data pushes forward the need for NoSQL databases which are better at handling a larger amount of data. Similarly to the previous study, the author also concluded that *MongoDB* is faster than traditional SQL – that is, NoSQL has better storage and load times.

This thesis will instead compare *MySQL* and *MongoDB* specifically to see if the pattern withholds (e.g., NoSQL outperforms SQL). There are a total of 6 different spatial data types which will be compared, including both Single-Geometry types as well as Multiple-Geometry types, some of which were tested in the after-mentioned studies. Testing both will yield results concerning the handling of these types in both databases. The types are:

- Single-Geometry types
    - **Point**
    - **LineString**
    - **Polygon**
- Multiple-Geometry types
    - **MultiPoint**
    - **MultiLineString**
    - **MultiPolygon**

There are more spatial data types available in both databases, but they are simply different versions of the ones already presented above, making their inclusion redundant. *MongoDB* also has less supported spatial data types overall, meaning there are some available in *MySQL* which cannot be directly compared and are thus omitted. The experiment in this thesis will use queries to extract data from these spatial data types, and aims to answer two central questions:

- Which spatial data type (Point, MultiPoint, LineString, MultiLineString, Polygon, and MultiPolygon) yields the fastest query speed in *MySQL* and *MongoDB* respectively?
- Which of the two databases *MySQL* and *MongoDB* yields the fastest query speed on geospatial data when considering all spatial data types?

3.2 Hypothesis

Although the after-mentioned studies suggest that a NoSQL database model will outperform traditional relational databases in terms of query speed, there is still uncertainty when it comes to *MySQL* as none of the after-mentioned studies ever tested it in relation to a NoSQL database, concerning geospatial data. However, assuming the pattern of NoSQL databases outperforming SQL databases will stay consistent, the hypothesis is as follows:

- There will be a difference between *MongoDB* and *MySQL* in terms of query speed.
- The data type will affect the response time of both databases and Single-Geometry types will be faster than Multiple-Geometry types.

4 Method

To compare both databases and their data types, an experiment methodology was chosen as it allows for a systematic and controlled comparison of different options. It makes it possible to isolate the specific impact of each technology being tested and compare their performance against a common benchmark to identify their strengths and weaknesses, in this case, the aforementioned databases and their query speeds (Wohlin et al., 2012). Furthermore, as experiments are controlled, no unknown variables affect the results as compared to case studies or other methodologies with uncontrollable aspects. Additionally, other studies have used similar methodologies when comparing databases (Baralis et al., 2017; Filip & Cegan, 2020; Laksono, 2018).

The presented method has some resemblance to studies by other authors such as Laksono (2018) and Filip & Cegan (2020) as they provide a good basis for the intended experiment of this thesis. In the following subchapters, the implementation of this experiment is detailed.

4.1 Independent Variables

To conduct the experiment, two distinct backend applications were developed, one for *MongoDB* and the other for *MySQL*. These applications possess identical functionality but, owing to the different implementations of each database, it was necessary to create separate applications rather than a unified one. This approach was adopted to avoid any potential alterations to performance resulting from the presence of redundant code which could cause overhead.

In this case, the two databases are considered independent variables as they are what ultimately affect the dependent variable, i.e. the response time. The applications themselves are also independent variables, as they host the application which in turn connects to a given database, thus also affecting the performance.

4.2 Dependent Variables

The fluctuating variable is the response time for a given query, aka the result of the independent variable (database). The unit of measurement is in milliseconds, as response times can be very small, but using more precise units such as microseconds would yield no useful information anyways as databases do not typically reach those speeds (Baralis et al., 2017; Li & Manoharan, 2013). Furthermore, this unit is representative of the problem statements in Chapter 3.2, as the unit of time is measured, e.g., the query speed.

4.3 Dataset

All data is generated rather than using real data from publicly available sources. While real data provides access to large datasets, thus saving time and resources, they seldom contain all the different data types which are tested. To use real data, different datasets would have to be analyzed to separate the data types, but this runs the risk of the data becoming unbalanced, as one dataset might contain more LineStrings than Points. The data may also consist of unneeded metadata which is not relevant to the experiment, as well as having a completely different data structure. Generated data, however, can be balanced artificially as there is complete control over the data structure and data contents.

To ensure that the geospatial data types are properly tested in both *MySQL* and *MongoDB*, certain considerations must be taken into account due to their structural differences. These considerations can be summarized as follows:

1. Firstly, while a point is a single entity, LineString and Polygons comprise multiple points. Consequently, both Linestrings and Polygons must contain an equivalent number of Points, since the addition of each Point can increase response times. Furthermore, the number of Points must not be less than four, as this is the minimum requirement for a valid Polygon.
2. Secondly, Multiple-Geometry types must be considered, such as MultiPoints, MultiLineStrings, and MultiPolygons. The number of instances within these data types must also be consistent between tests, although there are no minimum or maximum limits to the number of instances permitted. However, as long as there is parity in the number of instances (i.e. the same amount), the tests will be equitable.
3. Finally, single and Multiple-Geometry types may occur multiple times in both databases (i.e. multiple instances), meaning the number of rows in *MySQL* and documents in *MongoDB* will likewise be the same.

The hierarchy of the data structure can be intricate due to the high amount of nesting (e.g., a database comprising numerous MultiLineStrings, each of which contains multiple LineStrings, which, in turn, consist of many points). Therefore, a visual representation of the data structure has been created, which can be seen in Figure 17. The Figure displays an example using two MultiLineStrings, but the hierarchy would look similar when using a Single-Geometry type such as a Point or LineString. The only difference would be that the second layer (in this case, the MultiLineStrings) would be removed.

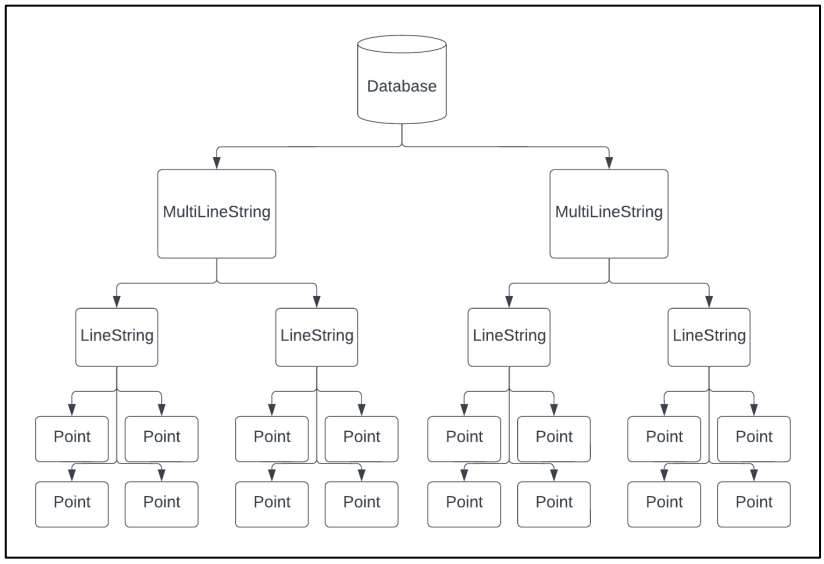


Figure 17 Example hierarchy with two MultiLineStrings

4.4 Measurements

The measurement process involves executing 1000 iterations of queries on each data type in both databases. The reason for the high number of iterations is that'll mitigate the potential for measurement errors such as spikes and ensures that the results obtained are accurate and not coincidental. The queries are dynamically performed by retrieving all geospatial data for a

designated data type in each of the databases, similar to what was done by Laksono (2018). The output of the queries is displayed in a tabular format, including the type of data and its corresponding value. Although spatial data is better represented by a visual format, the use of a map or similar visualization is beyond the scope of this thesis as the focus is on the queries rather than the visual representation of a response.

Measurement of the two databases was initially accomplished by determining the elapsed time from the initiation of the database request to the receipt of the response. The initiation of the request within this context is the moment the database receives a request, such as when a "search" button is pressed on a website. The measurements were then obtained using developer tools in the Firefox browser, as was done in the study conducted by Laksono (2018). By filtering requests, it is possible to get only the response time of the databases themselves and to later export them.

This approach has several benefits, including the widespread availability of the tool in every browser version, making replication of the experiment easier, and the ability to provide precise measurement of query times. The use of alternative tools, such as TamperMonkey, is not ideal as it introduces potential delays in the measurement process as it's separate from the artifact itself, while embedding measurement code directly into the application may also compromise the accuracy of the results due to the potential slowing of the application. By utilizing pre-existing developer tools, the risk of introducing bugs or flawed logic into the measurement process is also eliminated. A screenshot of Firefox developer tools showing the query speed of a *MySQL* get query can be seen in Figure 18.

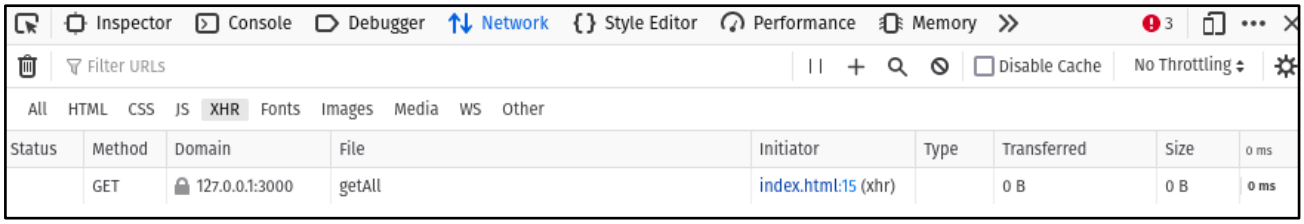


Figure 18 The response time from a get query from *MySQL*

Despite the many benefits of the method, it was ultimately changed as there is no control over the actual measurement process, i.e., when does the browser stop or start the timer? For this reason, code is embedded directly in the application with a small amount of overhead as a tradeoff. It does however yield more accurate data in the sense that the method of measurement is known.

4.5 Network and Hardware

All measurements are done within the same network, and the web artifact as well as all databases are hosted on the same computer, rather than on online services such as *AWS* or *MongoDB Atlas*. This means that not only is there complete control over the built application, but also all the databases. Additionally, no default settings are changed unless required, as this thesis is about testing the query speeds of these databases as equally as possible. Changing one thing in the other might improve or diminish the efficiency of queries accidentally which leads to unfair comparisons.

As the servers and the web artifact are hosted on the same system, there will be a very small delay between request and response. Typically, a server is not a host, but for testing purposes, this method will be sufficient, as the database speeds themselves would not be affected, only the response time to the host. Furthermore, there is a chance that the databases in online services have been altered, and as previously mentioned, hosting the databases gives control over them.

4.6 Alternative Methodologies

Although an experiment was chosen, there are other methodologies to consider as well, all of which provide their respective perks and disadvantages.

4.6.1 Case Study

Whilst an experiment does provide control, it also means that the tests are conducted in an isolated/artificial environment or from a "lab view". In a real-world context, there would be many uncontrollable aspects and many unknown variables that cannot be taken into consideration under a controlled environment (Runeson & Höst, 2009). For example, as mentioned in the problem section, there is a clear lack of research comparing *MySQL* and *MongoDB* within a geospatial context, but that might also indicate a low interest to use *MySQL* for this particular use case. As such, a case study approach could have been more appropriate as it would yield more realistic data.

However, an experiment may be enough to yield valuable insights regarding the performance of the two databases, or at least a general understanding thereof, as it indicates how a real-world system *could* perform. While experiments may be used to focus on the effect of certain factors, their results may be confirmed in replications or big case studies (Basili, Selby, and Hutchens, 1986). A case study provides a more realistic context for analyzing the performance of databases, while an experiment allows for a more controlled and systematic investigation of the factors that influence performance (Wohlin et al., 2012).

4.6.2 Simulation

Müller and Pfahl (2008) describe simulation models as "virtual laboratories where hypotheses about observed problems can be tested, and corrective policies can be experimented with before they are implemented in the real system". In other words, within the context of this thesis, while an experiment performs tests on real databases, simulations tests on virtual ones. This yields some perks over other methodologies, as simulations provide even greater control over the environment and nullify the need to acquire hardware as a single computer can be used to deploy them virtually, thus also making simulations less expensive. However, although simulations have become very accurate (Müller & Pfahl, 2008), they do not provide real-world data, but rather an estimation of the result. Thus, although the result may be accurate, an experiment will always provide real results as they are conducted within a real environment. Furthermore, previous studies comparing both databases with geospatial data have used experiments, rather than simulations.

4.7 Ethics

As the experiment in this thesis is primarily focused on software performance rather than software interaction, no human subject is involved, and some ethics are therefore irrelevant. For example, *informed consent* becomes a non-issue, and the same can be said about

confidentiality as all data is generated rather than taken from real sources (see 4.3), and any correlation with real-world instances is a coincidence.

The non-human considerations, relevant to the experiment in this thesis, include:

1. *Research integrity*, e.g., the "truthiness" of the research and its findings
2. *Validity*, e.g., the "trustworthiness" of the results

Regarding the first consideration, Anderson et al. (2013) explain that the validity of research findings is contingent upon the integrity of the research process, including data collection, analysis techniques, and publication practices. Transparency in the research method is crucial to ensure that no key elements are hidden, thereby preserving the credibility of the results. To address this issue, this thesis provides a clear explanation of the methodology and makes all source code used to get the results available on GitHub. This approach ensures that the methodology is transparent and can be easily replicated and verified by others.

Regarding the second consideration, there are some subcategories of validity (Wohlin et al., 2012):

- Construct Validity: This aspect refers to the accuracy of the operational measures in reflecting the concepts being studied and the research questions being investigated.
- Internal Validity: This aspect is relevant when investigating cause-and-effect relationships. There is a risk that a third factor is affecting the outcome being studied, which could compromise the internal validity of the results.
- External Validity: This aspect addresses the generalizability of the findings and their relevance to other cases or individuals.
- Reliability: This aspect deals with the consistency and dependability of the data and analysis. The study's results should be the same if another researcher were to conduct the same study, but threats to reliability include unclear data coding or questionnaire design.

Generally, validity is about making sure that the result is "valid" in the sense that the results are true and not biased by the researchers' subjective point of view. The after-mentioned subcategories are addressed in this thesis by creating problem statements (see 3.1), taking all variables into account (see 4.1 and 4.2), and making sure that the source code is available.

5 Implementation

This section will discuss the implementation of the web application used to benchmark all databases. First, inspirations and research upon which the implementation is based will be discussed, then the complete application will be explained, and finally, the pilot study will be detailed.

5.1 Previous Research and Literature Study

There were a total of 5 different papers which served as primary information sources and as inspiration for this experiment. These papers were found through means of different search terms primarily made in Google Scholar and IEEE, which includes “Geospatial Data SQL”, “Geospatial Data NoSQL”, “maps MongoDB”, “maps MySQL”, “geospatial data”, etc. These papers contain information that supported the development process and gave ideas previously not considered.

The first inspiration was the paper “Testing Spatial Data Deliverance in SQL and NoSQL Database using *NodeJS* Fullstack Web App” (Laksono, 2018). Not only is the paper relatively new, making it relevant, but the technologies and methods also resembled my initial idea of how the experiment explained in this thesis could be conducted. Although only point data types were used by Laksono (2018), there was still proof that the method was viable, and had the potential to be expanded upon. This paper was found early in the development process and served as a key motivation for the experiment.

Two other papers which inspired this experiment were “SQL versus NoSQL databases for geospatial applications” (Baralis et al., 2017) and “Performance Analysis of RDBMS and No SQL Databases: PostgreSQL, MongoDB and Neo4j” (Sharma et al., 2018). The former discusses many databases, both NoSQL and SQL, and includes descriptions for all supported geospatial data types within each database. Although neither *MongoDB* nor *MySQL* was tested (but discussed), the paper did compare two other NoSQL and SQL databases and yielded some useful information on how to evaluate results and how hardware affects the tests. The latter paper takes a more practical approach and gives examples of how data can be inserted and handled in both SQL and NoSQL databases, including *MongoDB*. Although the implementation presented in the paper slightly differed from the intended one for this thesis, it still provided valuable information on how queries could be used to insert and fetch data. Thus, these two papers supported the development of database management and to some extent data structure.

Finally, there is one additional work that inspired the experiment, which is “API Testing and Development with Postman: A practical guide to creating, testing, and managing APIs for automated software testing” (Westerveld, 2021). The book discussed the application *Postman* which is a free API Development Environment used for testing RESTful APIs and making HTTP requests to a server to check the response. The book primarily explains the usage of the software and, as the title suggests, how to create automated testing for APIs. This yielded some useful information, as the experiment application was going to have functionality like that of Postman, although in a more restricted form. For example, the experiment application was also to send fetch requests, but instead of sending a request to any endpoint, only two would be available (One for *MySQL* and one for *MongoDB*). Similarly, the experiment application was also to display the results of the fetch request somehow, which Postman does by utilizing a table, and thus this type of data visualization is used in the experiment application too.

Furthermore, Westerveld (2021) also explains the verification of SQL queries, and thus Postman also doubles as a testing tool for the experiment. A screenshot of Postman can be seen in Figure 19.

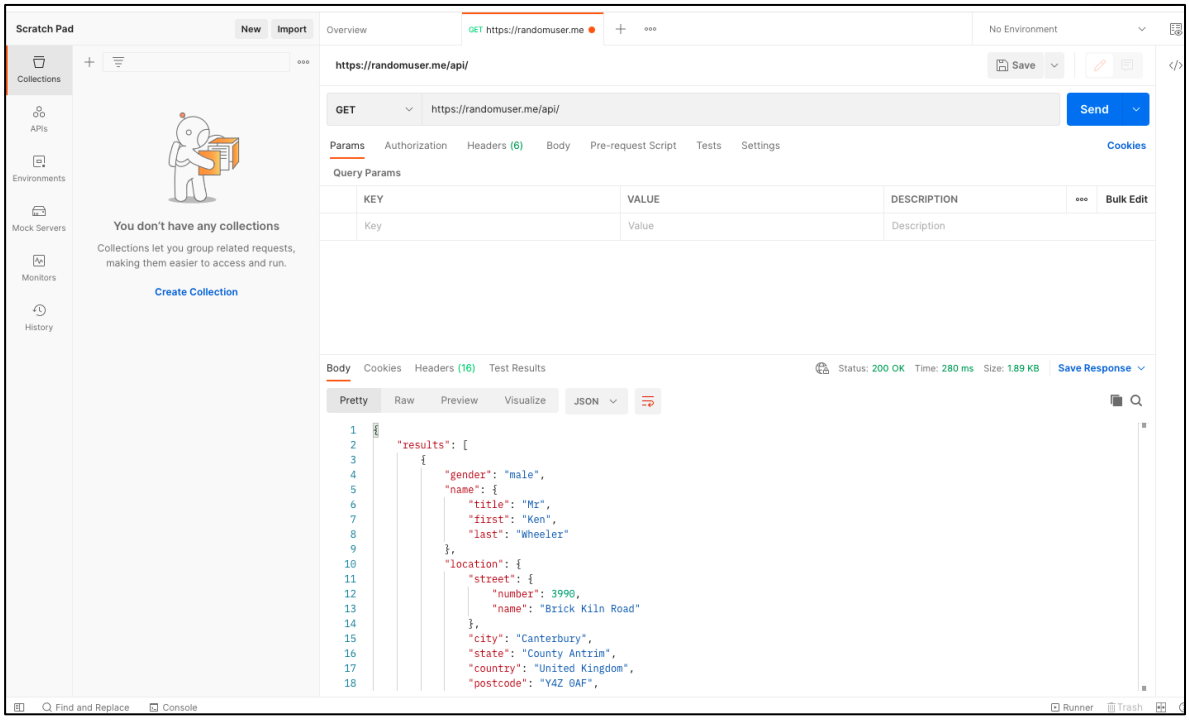


Figure 19 The GUI of Postman

5.2 Implementation and Progression

To test the response times of both *MongoDB* and *MySQL*, the following applications have been created:

- A data generator to create and insert geospatial data into both databases
- Two NodeJS servers that host endpoints
- A web application to make fetch requests against each database and display the result in a tabular format

This section aims to explain these applications, including the thought processes behind their creation as well as essential code and commits.

5.2.1 Preparation

Before beginning to code, a toolset had to be chosen to decide which tools to use when creating the after-mentioned applications. The choice of tools was mostly based on personal preference due to having previous experience with them, but some were also picked based on necessity or to make the development process easier. For example, although data can be generated and inserted into a database using JavaScript and NodeJS, it can be more convenient to choose a programming language that removes the need for asynchronous functions to reduce complexity. That is, some tools can be better for certain jobs, even though both tools could perform them.

The final version of the web application consists of 4 different "modules", which are frontend, backend, *MySQL*, and *MongoDB*. The front end uses pure JavaScript which was generated through TypeScript (v5.0.4). JavaScript works well together with HTML5 due to its ability to not only handle elements such as form inputs but also send fetch requests to the backend. It's also highly accessible, as language support for JavaScript is built into all modern browsers, thus nullifying the need to install anything to start using it as opposed to other languages (ASP.NET requires C# for example). The reason for using TypeScript to generate JavaScript comes down to better handling of bugs as type checking can be used to eliminate potential runtime errors early, thus reducing development time.

The backend meanwhile uses NodeJS (v19.6.0) to host servers for both databases as well as handling endpoints using the library ExpressJS (v4.18.2). Just like the front end, TypeScript was also used for the backend since NodeJS is based on JavaScript instead of PHP or other server languages. It's also easy to install libraries using NPM (Node Package Manager).

Both *MySQL* (v8.0.32) and *MongoDB* (v6.0.5) also had to be installed and started to enable NodeJS to connect to them. This was done using the APT package manager for *MySQL* and by building from source for *MongoDB*. Although *MongoDB* could be downloaded from APT, the official website claimed that the latest version may not be available there yet, and thus the reason for not using it. After installing both databases, the next step involved carrying out necessary setup tasks such as creating users and databases. Once these tasks were completed, the databases were started, thus enabling access to them.

Finally, as the data is generated rather than taken from real sources (see 4.3) a programming language also had to be picked to generate and insert data into *MySQL* and *MongoDB*. For this purpose, Python (v3.10.7) was chosen due to its ability to easily handle files and connect to databases.

5.2.2 Project Structure

Because two databases are being tested, each with separate functions and concepts, the project was essentially split into different folders which represent either *MySQL* or *MongoDB*. The folders share identical names with their corresponding databases, while their contents are comprised of files with the same names but distinct content. To clarify, a tree structure has been generated, which can be seen in Figure 20.

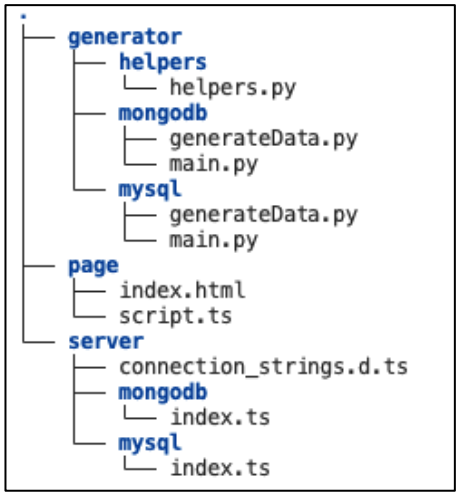


Figure 20 The tree structure of the project

As can be seen, both the “mongodb” and “mysql” folder contains a “index.ts” file in the “server” folder, while they both contain a “main.py” and “generateData.py” in the “generator” folder. The reason for this naming convention is to clarify that the files do the same thing but in different ways. For example, the “generateData.py” file generates the data needed for geospatial queries, but because *MySQL* and *MongoDB* have different syntaxes, they use different files. Throughout the rest of this section when mentioning one of these filenames it will be clarified if it’s the *MySQL* one or *MongoDB* one which is being explained. If it’s not clarified, both files are being referenced.

5.2.3 Table and Collection creation

Before beginning to generate data, all tables and collections had to be created in *MySQL* and *MongoDB*. Initially, this was done manually but was later changed to an automatic implementation to make the initial setup easier, which can be seen in commit *ca54*[1]. This was solved through a Python script called “main.py” (one for each database) which programmatically runs a query that creates a table or collection. Due to convince and security, some values were not hard-coded but rather fetches from a `.env` file that holds information such as the database name, table/collection name, port, etc.

Due to having different fundamental structures, *MongoDB* requires less code than *MySQL* as no tables must be created. Additionally, the *MongoDB* python library creates collections automictically if they do not exist when attempting to connect to them. The code for creating a table in *MySQL* can be seen in Figure 21, while the code to create a collection in *MongoDB* can be seen in Figure 22. Additionally, the `.env` file can be seen in Figure 23.

```
config = dotenv_values(".env")
sqldb = mysql.connector.connect(
    host=config["mysql_host"],
    user=config["mysql_user"],
    password=config["mysql_user_password"],
    database=config["mysql_database"],
)

create_table_query = """
    CREATE TABLE IF NOT EXISTS {} (
        id INT PRIMARY KEY AUTO_INCREMENT,
        type TEXT,
        g GEOMETRY NOT NULL SRID 4326
    )
    """.format(config["mysql_table_name"])
mycursor.execute(create_table_query)
```

Figure 21 Creation of tables in *MySQL*

[1] https://github.com/a20willa/Examensarbete2023/commit/ca54

```
config = dotenv_values(".env")
client = pymongo.MongoClient()
mydb = client[config["mongodb_database"]]
col = mydb[config["mongodb_collection_name"]]
```

Figure 22 Creation of collection in *MongoDB*

```
mysql_host="127.0.0.1"
mysql_user="a20willa"
mysql_user_password="mysql123"
mysql_database="mysql_database"
mysql_port=3306
mysql_table_name="spatial_data_testing"

mongodb_host="mongodb://127.0.0.1:27017"
mongodb_database="mongodb_database"
mongodb_collection_name="spatial_data_testing"
```

Figure 23 The `.env` file used when creating databases and tables/collections

5.2.4 Command Line Parser

Initially, a command line interface was not used at all, but rather all queries were hard coded and generated when the "main.py" (one for each database) Python file was executed. However, this eventually became cumbersome as more data types were added, and thus a command line parser was built to generate and run geospatial queries as it was easier than having to look up the correct line in the code and change it. Command line arguments also made it easier to modify a query on the fly, thus allowing for a highly customizable query. Although the command line parser was initially built in the "main.py" file (see commit *9b87*[2]), it was quickly moved to a separate file called "command_line_parser.py" (see commit *a2b9*[3]) that was eventually renamed to "helpers.py" (outside database folders). This reduced redundancy, as the code could be used by both "main.py" files in each database folder. It also separated different functionality of the codebase into chunks, making development easier.

The only functionality of the command line parser is to return the values of the command line arguments back to the "generateData.py" file, which it does by using the `sys.argv` parameters in Python. Checks are then made to see which arguments were included to run the code accordingly. A snippet of the implementation can be seen in Figure 24, while the "help" text for the command line parser can be seen in Figure 25.

[2] https://github.com/a20willa/Examensarbete2023/commit/9b87
[3] https://github.com/a20willa/Examensarbete2023/commit/a2b9

```
arguments, values = getopt.getopt(sys.argv[1:], "a:t:p:i:s:h", [
                            "amount=", "type=", "points=", "instances=", "seed=" "help"])
for currentArgument, currentValue in arguments:
    if currentArgument in ("-a", "--amount"):
        # Check if value is a number
        try:
            int(currentValue)
        except ValueError:
            print(
                "Invalid value on argument '-a' or '--amount', value must be a number")
            exit(1)

        amount = int(currentValue)

    elif currentArgument in ("-t", "--type"):
                # etc...
```

Figure 24 Command line arguments being parsed in Python

```
Usage: python src/generator/<mysql | mongodb>/main.py [OPTIONS]

Options:
-a, --amount INTEGER: Number of geometries to generate (default: 1)
-t, --type TEXT: Type of geometry to generate, must be one of 'point', 'multipoint', 'linestring', 'multilinestring', 'polygon' or 'multipolygon'
(default: 'point')
-p, --points INTEGER: The amount of points to generate for a linestring or polygon (default: 4, minimum: 4)
-i, --instances INTEGER: The amount of instances of geometries to add in a collection type (default: 1)
-s, --seed INTEGER: The seed used when generating (default: 420)
-h, --help: Displays this text

Example usage:
```

Figure 25 The help text for the command line argument parser tool

5.2.5 Data Generation

To insert data into *MySQL* and *MongoDB*, Python scripts were developed to generate queries for each database. These queries insert geospatial data based on a user's preferences using command line arguments, meaning data generation can be controlled (see 5.2.4). There is one primary script for each database, which is "generateData.py". This script contains a series of functions that generate a query for one of the 6 geospatial data types picked for the thesis. Essentially a query is built through string concentration, where coordinates are dynamically added to an insert statement. The functions to generate coordinates for *MongoDB* and *MySQL* are identical as they are parsed and converted to a valid query later in the process.

The coordinates are generated by simply creating a random number between -90 and 90 for both the longitude and latitude. The reason for not generating larger numbers is due to the limitations enforced by the spatial reference system in *MySQL* which sets certain limits for some SRIDs, in this case, 4326. Initially, these values were up to 180 but were later changed

in commit *dcb2*[4] when geospatial queries had been implemented (discussed later in 5.2.6). Figure 26 shows an example of how a single point is generated within the x and y value of -90 and 90 (note that the parameter "n" has no meaning here).

```
def generate_random_point_data(n):
    # Define the GeoJSON features list
    coordinates = []

    # Generate random coordinates in the range of -90 to 90 (lat and long)
    longitude = random.uniform(-90, 90)
    latitude = random.uniform(-90, 90)
    coordinates = [longitude, latitude]

    return coordinates
```

Figure 26 Random Point coordinates being generated in Python

The main difference between generating a point and other Single-Geometry data types is the number of points. Both a Polygon and a LineString contain many points instead of a single, and thus a simple for loop is used to insert multiple, which can be seen in Figure 27 (note that the parameter "num_points" decides the number of points).

```
def generate_random_linestring_data(num_points):
    # Define the MySQL features list
    coordinates = []

    # Generate random coordinates for the start and end points
    for i in range(num_points):
        longitude = random.uniform(-90, 90)
        latitude = random.uniform(-90, 90)
        coordinates.append([longitude, latitude])

    return coordinates
```

Figure 27 Random LineString coordinates being generated in python

The Polygon data type is special however as the first and last coordinates must be the same and contain at least 4 points. Additionally, the "right-hand rule" must be followed, meaning that exterior rings must be counterclockwise, and holes clockwise. This was solved by creating variables to hold the first value, as well as making some calculations to check if the polygon breaks the right-hand rule, and invert it if it does, thus making the correct side face outwards or inwards (see commit *2f8b*[5]). A snippet of the function generating the Polygon can be seen in Figure 28, where all these rules are resolved. The points themselves are generated in the same manner as the LineString.

[4] https://github.com/a20willa/Examensarbete2023/commit/dcb2
[5] https://github.com/a20willa/Examensarbete2023/commit/2f8b

```
# We need to save the first point
if i == 0:
    firstLine = [longitude, latitude]
    coordinates[0].append([longitude, latitude])
elif i == num_points - 1:
    coordinates[0].append(firstLine)
else:
    coordinates[0].append([longitude, latitude])

# Calculate the polygon's area
area = 0
for i in range(num_points):
    j = (i + 1) % num_points
    area += (
        coordinates[0][i][0] * coordinates[0][j][1]
        - coordinates[0][j][0] * coordinates[0][i][1]
    )

# If the area is negative, reverse the order
if area < 0:
    coordinates[0].reverse()
```

Figure 28 Resolving rules of a Polygon in Python

After generating the coordinates, the final query was built using two primary functions for each database: "generate_collection_of_datatype" and "generate_one_of_datatype." As suggested by their names, the former inserts Multiple-Geometry types, while the latter inserts Single-Geometry types. Both functions read command line arguments from the user and determine which type to generate, but create different queries depending on the database. The primary difference between the two functions is that "generate_collection_of_datatype" uses a for loop to append multiple Single-Geometry types to an array, effectively creating a Multiple-Geometry type. The final query is then returned and executed.

The code for generating Single-Geometry types can be seen in Figure 29, while the after-mentioned for loop to generate Multiple-Geometry types can be seen in Figure 30. Although these figures display code used for *MongoDB*, the same functionality is used for *MySQL*, with only slight differences to the code due to their respective data structure, and thus is not shown in this report.

```
def generate_one_of_datatype(datatype, randomSeed, pointsToGenerate):
    # Define the GeoJSON features list
    seed(randomSeed)
    features = []
    functionToRun = None
    type = ""

    if datatype == "point":
        functionToRun = generate_random_point_data
        type = "Point"
    elif datatype == "linestring":
        functionToRun = generate_random_linestring_data
        type = "LineString"
    elif datatype == "polygon":
        functionToRun = generate_random_polygon_data
        type = "Polygon"

    # Create the GeoJSON feature for the polygon
    geojson = {"type": type, "coordinates": functionToRun(pointsToGenerate)}

    # Return the GeoJSON string
    return json.dumps(geojson)
```

Figure 29 Generation of Single-Geometry type in Python

```
for i in range(int(amountOfInstancesInItem)):
    collection.append(functionToRun(int(pointsToGenerate)))
```

Figure 30 Generation of Multiple-Geometry type in Python by using a for loop

5.2.6 NodeJS

A NodeJS server together with ExpressJS was used to create endpoints that return geospatial data from both databases, which was done using a geospatial query that gets all geometries that exist in the current table or collection. Initially, a geospatial query was not used, but rather a query that fetched all items in the database regardless of the data type, which is a `select * from TABLE` query in *MySQL* and a `find()` query without any arguments in *MongoDB*. The reason for changing to a geospatial query instead (as seen in commit *0863*[6]) was that the after-mentioned queries do not test the fetching speed of geospatial data specifically, but rather the speed to fetch *any* data, and thus the tests would fail to answer the problem statement in this thesis. The geospatial query instead works by setting a point at a specific coordinate and then checking for "near" geometries within a certain distance. As all geospatial data is to be fetched, the distance was not set at all, effectively setting it to an unlimited value which means that the other geometries will be found no matter where their x and y coordinates are. Although fetching everything in a map may be unrealistic for real-world applications, it yields more correct statistics than using the initial method.

[6] https://github.com/a20willa/Examensarbete2023/commit/0863

MongoDB uses a "find" query with an object as a parameter value. This object configures which geometry to find, and which coordinate geometries will be checked among other things. The endpoint which returns the result from this query can be seen in Figure 31.

```
// Dummy endpoint
app_mongodb.get('/getAllMongodb', (req: any, res: any) => {
  // Establish and verify connection
const cursor = dbo.collection(String(data_mongodb.collection_name)).find({
    loc: {
      $near: {
        $geometry: {
          type: "Point",
          coordinates: [-83.5163830345708, 50.646429997753756]
        },
      }
    }
  });

  cursor.toArray((err: any, result: any) => {
    if (err) {
      console.error(err);
      return res.send({ response: "err" });
    }
    res.send({ response: result });
  });
});
```

Figure 31 Endpoint for retrieving all geometries in *MongoDB*

When it comes to *MySQL*, a "select" query is used instead which also gets all geometries within a certain area, once again with an unlimited range. Apart from using a different structure than *MongoDB*, both have the same basic function which can be seen in Figure 32.

```
function getRows(req: any, res: any) {
  // Run get query
  connection.query(`SELECT * from ${data_mysql.table_name} WHERE ST_Distance(g,
ST_GeomFromText('POINT(-83.5163830345708 50.646429997753756)', 4326))`, (err: Error, result: any)
=> {
    if (err) throw err;
    if (result.length != 0) {
      res.send({ response: result });
    } else {
      res.send({ response: "err" })
    }
  });
}
app_mysql.get('/getAllMysql', getRows)
```

Figure 32 Endpoint for retrieving all geometries in *MongoDB*

5.2.7 Front end

The front end oversees sending fetch requests to the NodeJS server, where a response is then received and displayed in a tabular format. This is achieved by using a for loop which creates a series of promises which executes “fetch” statements to a specific IP. The amount of fetch requests created depends on the number of “repetitions” and “datapoint” the user inputs into the GUI, where each repetition is how many times the datapoints are repeatedly inserted, or in other words how many times to repeat the test. The code for this can be seen in Figure 33.

```
for (let e = 0; e < Number(repeat.value); e++) {
    times.push([])

    for (var i = 0; i < Number(itterationsInput.value); i++) {
        let fetch_time = { "start": 0, "end": 0 }
        try {
            fetch_time.start = Number(Date.now())
            fetches.push(await fetch(
                'http://127.0.0.1:3000/' + database + '?cache=' + Math.random(),
                {
                    method: 'GET',
                    mode: 'cors',
                },
            ))
            fetch_time.end = Number(Date.now())
            times[e][i] = (Math.abs(fetch_time.end - fetch_time.start))
        } catch (e) {
            throw new Error(String(e))
        }
    }
}
```

Figure 33 Fetch requests being sent to the backend server

There are some parameters of note within this code. Firstly, the `database` variable represents either *MongoDB* or *MySQL* and is dependent on the value of a select box in the GUI. It decided which server to send the fetch request to, or in other words which database to test. Secondly, the `?cache` query parameter is used to prevent the browser from storing the previous value in the cache, thus causing a faster load time for a given query and compromising the tests. This works by randomizing the cache value of the parameter, which effectively changes the URL and forces the browser to make a new request. This fix was initially discovered and fixed in commit *5147*[7] as a pattern was examined where requests with the `304` status code (Not Modified) would always have a faster response time. Thirdly, the `cors` parameter is used to allow the fetch request to be sent to the local server. Without this option, the host and server would not be able to communicate, as they are on the same computer. This fix was implemented early in the process, as it was discovered that testing only worked on external servers. Lastly, the `await Promise.all(fetches)` code runs all fetch requests in order. The reason for using promises is to wait for each fetch request to finish before starting a new one.

[7] https://github.com/a20willa/Examensarbete2023/commit/5147

5.2.8 Time Measurements

A Firefox plugin was initially created to get all network data from all requests made, which is based on the method used by Laksono (2018). The reason for using a plugin rather than JavaScript in the front end was that Firefox plugins have access to a function called `getHAR()`. This is a function that fetches all network activity made within the current page in the HAR format, which then can be parsed to extract useful data such as response times. The data can then be downloaded by creating an anchor element with the data embedded as an object URL, and pragmatically clicking on it. In this case, the data is reconstructed to a JSON element as it can easily be parsed when creating diagrams. The code for this can be seen in Figure 34, and the commit can be seen in *a9b6*[8].

```
browser.devtools.network.getHAR().then((har) => {
    const harString = { values: [] }

    for (entry of har.entries) {
        if (entry.request.url.includes("getAll")) {
            harString.values.push({ time: entry.time, request: entry.request });
        }
    }

    const file = new Blob([JSON.stringify(harString)], { type: 'application/json' });
    const a = document.createElement('a');
    a.href = URL.createObjectURL(file);
    a.download = 'geospatial_test_data.json';
    document.body.appendChild(a);
    a.click();
    document.body.removeChild(a);
});
```

Figure 34 Network data being downloaded in JSON format

This plugin was later replaced however, as there is no way to know exactly how the times are measured within the browser, thus making it unclear what exactly is being measured. Times were instead measured by using the `Date` class in JavaScript to get the time exactly before and after the request, then getting the difference. The implementation can be seen in the aforementioned Figure 33 and in commit *782b*[9].

5.3 Pilot Study

All tests were initially conducted on a factory reset laptop running Linux Lite, but due to unstable performance and old age, a newer stationary computer was used instead, although not factory reset. All tests were made with only the necessary applications active, which are *MySQL*, *MongoDB*, Vscode (v1.77), and Firefox (v111.0.1 64-bit). The specs of the computer used for testing can be seen in Figure 35 and the operating systems used can be seen in Figure 36.

[8] https://github.com/a20willa/Examensarbete2023/commit/a9b6
[9] https://github.com/a20willa/Examensarbete2023/commit/784b

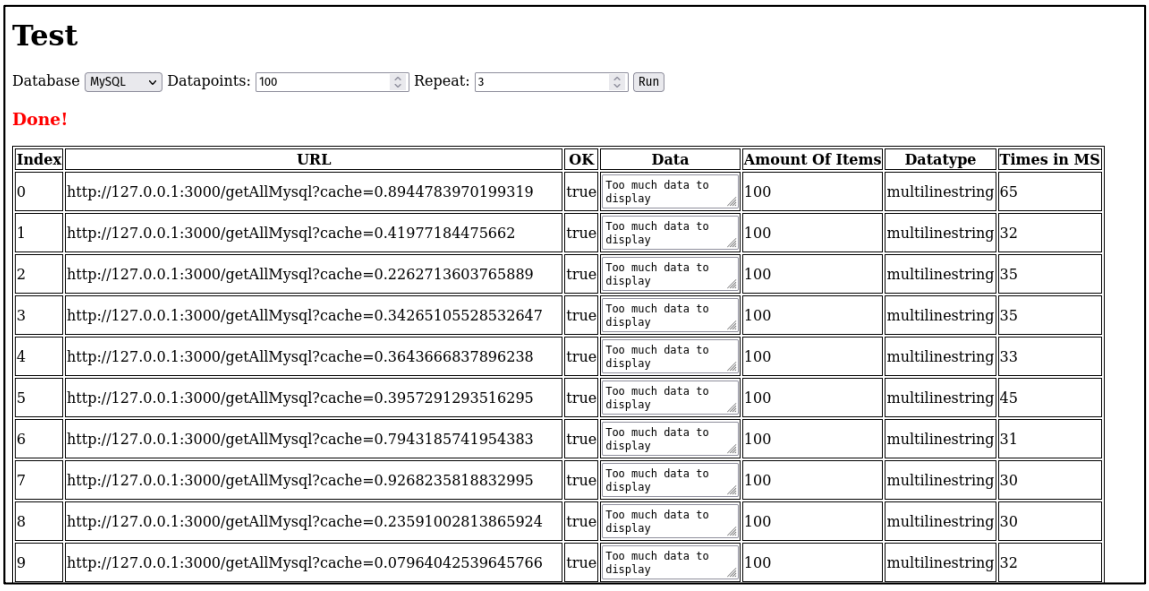
Specification	Description
CPU Type	Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz
GPU	NVIDIA GP106 [GeForce GTX 1060 3GB]
Memory	16 GB 2400 MT/S DDR4

Figure 35 Hardware specifications

Specification	Description
OS	Pop!_OS 22.04 LTS
Kernel	Linux 6.2.6-76060206-generic (x86-64)

Figure 36 Software specifications

To determine whether the chosen method for this thesis would yield useful results, a pilot study was conducted. Within the context of this study, "useful" refers to data that can be used to analyze the differences between the two databases and provides an incentive to continue experimenting on a larger scale. To achieve this, a single web artifact was created for the front end (see 5.2.7), where the user can choose the database to test as well as the number of iterations (i.e., fetch requests) to make. The reason for making only one web artifact is that the only difference between testing *MongoDB* and *MySQL* is the endpoint to call, which is why a select box is used instead. The endpoints themselves (i.e. the servers) are different, however, as they fetch data from different databases, thus requiring different implementations. The web artifact in question can be seen in Figure 37.



Index	URL	OK	Data	Amount Of Items	Datatype	Times in MS
0	http://127.0.0.1:3000/getAllMysql?cache=0.8944783970199319	true	Too much data to display	100	multilinestring	65
1	http://127.0.0.1:3000/getAllMysql?cache=0.41977184475662	true	Too much data to display	100	multilinestring	32
2	http://127.0.0.1:3000/getAllMysql?cache=0.2262713603765889	true	Too much data to display	100	multilinestring	35
3	http://127.0.0.1:3000/getAllMysql?cache=0.34265105528532647	true	Too much data to display	100	multilinestring	35
4	http://127.0.0.1:3000/getAllMysql?cache=0.3643666837896238	true	Too much data to display	100	multilinestring	33
5	http://127.0.0.1:3000/getAllMysql?cache=0.3957291293516295	true	Too much data to display	100	multilinestring	45
6	http://127.0.0.1:3000/getAllMysql?cache=0.7943185741954383	true	Too much data to display	100	multilinestring	31
7	http://127.0.0.1:3000/getAllMysql?cache=0.9268235818832995	true	Too much data to display	100	multilinestring	30
8	http://127.0.0.1:3000/getAllMysql?cache=0.23591002813865924	true	Too much data to display	100	multilinestring	30
9	http://127.0.0.1:3000/getAllMysql?cache=0.07964042539645766	true	Too much data to display	100	multilinestring	32

Figure 37 The pilot study application

Only a small amount of data was created and inserted into both *MySQL* and *MongoDB*, and a total of 900 fetch requests were performed for each database (i.e., 100 data points x 3 repetitions x 3 data points). As the tests are only supposed to indicate if the chosen method is viable, the low amount of fetch requests is sufficient. After running a test with the “Run” button, a table will be shown that displays the following information:

- An index indicating in which order the request was made in
- An URL to the endpoint
- An OK status that indicates if the fetch request was successful (i.e., a response was received)
- The number of items fetched, for example, 100 points
- The data type of the items fetched
- The time it took for the query to complete

5.3.1 Pilot Study Results

The data obtained from the pilot study web application is provided in the form of a text file, which includes the duration of each request completion. The measurements were conducted by performing tests on each data type within each database, repeating the process 100 times over the course of 3 test cycles. Consequently, a total of 300 measurements were obtained for each data type. These individual time values were subsequently averaged to derive a single data point for each repetition. Specifically, the 100th, 200th, and 300th data points were averaged to obtain a representative value. This approach resulted in the generation of 100 data points, which were subsequently exported and utilized for analysis. The precise query employed to generate the data can be found in Appendix A.

In Figure 38, pictures A and B display the results for *MySQL* while pictures C and D display the results for *MongoDB*.

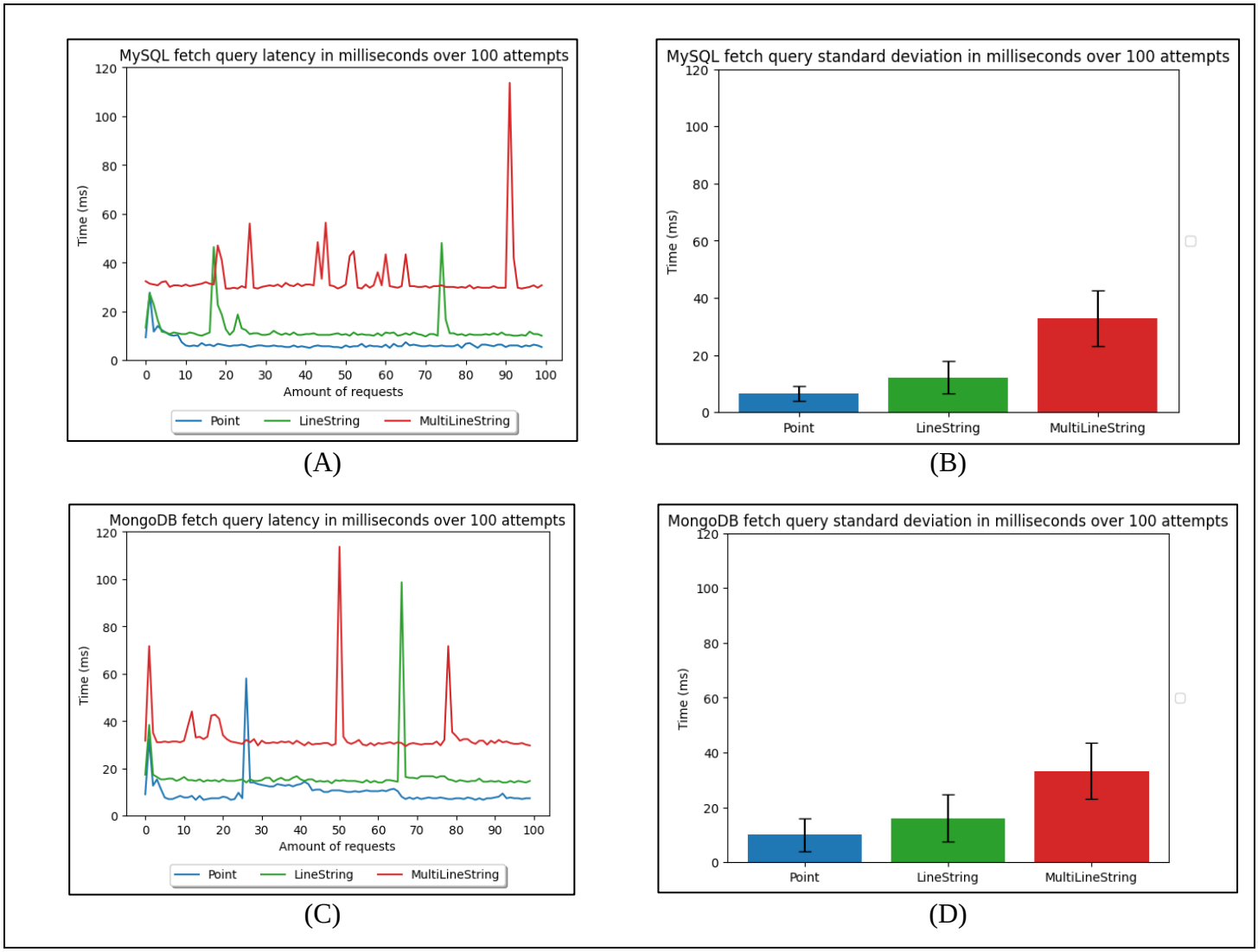


Figure 38 The pilot study results for *MySQL* and *MongoDB* pictures A and B display a line chart and STD diagram for *MySQL*, while pictures C and D show the same for *MongoDB*

5.3.2 Pilot Study Discussion

The primary objective of the pilot study was to determine whether there was sufficient motivation to pursue a larger-scale experiment by analyzing the results of a small number of queries conducted on each database. The findings derived from the study are deemed to be valuable and incentivizing, as they challenge the assertions of multiple papers, such as ”Testing Spatial Data Deliverance in SQL and NoSQL Database Using NodeJS Fullstack Web App” by Laksono (2018), which claim that NoSQL databases outperform SQL databases. Specifically, the initial pilot study suggests that *MySQL* have better performance, as measurements on *MongoDB* in Figure 38 picture A and B show lower speeds than *MySQL* in picture C and D. However, an ANOVA test has also been performed, to verify that there is a difference between the measurements of both databases, which can be seen in Figure 39. The resulting p-value became 1.66457607e-07 for the Point data type, 0.00017211 for the LineString data type, and 0.7692465 for the MultiLineString data type. Thus, the means are different for both the Point and LineString data types, which means the null hypothesis can be safely denied, as the p-value is below 0.05. The same cannot be said for the MultiLineString values, however, as the p-value was above 0.05, thus approving the null hypothesis. There is a chance that this is a result of random chance as there are many spikes that affect the result, and further testing may reveal different data. However, there is enough evidence to suggest there is a difference between the two database fetch speeds, although not a major one.

Datatype	**ANOVA**	**p-value**	**Result**
POINT	29.47822012	1.64457607e-07	The means are different
LINESTRING	14.66668793	0.00017211	The means are different
MULTILINESTRING	0.08629686	0.7692465	No differences in means

Figure 39 ANOVA values between MySQL and MongoDB geospatial datatypes

One reason for this disparity (i.e., that studies claim *MongoDB* is significantly faster) may be attributed to *MongoDB's* superior handling of larger datasets, while *MySQL* excels at processing smaller ones. It is plausible that *MySQL* may struggle when a larger-scale experiment is conducted, as Bhogal and Choksi (2015) explained that NoSQL databases are particularly adept at handling big data, and this experiment may not have reached that threshold. In other words, *MongoDB* has not had the opportunity to demonstrate its optimal performance. To ascertain whether this is the case, a larger experiment must be conducted. Thus, the pilot study was successful in providing valuable data that supports the need for further investigation.

6 Evaluation

The pilot study results showed that *MySQL* and *MongoDB* could be compared, as they yielded different and incentivizing results. Specifically, *MySQL* had a lower response than *MongoDB* by a small margin when measuring the three data types Point, LineString, and MultiLineString, which directly contradicts the conclusions drawn from other studies claiming NoSQL databases are faster than SQL databases when querying geospatial data. In the upcoming test, the number of data points will be increased as it will lessen the chance of randomness having an effect, thus giving a more accurate result. It will also increase the "baseline", as loading times naturally will become higher as more data needs to be queried. These tests aim to answer the problem statement in this thesis and to either accept or deny the hypothesis.

6.1 Presentation of study

As previously mentioned, the pilot study showed there is an incentive to continue to a bigger study, where "bigger" in this case means more data points, more data per query, more iterations, and more data types being tested. Specifically, the amount of data points is increased from 100 to 1000, the amount of data per query is increased from 100 to 500, the number of iterations is increased from three to five and the data points tested are Point, MultiPoint, LineString, MultiLineString, Polygon, and MultiPolygon. Other parameters, such as the amount of data and seed, are not changed to preserve consistency from the pilot study. One exception to this is the number of points within a LineString and MultiPolygon, which has been reduced from 20 to four. The reason for lowering the number of points primarily comes down to the amount of time it takes to make a test, but also due to a programming error where generating more points would sometimes cause an error within *MongoDB* where walls within a polygon geometry would collide, thus halting the insertion process altogether. The artifact configuration for tests, as well as the configuration for the data generated (see 4.3), can be seen in Figure 40. The exact queries used to generate the data can also be seen in Appendix B.

Description	**Value**
Datapoints	1000
Repetitions	5
Number of geometries	500
Number of points in a LineString or Polygon	4
Number of instances in Multiple-Geometries	5
Seed	200
Data types being tested	Point, MultiPoint, LineString, MultiLineString, Polygon, MultiPolygon

Figure 40 Testing details

The tests are being conducted on hardware described in Figure 36, and all applications except VsCode and the Mozilla Firefox Browser are turned off.

6.2 Analysis

This chapter will attempt to answer the problem statement and validate or deny the hypothesis based on the results from the final study. The first question regards the fetch time for each data type within each database.

6.2.1 How does the amount of data affect the query speeds?

To check which of the data types performed the best within their respective databases, 1000 measurements were made against each data type over 5 iterations. The results were averaged out to remove potential spikes which may occur from one value being significantly higher than the other, thus yielding 1000 data points per data type. Seeing as a line diagram makes it hard to compare times, STD bar charts have been created instead, which further average the 1000 data points for a given data type to create a single value. The results for *MySQL* can be seen in Picture A while the result for *MongoDB* and be seen in Picture B in Figure 41.

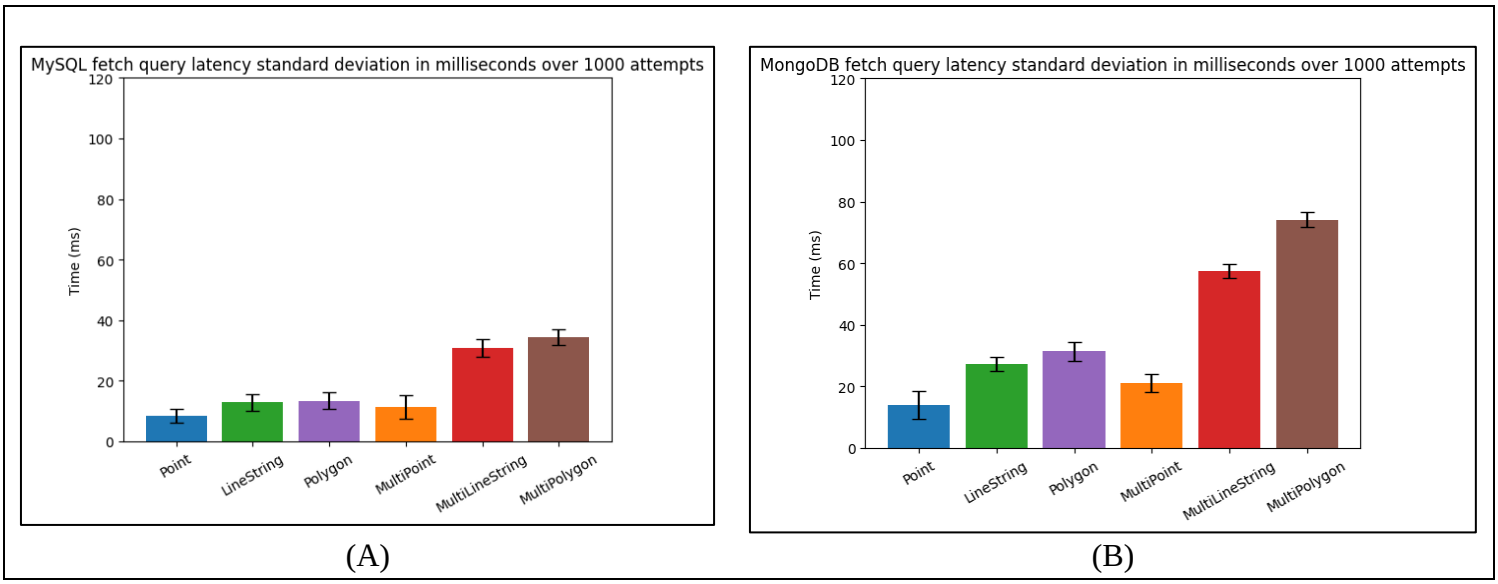
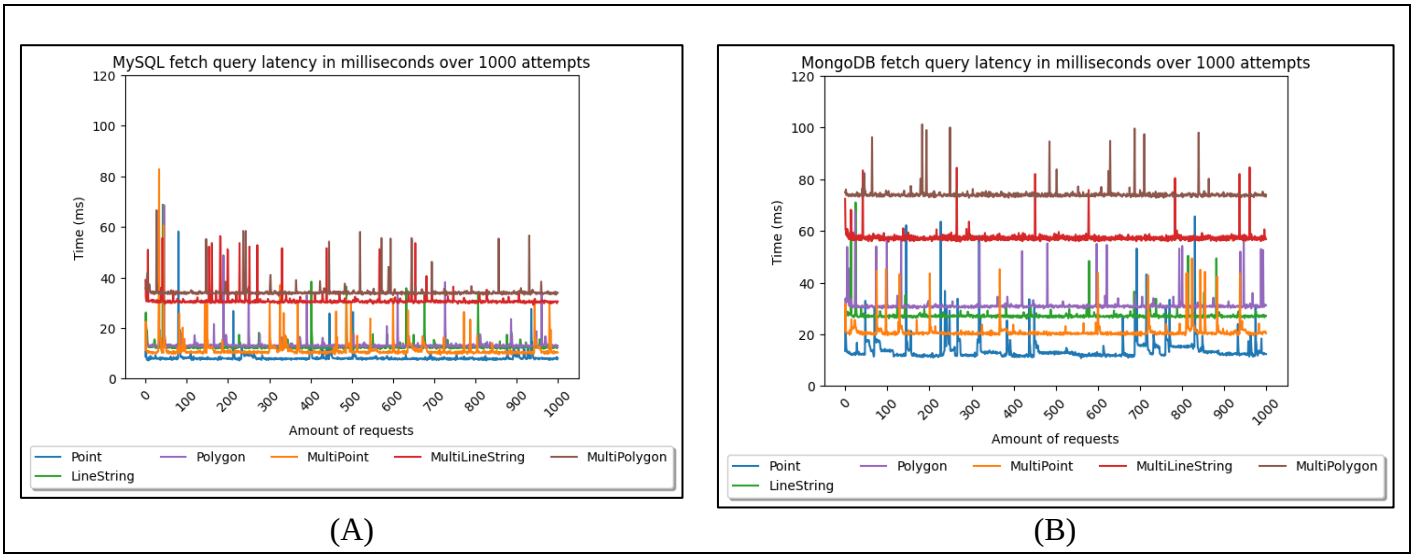


Figure 41 *MySQL* and *MongoDB* standard deviation for all data types

As can be seen in the above charts, the observed pattern aligns with the findings of the pilot study. It demonstrates that an increase in the number of points leads to a longer query time, thereby explaining why the Point data type exhibits the fastest performance while Multiple-Geometry types exhibit the slowest performance. The overlapping standard deviations for certain data types in both *MySQL* and *MongoDB* suggest that the queries are completed within a similar timeframe, which is expected as some data types, such as LineString and Polygon, have an identical number of points, leading to the overlap in standard deviations.

Notably, the MultiPolygon data type exhibits slower response times compared to MultiLineString in both databases, despite having the same number of points, where the only distinction between them is their structure. This discrepancy may indicate that querying a MultiPolygon takes longer than querying a MultiLineString, although to ensure that any spikes or anomalies do not affect the results, a line chart was employed to assess whether the performance remains consistent throughout the entire test, which can be seen in figure 42 where the line chart for *MySQL* can be seen in picture A and the line chart of *MongoDB* can be seen in picture B. Although there are many spikes, a generally linear trend can be observed for all data types. Additionally, after performing ANOVA tests (which can be seen in Figure 43), the p-values for all data types were found to be below or exactly 0, indicating a significant

difference in query times among the tested data types, thus confirming that the spikes did not have a big enough effect to change the outcome if they were not there. Tuckey tests were also made to check what groups does or does not overlap within the respective databases, which can be seen in Figure 44 for MySQL and Figure 45 for MongoDB.



(A) (B)

Figure 42 Line charts of *MySQL* and *MongoDB* performance over time

Datatype	**ANOVA**	**p-value**	**Result**
POINT	1185.20022991	2.50930617e-204	The means are different
LINESTRING	16855.02659251	0	The means are different
POLYGON	17873.25499775	0	The means are different
MULTIPOINT	3816.49038982	0	The means are different
MULTILINESTRING	50096.79316997	0	The means are different
MULTIPOLYGON	126737.37956641	0	The means are different

Figure 43 ANOVA values between MySQL and MongoDB geospatial datatypes

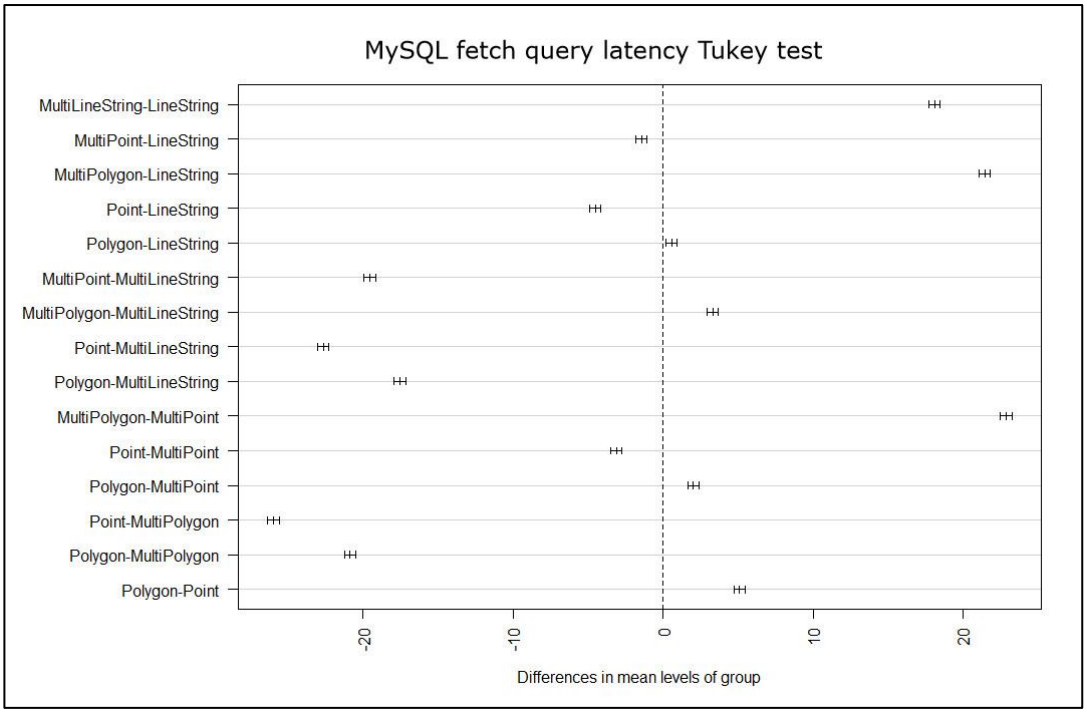


Figure 44 Tukey test of fetch query latency in MySQL

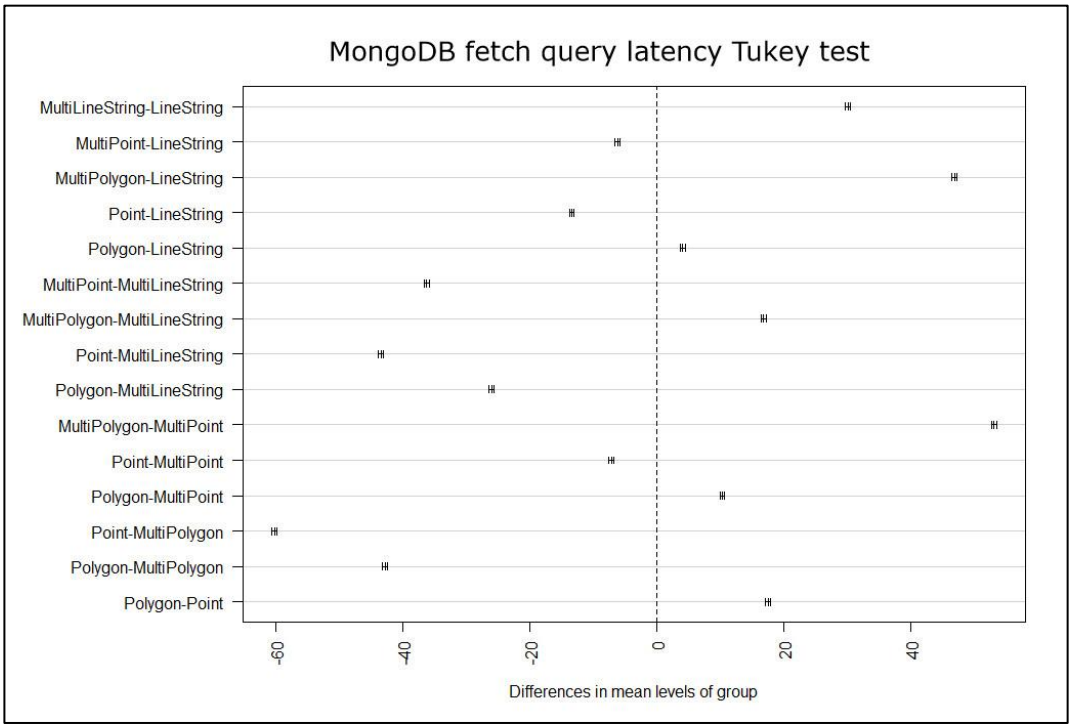


Figure 45 Tukey test of fetch query latency in MongoDB

To make sure that the number of points correlates to the response time, a scatter chart was also made which maps the number of points with speed using the after-mentioned data types. The same results can once again be examined, where Single-Geometries (e.g., Point) is the fastest and Multiple-Geometries are the slowest as dependent on their number of points. The Polygon data type can once again be seen as having a higher value than the LineString despite having

the same number of Points, somewhat proving that it's handled differently by the database queries. The scatter chart for *MySQL* can be seen in picture A and the scatter chart for *MongoDB* can be seen in picture B in Figure 46.

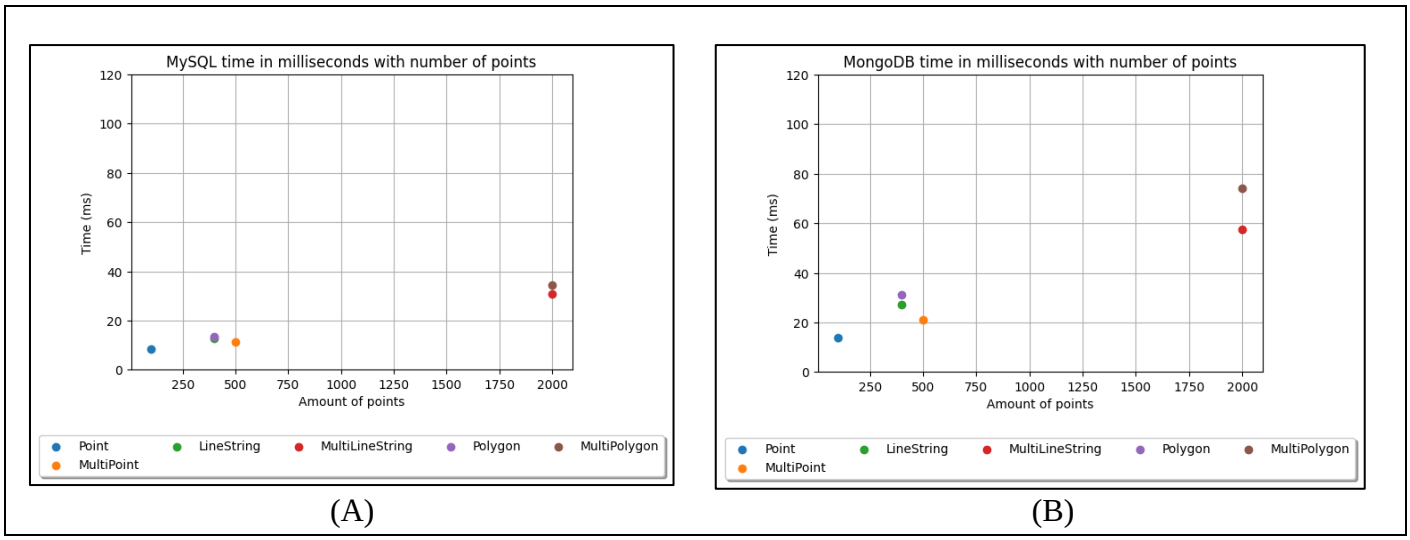


Figure 46 Scatter diagram showing the correlation between the number of points and time

6.2.2 Which database performed the best overall?

To answer the question of which database performed the best overall, the means of all measurements for each database was calculated to get the average query time when considering all data types, which is seen in Figure 47. As expected, based on the charts in Figures 42 and 46, *MongoDB* overall had a slower query speed, as all data types had a higher query time in *MongoDB* compared to *MySQL*.

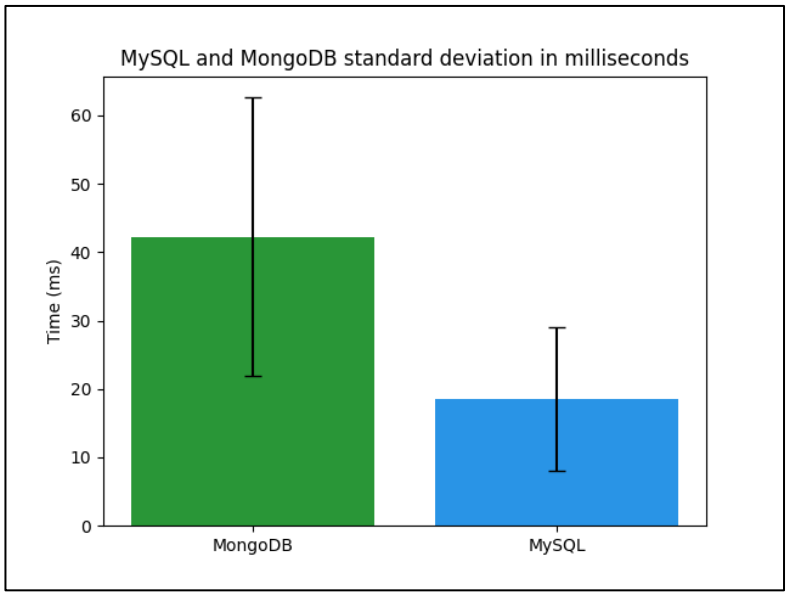


Figure 47 *MySQL* and *MongoDB* standard deviation

The standard deviation shows that the values fluctuate a lot, which is expected as all data types are taken into consideration (i.e., Point has a very low average response time and Polygons has

a very high average response time). Using a SE chart, however, reveals that there is no chance that the speed of both databases would be similar and overlap, as can be seen in Figure 48. The p-value from ANOVA is also 0 since there is such a big margin between the speed of both databases, which can be seen in figure 49.

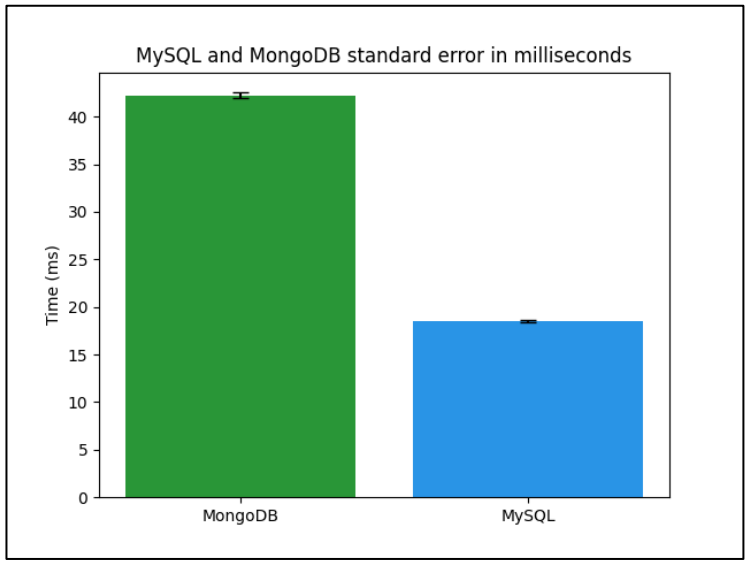


Figure 48 *MySQL* and *MongoDB* standard error

Database	ANOVA	p-value	Result
MySQL		0	The means are different
MongoDB		0	The means are different

Figure 49 *MySQL* and *MongoDB* standard error

6.3 Conclusions

The purpose of the study was to compare the speed of geospatial fetch queries in both *MySQL* and *MongoDB* by using a geospatial dataset. Different geospatial data types have been tested to yield insight if, why and how they differ in speeds, and if there are any differences or similarities between both databases in this aspect. Additionally, testing has also been made to see if the amount of data could explain the disparities examined in the Pilot Study, or if the data indeed indicated a true result. Based on the results, it is now possible to answer the questions in the problem statement and to approve or reject the hypothesis, of which there are two:

H1: There will be a difference between MongoDB and MySQL in terms of query speed.	**Approved**, since MySQL had a much faster fetch speed than MongoDB
H2: The data type will affect the response time of both databases and Single-Geometry types will be faster than Multiple-Geometry values.	**Approved**, since Single-Geometry types had a faster fetch speed than Multiple-Geometry types due to the difference number of points

The first and primary question is "*Which spatial data type (Point, MultiPoint, LineString, MultiLineString, Polygon, and MultiPolygon) yields the fastest query speed in MySQL and*

MongoDB respectively?". Based on the results, it can be safely concluded that the data type Point always will yield the fastest response time as results have shown that the fewer points a data type inherits the less time it will take to query it. The reason the Point data type is the quickest comes down to the fact it far has the smallest number of points, which as indicated by its name, is one. This also approves of the second hypothesis as the data type indeed influence the response time. Furthermore, Single-Geometry types such as Point, LineString, and Polygon indeed had a faster response time than Multiple-Geometry values such as MultiPoint, MultiLineString, and MultiPolygon, thus approving that theory as well.

The second question is "*Which of the two databases MySQL and MongoDB yields the fastest query speed on geospatial data when considering all spatial data types?*". Looking at the results there is indeed a difference between the query speed of both databases, as it took longer for all data types to be fetched in *MongoDB* than MySQL, effectively proving that *MySQL* handles geospatial queries better using its geospatial queries. This also confirms the first hypothesis which claims that there will be a difference in query speeds meaning that the null hypothesis can be denied.

In the end, the number of Points within a data type seems to be the deciding factor of the query speed for a data type, but the speed this occurs at differs between databases. The study has proven that when comparing these two databases using the after-mentioned methods, applications, and data, *MySQL* comes out faster, but further testing using even bigger datasets or different methods and applications may produce other results.

7 Final Thoughts

7.1 Summary

This study aimed to compare the query speed of the SQL database *MySQL* and NoSQL databases *MongoDB* using geospatial datasets and queries. The work was motivated by the fact that the availability and size of geolocated data have increased over the years, leading to the exploitation of such data by many services. Social media platforms like Twitter and Facebook, along with smart devices and online platforms continuously generate new data, and various citizen-centric services such as hospitals, banks, and disaster management use such data extensively and have become crucial for their operations. However, while the amount of geospatial data grows, consumers still expect quick response times. Therefore, selecting an appropriate database to handle geospatial data is crucial as it significantly impacts application performance.

To compare the databases and their data types, an experimental methodology was chosen, as this approach allows for a systematic and controlled comparison of different options. By isolating the specific impact of each technology being tested, it becomes possible to compare their performance against a common benchmark and identify their strengths and weaknesses, specifically in terms of query speeds. This methodology ensures that no unknown variables affect the results, unlike case studies or other methodologies with uncontrollable aspects. Similar methodologies have been employed in previous studies comparing databases.

The results confirmed hypothesis *H2* which claimed that Single-Geometry types were faster than Multiple-Geometry types, as can be seen in the bar charts and ANOVA tests. Furthermore, *MySQL* demonstrated faster query speeds across all spatial data types compared to *MongoDB*, thus confirming hypothesis *H1* which claims there would be a difference in speed between the two databases. It was concluded that the number of points within a data type was the primary factor influencing query speed, with *MySQL* outperforming *MongoDB* in this aspect, but further testing with different datasets and methodologies may yield alternative findings.

7.2 Discussion

This thesis set out to compare the two databases *MySQL* and *MongoDB*, more specifically their query speed when using geospatial queries on geospatial data. To that end, the thesis could be considered a success, as it indeed was possible to generate statistically significant measured data and based on it create a verdict, i.e., which database performed the best. However, although measurements were made, some aspects may challenge the validity of how they were performed. For one, it is unclear if the geospatial queries used are directly comparable as there are other functions available to query data within both databases. The geospatial query methods used were only one of many, and using others may yield different results. The reason this was not taken into consideration during the study was mainly due to time constraints, as I realized rather late that the query methods I initially used were invalid (see 5.2.6). Therefore, I had to pick new ones rather quickly, without the time to properly check in detail what exactly they do. This is worth controlling in future studies, and perhaps even multiple different geospatial query methods could be compared too. Another problem with the queries specifically is that they are used in a rather unrealistic way as all geospatial data in the entire database is fetched. In any real-world application, only the needed data would be fetched to avoid unnecessary load, as a user would typically only need a small part when using Google Maps or similar services, and this may be the very way the designers of the databases meant for the queries to

be used. This has not been tested in any capacity, and the results could be different if queries were used this way. The motivation behind getting all data comes down to consistency, as the same amount of data must be received across tests to balance them. This could of course be achieved if data was more carefully generated, but time restrictions once again hindered this progression. Thus, the results shown may not reflect the response time of all query methods and may also be considered unrealistic.

Regarding unrealistic results, all the after-mentioned tests were made on a local computer, which may result in unrealistically fast response times. The host and user are seldom on the same local computer, as geospatial services such as Google Maps or Microsoft Maps often serve their content over a network. Thus, the times examined in this thesis can be considered unrealistic as they yield extremely fast times since there is virtually no travel time between the database and the backend. Using a service such as AWS could have provided a more realistic outcome as it not only is popular and thus often used but also provides a "gap" between the computer and server.

When considering the results themselves, a certain level of uncertainty arises regarding their validity, as both databases exhibit numerous spikes of unknown origin, as depicted in Figure 41. Despite several attempts to ascertain their cause, no definitive conclusions could be reached. One such effort involved investigating whether imbalanced data could be responsible for inflating the recorded times. However, even when querying databases containing no data whatsoever, these spikes would manifest. Another investigation was focused on determining whether the spikes originated from the server itself, but no evidence of their presence was found there, indicating a potential association with the front end. Notably, the use of developer tools did identify the presence of spikes, yet failed to provide any insights into their origin. Lastly, an examination was conducted to ascertain whether the issue could be attributed to excessively rapid requests. However, even with a deliberate one-second interval between each request, the spikes persisted. Only after all tests were made was the cause finally found, which turned out to be due to plugins in Firefox which not only was the cause of the spikes but also added overhead. To confirm this, a test was made where 500 data points were created by querying a LineString in *MongoDB* using the normal browser first and then incognito mode which had no addons active. By comparing them, it becomes clear that the plugins indeed were the cause, as the spikes are completely absent in incognito mode, which can be seen in Appendix C. The spikes are not common enough to have a big effect on the test results however, and as it stands the results would be the same since all measurements were made on the same conditions, thus having the same overhead. Of course, if there were more time, all tests would be remade.

Finally, another aspect to consider is whether creating a random ID behind each endpoint (see 5.2.7) to hinder cache yields false results in the sense that a real user would want the data cached for faster response times. The reason for adding the random ID was to make sure all measurements were made under equal terms, but this also means the test has been modified to some extent. On the other hand, if the result was grabbed from cache, how would query speeds even be tested? Would the query be made, or would cache take priority? Since it's not known exactly how the caching works, using a random ID to consistently create new queries could be considered beneficial as the study is about fetch times from a database, rather than fetch times from cache.

7.2.1 Related research

Some papers offer interesting insight and complement the results examined in this study. Firstly, in a study conducted by Laksono (2018), the author examines and compares the performance of *MongoDB* and PostGIS when handling geospatial data. The study focuses on the suitability of traditional SQL databases in managing big geospatial data and explores whether NoSQL databases present a viable alternative. While this study served as a significant source of inspiration, it should be noted that the amount of data used in this thesis experiment was substantially reduced compared to Laksnnonno's (2018) study. The findings of the current study contradict the conclusion reached by Laksono (2018), as it demonstrates that *MongoDB* exhibits better response times. On the other hand, Gyorodi et al. (2015) express a more optimistic viewpoint, suggesting that traditional SQL databases can outperform NoSQL databases in certain scenarios, particularly when working with smaller, structured datasets. They also emphasize that the continued utilization of *MySQL* is justified due to its feature sets, such as data integrity and scalability. It is important to highlight that although the paper directly compared *MySQL* and *MongoDB*, it did not involve geospatial queries. While the study concluded that *MongoDB* was faster, the opposite was observed when employing geospatial data with a smaller dataset, suggesting that *MySQL* is indeed faster with a limited amount of data.

Secondly, in a study conducted by Baralis et al. (2017), multiple databases, including both traditional and NoSQL databases, were compared to identify their respective strengths and weaknesses. Similar to the findings of Gyorodi et al. (2015), the study revealed that NoSQL databases exhibited better performance, while traditional SQL databases demonstrated superior scalability. The impact of the database size on query speeds for specific data was also examined, with the study determining that it did not yield any significant difference. However, it should be noted that the study in this thesis queries the entire database rather than only a portion, resulting in increased time as the amount of data grows. Additionally, the effect of adding additional data (i.e., overhead) on the retrieval time when only a subset of the data was queried was not explored. Although this aspect could have been worth investigating, it was excluded from the study as it would not provide relevant information to address the research questions posed.

Lastly, Sharma et al. (2018) compared three databases, namely *PostgreSQL*, *MongoDB*, and *Neo4j*. In their study, the author also concluded that *MongoDB* outperformed the other traditional databases overall, particularly when a large amount of data was queried. Considering this and the after-mentioned studies, there seems to be sufficient evidence that the study conducted in this thesis indeed correctly identifies the speed of the different databases and that they even may be expected as only a small amount of data was used. Had a larger dataset been used, the opposite result may have been examined where *MySQL* comes out slower than *MongoDB*. In hindsight, the amount of data is rather important when conducting tests on databases and should have been considered at the start.

7.2.2 Ethical Aspects

Due to the nature of this study, many ethical aspects can be ignored as they already are resolved. For example, there are no humans involved in the experiment, and thus informed consent and privacy protection are not a concern. Furthermore, because all data is generated, there is no chance that the data may be bound to any individuals such as coordinates leading to street addresses or specific points of interest which may have been present in real data.

Some aspects which were taken into consideration include the data generation process and transparency of how the study was conducted. For example, there are no hidden details, or configurations made which was not mentioned in this thesis. Furthermore, all code and hardware used are publicly available, meaning that the integrity of the results could be verified by another user. This same principle applies to the underlying data used, as the data generation tool is part of the codebase, including the exact queries used which can be found in the README file.

Finally, the study tried to make the comparison itself as fair as possible by avoiding modifying data afterward or taking any other action which may change the outcome of the experiment. This study does not inherit any gains, such as money or publicity by falsifying the results, and it would be unethical to do so. This is why all charts being compared have the same height, for example, to not make one seem "bigger" than the other, yielding a misleading perspective of the results.

7.2.3 Social benefit

The primary objective of conducting this study was to address the scarcity of comparative research focusing specifically on geospatial data in *MySQL* and *MongoDB*. Consequently, the study holds significant value as it contributes new empirical data regarding the performance of these two database systems. Particularly intriguing is the fact that the results directly contradict findings from previous studies, which assert that non-relational databases outperform relational databases. In this case, the results demonstrate that such claims do not hold for *MySQL*, providing valuable insights for decision-making when selecting between *MySQL* and *MongoDB* in similar scenarios. Furthermore, the study offers interesting observations regarding the impact of data volume within a datatype on query speed. By exploring this relationship, individuals are empowered to make informed adjustments and optimizations based on their specific requirements. These insights can be highly valuable for users who need to choose between *MySQL* and *MongoDB*, considering their specific circumstances involving geospatial data.

Another aspect of note is technology adoption. The comparative analysis of *MySQL* and *MongoDB* with geospatial data provides valuable insights for technology adoption within society. Organizations and developers can make informed choices about which database system to adopt based on their specific requirements, workload characteristics, and performance expectations. This knowledge can drive the adoption of appropriate technologies, leading to better system architectures, more efficient data management, and improved overall technological advancements in geospatial applications.

7.3 Future Work

The primary aspect which can be expanded upon in a future study would be the amount of data used and the number of data points created. It took a very long time to query the data, and due to time constraints, no more than 1000 data points could be created for each datatype, which contained only a small amount of data. This may not be sufficient to get concrete results, especially since a real-world database typically is a lot bigger (Zhou et al., 2009). As mentioned in the pilot study section (see 5.3.2), *MongoDB* may perform better using bigger datasets, which also is supported by authors such as Laksono (2018). Although the data amount was increased in the final study, it may not be enough to be considered "big data", and thus the full potential of *MongoDB* was not reached. Future works could try with bigger datasets to see if it

makes a difference, and if *MongoDB* indeed outperforms *MySQL* as suggested by many studies.

Another area future works can expand upon is the datatype formats. In the study, coordinates were the only information stored, but real geospatial could contain more information such as names of locations and other metadata related to area or position. This could yield more realistic data as it would be more like platforms such as Google Maps which uses metadata extensively. Furthermore, the queries themselves could use the metadata to get a position, which would further imitate that of a real-world service. For example, instead of simply getting all data from the entire database, only some items could be fetched using a search function on a keyword to get results. More utility would be achieved by performing such a study, as many actors would want that kind of functionality.

Lastly, more research on exactly how the number of points affects the response time could serve as important information. Although this study does conclude that the number of points will increase the query time, there is still uncertainty if the growth is exponential or linear. This could be found by creating multiple tests with different points and then comparing them, for example, 10, 20, 30, etc. The data used in this thesis is too uneven to draw any conclusions, and thus the question “how does query times grow when using different numbers of points” is not possible to answer. This is an interesting question as knowing growth means one can adjust to it and make calculations using a formula to get response times, rather than creating new measurements all the time. It’s also interesting to see if differs between datatypes, and if that can explain why the MultiPolygon has a higher response time than the MultiLineString.

References

Baralis, E., Dalla Valle, A., Garza, P., Rossi, C., & Scullino, F. (2017). SQL versus NoSQL databases for geospatial applications. *2017 IEEE International Conference on Big Data (Big Data)*, 3388–3397. https://doi.org/10.1109/BigData.2017.8258324

Basili, V. R., Selby, R. W., & Hutchens, D. H. (1986). Experimentation in Software Engineering. *IEEE Transactions on Software Engineering*, *SE-12*(7), 733–743. https://doi.org/10.1109/TSE.1986.6312975

Bhogal, J., & Choksi, I. (2015). Handling Big Data Using NoSQL. *2015 IEEE 29th International Conference on Advanced Information Networking and Applications Workshops*, 393–398. https://doi.org/10.1109/WAINA.2015.19

Butler, H., Daly, M., Doyle, A., Gillies, S., Hagen, S., & Schaub, T. (2016, August). *The GeoJSON Format*. https://doi.org/10.17487/RFC7946

Filip, P., & Cegan, L. (2020). Comparison of MySQL and MongoDB with focus on performance. *Proceedings - 2nd International Conference on Informatics, Multimedia, Cyber, and Information System, ICIMCIS 2020*, 184–187. https://doi.org/10.1109/ICIMCIS51567.2020.9354307

Gao, F., Yue, P., Wu, Z., & Zhang, M. (2017). Geospatial data storage based on HBase and MapReduce. *2017 6th International Conference on Agro-Geoinformatics, Agro-Geoinformatics 2017*. https://doi.org/10.1109/AGRO-GEOINFORMATICS.2017.8047040

Gyorodi, C., Gyorodi, R., Pecherle, G., & Olah, A. (2015). A comparative study: MongoDB vs. MySQL. *2015 13th International Conference on Engineering of Modern Electric Systems, EMES 2015*. https://doi.org/10.1109/EMES.2015.7158433

Khalilizangelani, Y., & Ghaffarian, S. (n.d.). *A Study of Geospatial Data Processing Based on Cloud Computing*. https://doi.org/10.13140/2.1.4307.9680

Laksono, D. (2018). Testing Spatial Data Deliverance in SQL and NoSQL Database Using NodeJS Fullstack Web App. *2018 4th International Conference on Science and Technology (ICST)*, 1–5. https://doi.org/10.1109/ICSTC.2018.8528705

Li, Y., & Manoharan, S. (2013). A performance comparison of SQL and NoSQL databases. *IEEE Pacific RIM Conference on Communications, Computers, and Signal Processing - Proceedings*, 15–19. https://doi.org/10.1109/PACRIM.2013.6625441

Müller, M., & Pfahl, D. (2008). Simulation methods. *Guide to Advanced Empirical Software Engineering*, 117–152. https://doi.org/10.1007/978-1-84800-044-5_5/COVER

MySQL :: MySQL 8.0 Reference Manual :: 11.4.1 Spatial Data Types. (2023). https://dev.mysql.com/doc/refman/8.0/en/spatial-type-overview.html

MySQL :: MySQL 8.0 Reference Manual :: 11.4.2 The OpenGIS Geometry Model. (2023). https://dev.mysql.com/doc/refman/8.0/en/opengis-geometry-model.html

Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, *14*(2), 131–164. https://doi.org/10.1007/S10664-008-9102-8/TABLES/27

Schmid, S., Galicz, E., & Reinhardt, W. (2015). WMS performance of selected SQL and NoSQL databases. *International Conference on Military Technologies (ICMT) 2015*, 1–6. https://doi.org/10.1109/MILTECHS.2015.7153736

Sharma, M., Sharma, V. D., & Bundele, M. M. (2018). Performance Analysis of RDBMS and No SQL Databases: PostgreSQL, MongoDB and Neo4j. *2018 3rd International Conference and Workshops on Recent Advances and Innovations in Engineering (ICRAIE)*, 1–5. https://doi.org/10.1109/ICRAIE.2018.8710439

Swardika, I. K., & Santiary, P. A. W. (2017). Speed of spatial query of satellite data on various database storage engine. *Proceedings - 2016 International Electronics Symposium, IES 2016*, 465–470. https://doi.org/10.1109/ELECSYM.2016.7861051

Westerveld, D. (2021). *API Testing and Development with Postman: A practical guide to creating, testing, and managing APIs for automated software testing*. Packt Publishing Ltd.

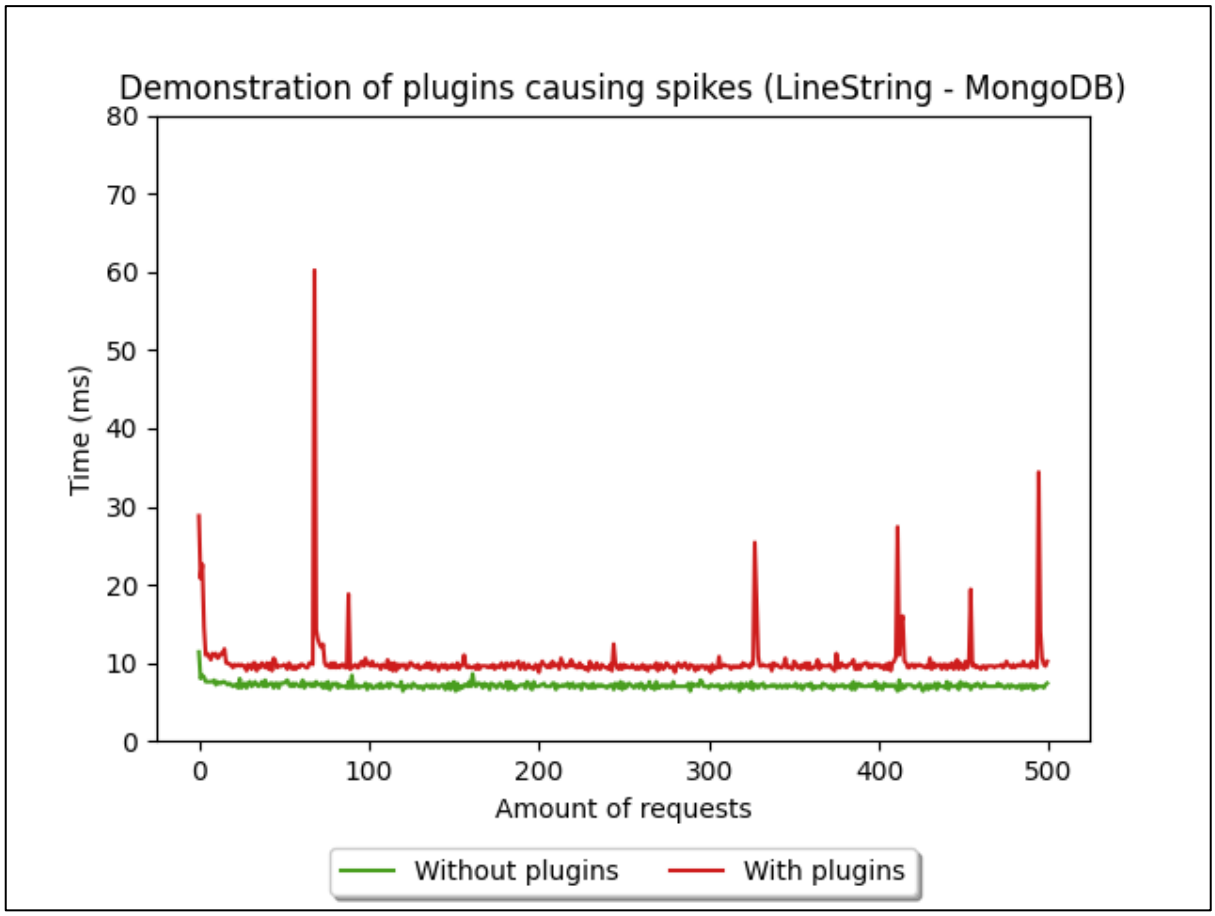
Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). Experimentation in software engineering. *Experimentation in Software Engineering*, *9783642290442*, 1–236. https://doi.org/10.1007/978-3-642-29044-2/COVER

Zhou, Z., Zhou, B., Li, W., Griglak, B., Caiseda, C., & Huang, Q. (2009). Evaluating query performance on object-relational spatial databases. *Proceedings - 2009 2nd IEEE International Conference on Computer Science and Information Technology, ICCSIT 2009*, 489–492. https://doi.org/10.1109/I

Appendix A - Pilot study data generation queries

	MySQL	MongoDB
Point	python3 src/generator/mysql/main.py --amount 100 --type point --points 20 --instances 5 --seed 200	python3 src/generator/mongodb/main.py --amount 100 --type point --points 20 --instances 5 --seed 200
LineString	python3 src/generator/mysql/main.py --amount 100 --type linestring --points 20 --instances 5 --seed 200	python3 src/generator/mongodb/main.py --amount 100 --type linestring --points 20 --instances 5 --seed 200
MultiLineString	python3 src/generator/mysql/main.py --amount 100 --type multilinestring --points 20 --instances 5 --seed 200	python3 src/generator/mongodb/main.py --amount 100 --type multilinestring --points 20 --instances 5 --seed 200

Appendix B - Final study data generation queries

	MySQL	MongoDB
Point	python3 src/generator/mysql/main.py --amount 500 --type point --points 4 --instances 5 --seed 200	python3 src/generator/mongodb/main.py --amount 500 --type point --points 4 --instances 5 --seed 200
MultiPoint	python3 src/generator/mysql/main.py --amount 500 --type multipoint --points 4 --instances 5 --seed 200	python3 src/generator/mongodb/main.py --amount 500 --type multipoint --points 4 --instances 5 --seed 200
LineString	python3 src/generator/mysql/main.py --amount 500 --type linestring --points 4 --instances 5 --seed 200	python3 src/generator/mongodb/main.py --amount 500 --type linestring --points 4 --instances 5 --seed 200
MultiLineString	python3 src/generator/mysql/main.py --amount 500 --type multilinestring --points 4 --instances 5 --seed 200	python3 src/generator/mongodb/main.py --amount 500 --type multilinestring --points 4 --instances 5 --seed 200
Polygon	python3 src/generator/mysql/main.py --amount 500 --type polygon --points 4 --instances 5 --seed 200	python3 src/generator/mongodb/main.py --amount 500 --type polygon --points 4 --instances 5 --seed 200
MultiPolygon	python3 src/generator/mysql/main.py --amount 500 --type multipolygon --points 4 --instances 5 --seed 200	python3 src/generator/mongodb/main.py --amount 500 --type multipolygon --points 4 --instances 5 --seed 200

Appendix C - Linechart of plugins causing spikes