ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

ΣΧΟΛΗ ΔΙΟΙΚΗΣΗΣ ΕΠΙΧΕΙΡΗΣΕΩΝ

SCHOOL OF BUSINESS

ΤΜΗΜΑ ΔΙΟΙΚΗΤΙΚΗΣ ΕΠΙΣΤΗΜΗΣ & ΤΕΧΝΟΛΟΓΙΑΣ

DEPARTMENT OF MANAGEMENT SCIENCE & TECHNOLOGY

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΙΩΑΝΝΗΣ ΠΑΠΑΔΑΤΟΣ (8190314)

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: ΔΙΟΜΗΔΗΣ ΣΠΙΝΕΛΛΗΣ

09/07/2023

ΟΙΚΟΝΟΜΙΚΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΑΘΗΝΩΝ

ATHENS UNIVERSITY
OF ECONOMICS
AND BUSINESS

ΣΧΟΛΗ
ΔΙΟΙΚΗΣΗΣ
ΕΠΙΧΕΙΡΗΣΕΩΝ
SCHOOL OF
BUSINESS

ΤΜΗΜΑ
ΔΙΟΙΚΗΤΙΚΗΣ
ΕΠΙΣΤΗΜΗΣ &
ΤΕΧΝΟΛΟΓΙΑΣ
DEPARTMENT OF
MANAGEMENT
SCIENCE &
TECHNOLOGY

ΤΙΤΛΟΣ: Radius Query Performance Benchmarking

Abstract

In the last few years, there has been a rise in Location-based services (LBS) applications that utilize geospatial data to offer personalized services based on the user's real-time location. In user-facing applications, low latency is crucial for a seamless user experience (UX). The choice of a spatial database heavily influences the performance of a location-based services (LBS) application, hence becoming a critical decision. To ensure an unbiased and informed decision, it is common practice to perform a database benchmark. In this study, we conducted a benchmark to identify the most efficient spatial database among PostgreSQL with the PostGIS extension and MongoDB in radius queries. The results indicate the performance superiority of PostGIS over MongoDB in all the evaluated scenarios. One of the key aspects of this study is that the benchmark conducted is fully reproducible to allow for independent verification of the results and promote the advancement of knowledge in the field.

Table of Contents

I. Introduction

Geospatial data (or Geodata) refers to data associated with a specific location on the Earth's surface, such as coordinates (latitude and longitude), postal codes, or other location identifiers [1]. Geospatial data are considered really important when analyzing data because they provide the additional context of where something happened [2]. In this study, we will only make use of a particular type of geospatial data, limiting our scope to points defined by latitude and longitude coordinates. Points do not have an extent; they represent a single location [3].

In the last few years, the availability of geospatial data has significantly increased. This trend will continue to grow since the use of mobile phones has been well integrated into our daily routines [5, 10, 11]. Every day we interact with a plethora of mobile applications that leverage the GPS (Global Positioning System) capabilities of mobile devices to determine our current location and provide us with personalized location-based services [6, 7]. "Location-based services (LBS) are the delivery of data and information services where the content of those services is tailored to the current or some projected location and context of a mobile user [4]".

Popular examples of such applications include Uber, matching its users with nearby drivers based on their real-time location; Airbnb, allowing its users to search for and book rentals within a specified radius from a particular location; Yelp, enabling its users to search for nearby businesses, such as restaurants, cafes, or other services, from their current location or a particular address; Foursquare, allowing its users to discover and share information about nearby points of interest, such as restaurants, bars, and landmarks, and check-in to these locations to share their experiences with friends; DoorDash, matching its users with nearby restaurants that offer delivery services, and many more.

In user-facing applications, performance, particularly low latency, plays a crucial role in shaping the user experience (UX). When a system holds and manages large amounts of data, its performance becomes increasingly reliant on the efficiency of the underlying database. Consequently, when performance issues arise, the database is frequently the primary target of scrutiny. Thus, performance is one of the most important factors when a company chooses a database for its product or service [12].

Location-based services (LBS) applications utilize spatial databases instead of typical databases. Spatial databases are an extension of the typical databases that enables them to store and retrieve geospatial information, such as Points, Lines, and Polygons. The most popular open-sourced spatial database is PostgreSQL with the PostGIS extension. Also, MongoDB is the most popular open-sourced NoSQL spatial database with many built-in spatial capabilities [1].

Therefore, the choice of a spatial database is a critical decision for a location-based services (LBS) application. To ensure an unbiased and informed decision, it is common practice to perform a database benchmark [18]. In this study, we conducted a read-only benchmark using a synthetic dataset, to compare the performance of PostgreSQL with the PostGIS extension and MongoDB in the radius query. The radius query, also known as proximity query, is a spatial query that retrieves all data objects that fall within a specified radius distance from a center point, defined by its latitude and longitude coordinates.

The main contributions of this paper are the identification of the most efficient spatial database for radius queries with datasets exhibiting similar characteristics to those used in this study, and the development of a fully reproducible benchmark that promotes the advancement of knowledge in the field.

The paper is organized as follows: Section II provides details about the related work, Section III describes the datasets used to conduct the benchmark, Section IV describes the data models employed in the benchmark, Section V outlines the queries employed in the benchmark, Section VI describes the setup employed to conduct the benchmark, and Section VII analyzes the experimental results.

II. Related Work

Geospatial data is becoming increasingly important with the rise of location-based services (LBS) applications and choosing the right spatial database can have a significant impact on the application's performance. This is also reflected in the increasing number of studies that are comparing the efficiency of SQL and NoSQL databases for processing geospatial data.

For instance, the authors of [5] evaluated the efficiency of PostGIS and MongoDB in the processing of geospatial data. By executing each query 10 times and measuring the average response time, the authors found that MongoDB outperformed PostGIS for point containment, intersection, and radius query for radius distances of 100km, 200km, and 500km. However, for a radius of 1000km, MongoDB proved to be 3 times slower than PostGIS. Some limitations of this study are the usage of a very large radius compared to what is normally used in most location-based services (LBS) applications. Additionally, the authors do not mention anything about a warmup phase. Without a warmup, benchmark queries may suffer from cold start performance penalties, as the database systems need to load data into memory and adapt their query optimization strategies [14].

In contrast, the authors of [6] focused on spatio-temporal queries and compared the performance of PostGIS and MongoDB. By executing each query 5 times and measuring the average response time, they discovered that PostGIS outperformed MongoDB in all queries both for a single node and 5-node cluster setup. Specifically, for the Q5 query that belongs to the radius-based query category, PostGIS is approximately 1.5 times faster using the ST_Buffer function instead of ST_DWithin.

Another study investigated the efficiency of bounding box queries, the authors of [7] specifically assessed Azure SQL Database and Azure DocumentDB by load testing two web services. The authors found that the average response time of Azure DocumentDB is on average 1.5 times faster than Azure SQL Database, however, it can manage fewer concurrent requests for the same price range.

Furthermore, it is worth noting that the lack of reproducibility is a common issue within our field. None of the studies reviewed in this literature review provide the necessary scripts or even the queries required to reproduce their benchmark experiments accurately. This lack of reproducibility hinders the transparency and reliability of the reported results, making it challenging for researchers and practitioners to validate and build upon the findings. Reproducibility is a crucial aspect of scientific research as it allows for independent verification and promotes the advancement of knowledge in the field [13, 14, 16].

Building upon the existing body of research, my work aims to create a fully reproducible benchmark for PostgreSQL with the PostGIS extension and MongoDB spatial databases using the same methodology as [7, 8]. This methodology involves conducting load tests on API endpoints that utilize the benchmark queries. The reason behind my choice to follow this methodology is that there is no widely used spatial benchmark [9] or single benchmarking tool that can handle both of these spatial databases. Using different benchmarking tools might impact the observed performance differences since they may introduce non-constant variations in how the data is fetched and how the queries are executed. However, it is important to note that the absolute performance difference between the systems cannot be accurately computed using this methodology since the spatial databases are not the only factor affecting the response time.

III. Datasets

In order to evaluate the performance of radius queries in a location-based services (LBS) application context, the use of a representative dataset was essential [19]. The dataset was carefully constructed using two primary sources: the businesses dataset provided by Yelp and the MyFitnessPal dataset. The original datasets are publicly available at https://zenodo.org/record/8074982. Additionally, the Jupyter Notebooks used to construct the synthetic datasets, which were then used to populate the spatial databases for the benchmark can be found at https://github.com/Radius-Query-Performance-Benchmarking/Dataset. We utilized these source datasets, to construct a Stores and a Products dataset. Our aim was to simulate a realistic location-based services (LBS) application, where users would be able to find products near their location while considering their nutritional values.

The Yelp businesses dataset served as the foundation for constructing the Stores dataset, by using the locations of the businesses as the locations of the Stores. The Stores dataset consists of approximately 145,000 records. By leveraging the Yelp dataset, we ensure that the store locations within our dataset are distributed in a manner that aligns with real-world location-based services (LBS) applications.

The MyFitnessPal dataset served as the foundation for constructing the Products dataset, by using the nutrients of the food entries as the nutrients of the Products. The Products dataset consists of approximately 1.850.000 records. By leveraging the MyFitnessPal dataset, the products in our dataset will mirror the nutritional composition of real food items. Thus, the relative distributions of the nutrients in our products will be accurate, which in turn will make the results of the benchmark with respect to filtering more reliable. The products were assigned to the stores in a round-robin fashion, resulting in each store having about 13 products.

Finally, we created two additional datasets, one containing 50.000 Stores along with their corresponding products, and another one containing 100.000 Stores along with their corresponding products. Both datasets are subsets of the main dataset. More details about the dataset creation, such as the preprocessing steps can be found in the corresponding Jupyter Notebook.

IV. Data Models

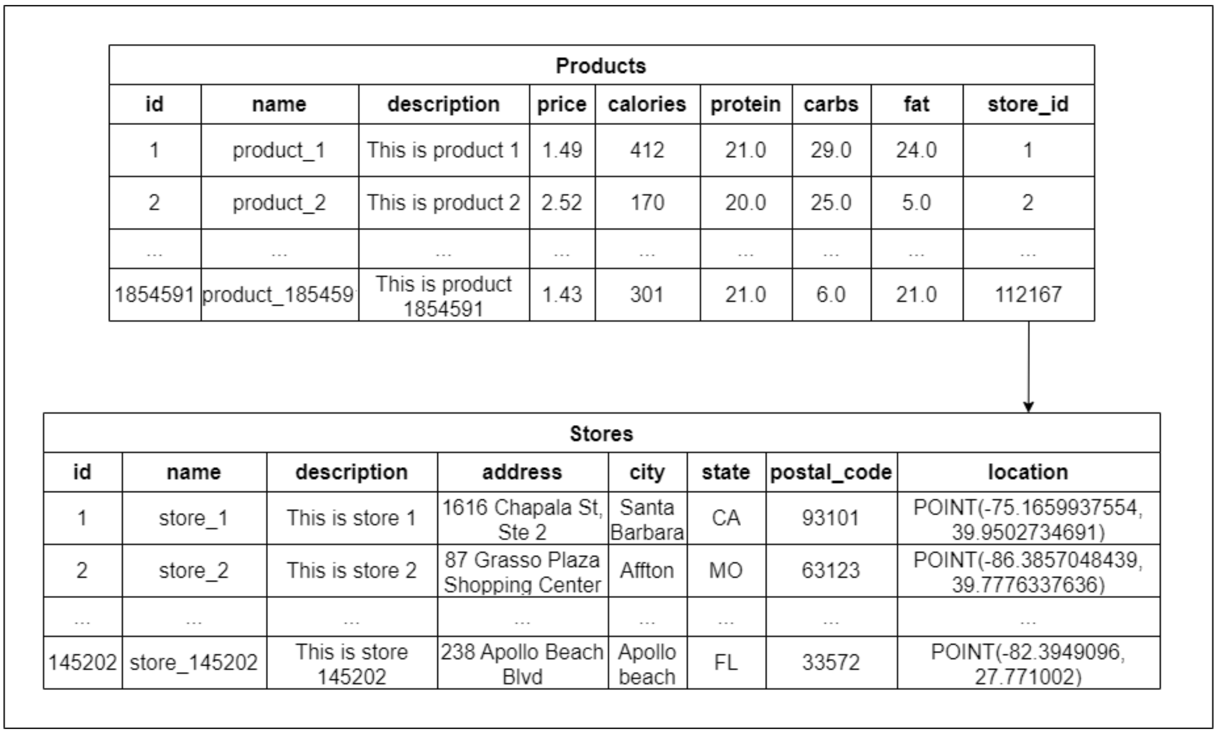
In this section, we will discuss the data models employed in the Radius Query Performance Benchmarking. The choice of the data model can significantly impact the performance, especially for the document data model [12, 20].

The relational data model, employed in PostgreSQL with the PostGIS extension, was constructed based on the principles of the third normal form (3NF) to ensure data integrity and minimize redundancy. The model consists of two tables: Stores and Products.

The Stores table is designed to store information about the different stores. Each store is uniquely identified by an "id" column, serving as the primary key. The "name" column holds the generic name of the store. The "description" column provides a generic description of the store. The address of the store is stored in the "address" column, while the "city" and "state" columns hold the corresponding city and state information. The "postal_code" column stores the postal code of the store's location. Additionally, the "location" column stores the precise geographical coordinates of each store. Finally, to optimize spatial queries involving store locations, an index is created on the "location" field using the GIST index type.

The Products table is designed to store information about the various products assigned to each store. Each product is uniquely identified by an "id" column, serving as the primary key. The "name" column holds the generic name of the product. The "description" column provides a generic description of the product. The "price" column stores the price of the product, while the "calories", "protein", "carbs", and "fat" columns hold nutritional information about the product. Additionally, to establish a relationship between products and stores, the "store_id" column is included, referencing the "id" column of the Stores table as a foreign key. Finally, to optimize queries involving retrieving products by store, an index is created on the "store_id" field.

The tables contain columns that will not be used in any of the queries described in the next section, to increase the size of each row. The reason is that the size of each row in a table directly affects how many rows can fit within a single page. For example, by including the "description" column in the table, we increase the size of each row, which, in turn, affects the number of rows that can fit on a single page. Reducing the number of pages required to store the data can impact the performance. When performing operations like querying or updating data, the database system often needs to read or write entire pages. If a larger number of rows can fit within a single page, it means fewer pages need to be read or written to perform operations on a given amount of data. This can result in reduced disk I/O operations and improved performance [21].



Products

id	name	description	price	calories	protein	carbs	fat	store_id
1	product_1	This is product 1	1.49	412	21.0	29.0	24.0	1
2	product_2	This is product 2	2.52	170	20.0	25.0	5.0	2
...	...	...	...	...	...	...	...	...
1854591	product_185459	This is product 1854591	1.43	301	21.0	6.0	21.0	112167

Stores

id	name	description	address	city	state	postal_code	location
1	store_1	This is store 1	1616 Chapala St, Ste 2	Santa Barbara	CA	93101	POINT(-75.1659937554, 39.9502734691)
2	store_2	This is store 2	87 Grasso Plaza Shopping Center	Affton	MO	63123	POINT(-86.3857048439, 39.7776337636)
...	...	...	...	...	...	...	...
145202	store_145202	This is store 145202	238 Apollo Beach Blvd	Apollo beach	FL	33572	POINT(-82.3949096, 27.771002)

Figure 1: Relational Data Model

The document data model, employed in MongoDB, required careful consideration due to the flexibility and schemaless nature of document-oriented databases. Several options were evaluated, taking into account the broader range of operations that a typical location-based services (LBS) application needs to perform. Our focus was not solely on optimizing the data model for the queries of the benchmark. This enhances the usefulness of our analysis, by making our findings more relevant to real-world location-based services (LBS) applications.

One option was to fully embed the products within each store document. However, this option was deemed suboptimal for two primary reasons. First, updating individual products within embedded documents would result in inefficiency, as modifying a single product would require updating the entire store document. Second, in real-world applications, individual product queries, such as displaying additional details in a detailed view, would necessitate the retrieval of products individually, which would be cumbersome and inefficient.

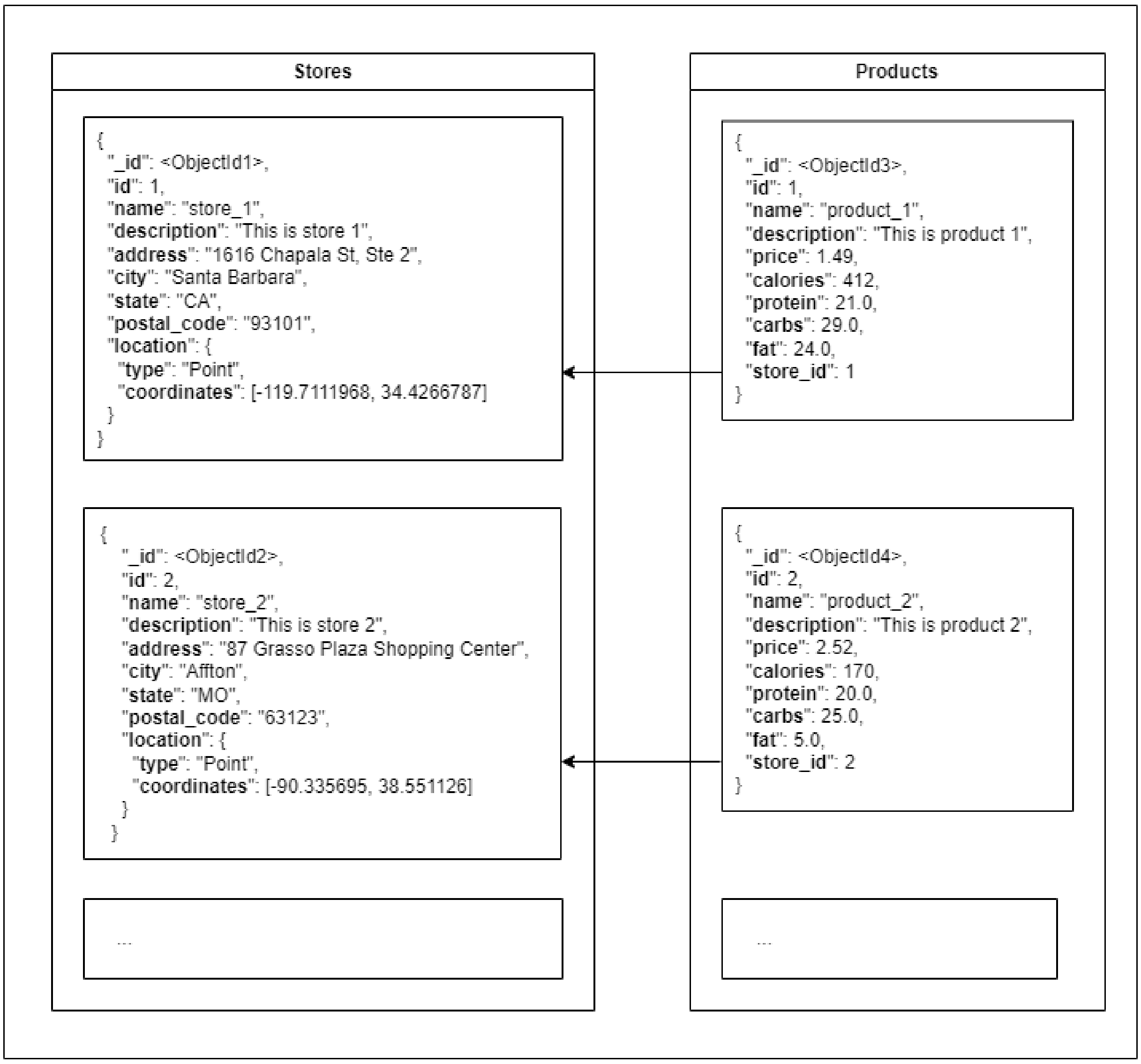
Another option involved partial denormalization by storing an array of product ids within each store document. While this option does not face the challenges of the previous option, it is not optimal, since searching for products using their ids would require multiple index scans, as each product would need to be individually located and retrieved.

To optimize the document data model, a fully normalized approach was adopted, resembling the relational data model to a certain extent. Each product document contained a “store_id” field as a reference to a document of the Stores collection. Moreover, an index was created on the “store_id” field within the product documents to enhance query performance. This approach enabled efficient retrieval of a store's products using a single index scan, as the products associated with a particular store would be stored adjacent to each other in the index due to sorting.

The Stores collection in MongoDB is designed to store information about the different stores. Each store is uniquely identified by the "id" field, serving as a primary key. The "name" field holds the generic name of the store. The "description" field provides a brief description of the store. The address of the store is stored in the "address" field, while the "city" and "state" fields hold the corresponding city and state information. The "postal_code" field stores the postal code of the store's location. Additionally, the "location" field is structured as a GeoJSON object, specifying the type as "Point" and containing the coordinates array to store the precise geographical coordinates of each store. Finally, in order to optimize spatial queries involving store locations, an index is created on the "location" field using the "2dsphere" index type.

The "Products" collection in MongoDB is designed to store information about the various products assigned to each store. Each product is uniquely identified by the "id" field, serving as the primary key. The "name" field holds the generic name of the product, providing a recognizable identifier. The "description" field offers a brief description of the product, providing additional information about its features or properties. The "price" field stores the price of the product, while the "calories", "protein", "carbs", and "fat" fields hold nutritional information for the product. Additionally, to establish a relationship between products and stores, the "store_id" field is included, referencing the "id" field of the "Stores" collection similar to a foreign key. Finally, to optimize queries involving retrieving products by store, an index is created on the "store_id" field.

In MongoDB, each document has a unique identifier field called “_id” that serves as the primary key. In order to simplify the process of generating the JSON arrays used to load the data into MongoDB, we decided to use an auto-increment indexed field called "id" instead of the automatically generated "_id" field as our primary key. However, we anticipate our findings to be the same as if we were using the “_id” field as our primary key since this decision should not entail any performance penalty for reads.



```
{
  "_id": <ObjectId1>,
  "id": 1,
  "name": "store_1",
  "description": "This is store 1",
  "address": "1616 Chapala St, Ste 2",
  "city": "Santa Barbara",
  "state": "CA",
  "postal_code": "93101",
  "location": {
    "type": "Point",
    "coordinates": [-119.7111968, 34.4266787]
  }
}
```

```
{
    "_id": <ObjectId2>,
    "id": 2,
    "name": "store_2",
    "description": "This is store 2",
    "address": "87 Grasso Plaza Shopping Center",
    "city": "Affton",
    "state": "MO",
    "postal_code": "63123",
    "location": {
      "type": "Point",
      "coordinates": [-90.335695, 38.551126]
    }
  }
```

```
...
```

```
{
  "_id": <ObjectId3>,
  "id": 1,
  "name": "product_1",
  "description": "This is product 1",
  "price": 1.49,
  "calories": 412,
  "protein": 21.0,
  "carbs": 29.0,
  "fat": 24.0,
  "store_id": 1
}
```

```
{
  "_id": <ObjectId4>,
  "id": 2,
  "name": "product_2",
  "description": "This is product 2",
  "price": 2.52,
  "calories": 170,
  "protein": 20.0,
  "carbs": 25.0,
  "fat": 5.0,
  "store_id": 2
}
```

```
...
```

Figure 2: Document Data Model

V. Queries

To test the performance of each spatial database on the radius query we will use the following two queries:

I. Find all stores in proximity up to the specified maximum distance from the given position (longitude, latitude)

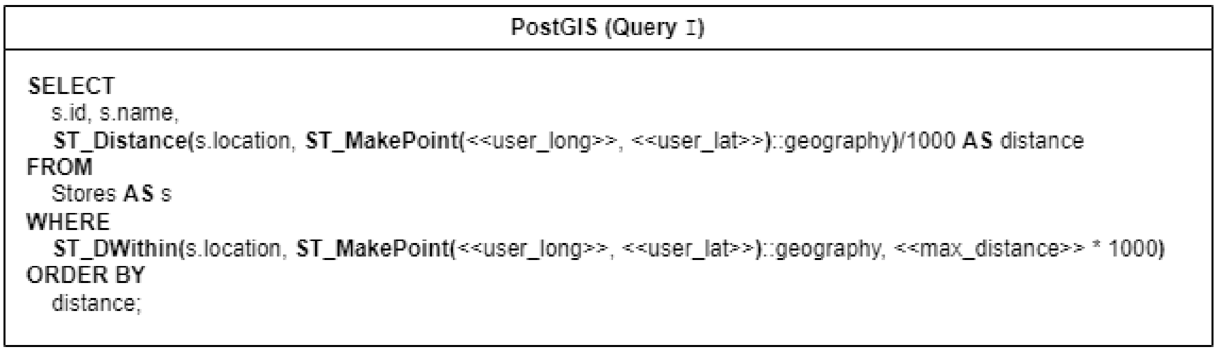
II. Find all products belonging to stores in proximity up to the specified maximum distance from the given position (longitude, latitude), that fall within the specified price and calorie ranges

The user experience (UX) can play a crucial role in shaping the workload of an application. In the context of query II, which involves finding all products belonging to stores within a specified proximity from a given position, that falls within the specified price and calorie ranges, it is common to break down the query into two distinct parts: the radius query (query I) and the filtering process. The radius query determines the set of stores within the specified maximum distance, while the subsequent filtering step narrows down the results based on price and calorie ranges.

Dividing a long operation into two smaller ones inherently improves the UX by providing more granular feedback and a sense of progress. Moreover, in the context of the specific location-based services (LBS) application we are considering, by breaking down the query users would be able to perform multiple searches from the same location without incurring the additional overhead of executing the radius query each time.

It is important to note that query II was included in the benchmark to account for scenarios where this type of query might be relevant, even though it involves a heavier operation due to the join and filtering. This inclusion enables a comprehensive evaluation of performance for different use cases. Additionally, to the best of our knowledge, similar studies that are not specifically focused on radius queries have not considered something similar.

The provided PostGIS query represents query I, of the Radius Query Performance Benchmarking, which involves performing a radius query to find stores within a specified proximity from a given position.


PostGIS (Query I)

```
SELECT
  s.id, s.name,
  ST_Distance(s.location, ST_MakePoint(<<user_long>>, <<user_lat>>)::geography)/1000 AS distance
FROM
  Stores AS s
WHERE
  ST_DWithin(s.location, ST_MakePoint(<<user_long>>, <<user_lat>>)::geography, <<max_distance>> * 1000)
ORDER BY
  distance;
```

Figure 3: PostGIS Query I

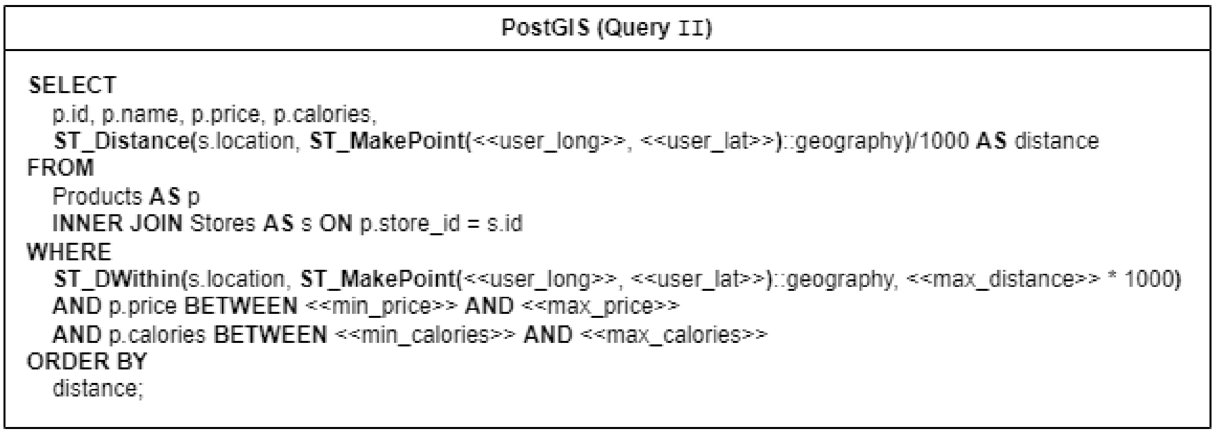
The SELECT statement retrieves the store id, store name, and the calculated distance from the store's location to the user's specified position. The distance is calculated using the ST_Distance function, which measures the distance in meters between the store's location and the user's specified position. The distance is divided by 1000 to convert it from meters to kilometers.

The FROM clause specifies the Stores table, indicating that the store data will be retrieved from this table.

The WHERE clause filters the stores based on their proximity to the user's position. It utilizes the ST_DWithin function, which checks if the distance between the store's location and the user's specified position is within a specified maximum distance. This condition ensures that only stores within the desired proximity are included in the result set. It is worth noting that the ST_DWithin function can take advantage of the GIST (Generalized Search Tree) index on the “location” column, which enhances the performance of spatial queries.

The ORDER BY clause arranges the results in ascending order based on the calculated distance. This allows the stores to be presented to the user in increasing order of their proximity to the specified position.

The provided PostGIS query represents query II, of the Radius Query Performance Benchmarking, which involves finding all products belonging to stores within a specified proximity, falling within specified price and calorie ranges.



PostGIS (Query II)

```
SELECT
   p.id, p.name, p.price, p.calories,
   ST_Distance(s.location, ST_MakePoint(<<user_long>>, <<user_lat>>)::geography)/1000 AS distance
FROM
   Products AS p
   INNER JOIN Stores AS s ON p.store_id = s.id
WHERE
   ST_DWithin(s.location, ST_MakePoint(<<user_long>>, <<user_lat>>)::geography, <<max_distance>> * 1000)
   AND p.price BETWEEN <<min_price>> AND <<max_price>>
   AND p.calories BETWEEN <<min_calories>> AND <<max_calories>>
ORDER BY
   distance;
```

Figure 4: PostGIS Query II

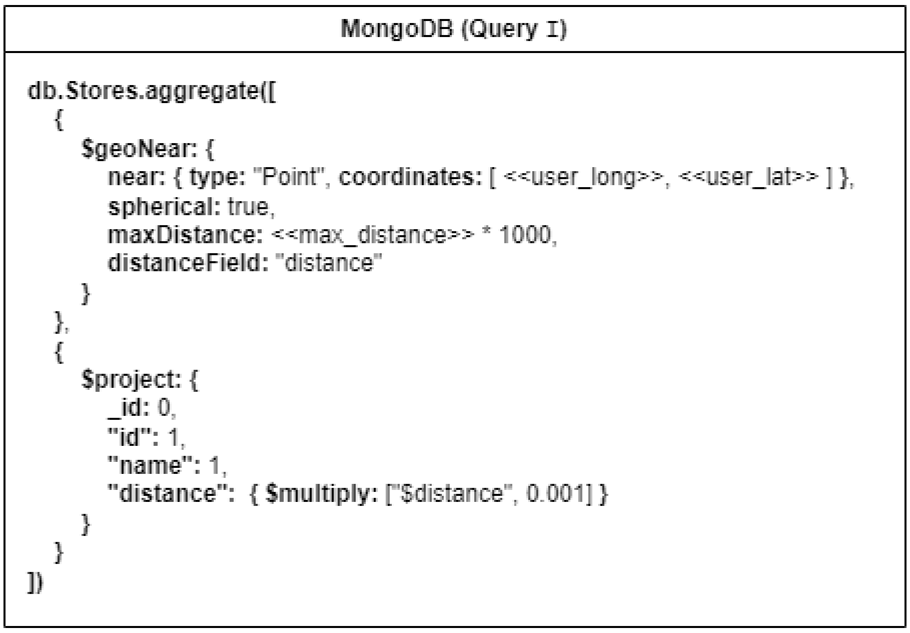
The SELECT statement retrieves the product id, product name, price, calories, and the calculated distance from the store's location to the user's specified position. The distance is calculated using the ST_Distance function, which measures the distance in meters between the store's location and the user's specified position. The distance is divided by 1000 to convert it from meters to kilometers.

The FROM clause specifies two tables: Products and Stores. These tables are joined using the INNER JOIN operation, where the value of the “store_id” column in the Products table matches the value of the “id” column in the Stores table. This ensures that only products from stores within the specified proximity are included in the result set. It is worth noting that the index on the “store_id” column of the Products table makes the join operation with the Stores table more efficient. This index enables faster lookup and retrieval of the associated store information during the join process.

The WHERE clause further filters the products based on multiple conditions. It utilizes the ST_DWithin function, which checks if the distance between the store's location and the user's specified position is within a specified maximum distance. This condition ensures that only products from stores within the desired proximity are considered. It is worth noting that the ST_DWithin function can take advantage of the GIST (Generalized Search Tree) index on the “location” column, which enhances the performance of spatial queries. Moreover, it ensures that only products with prices and amounts of calories that fall within the specified price and calorie ranges are included in the result set.

The ORDER BY clause arranges the results in ascending order based on the calculated distance. This ensures that the products are presented to the user in increasing order of their proximity to the specified position.

The provided MongoDB query represents query I, of the Radius Query Performance Benchmarking, which involves performing a radius query to find stores within a specified proximity from a given position. The query utilizes the MongoDB Aggregation Pipeline and consists of the $geoNear and $project stages described below.



MongoDB (Query I)

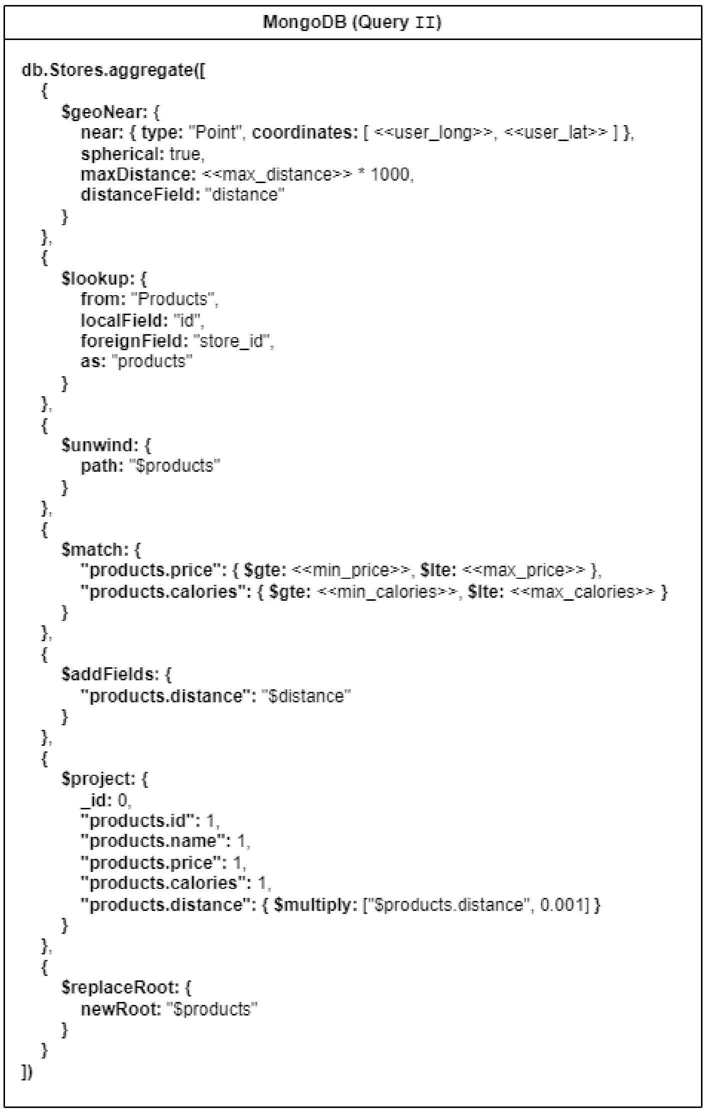
```
db.Stores.aggregate([
   {
      $geoNear: {
         near: { type: "Point", coordinates: [ <<user_long>>, <<user_lat>> ] },
         spherical: true,
         maxDistance: <<max_distance>> * 1000,
         distanceField: "distance"
      }
   },
   {
      $project: {
         _id: 0,
         "id": 1,
         "name": 1,
         "distance":  { $multiply: ["$distance", 0.001] }
      }
   }
])
```

Figure 5: MongoDB Query I

The $geoNear stage performs the geospatial search based on the provided location information. This stage takes advantage of MongoDB's geospatial capabilities and can optimize the query execution using the 2dsphere index on the “location” field. The 2dsphere index is a geospatial index specifically designed for performing geospatial calculations on a spherical surface, such as the Earth's surface. By utilizing this index, the $geoNear stage can efficiently search for documents near a specified point, taking into account the spherical nature of the Earth.

The $project stage is used to reshape the documents in the pipeline, including or excluding specific fields. In this case, it selects the "id", "name", and "distance" fields to be included in the output. The "distance" field is calculated by multiplying the "distance" value, obtained from the $geoNear stage, by 0.001 to convert it from meters to kilometers.

The provided MongoDB query represents query II, of the Radius Query Performance Benchmarking, which involves finding all products belonging to stores within a specified proximity, falling within specified price and calorie ranges. The query utilizes the MongoDB Aggregation Pipeline and consists of the $geoNear, $lookup, $unwind, $match, $addFields, $project, and $replaceRoot stages described below.



MongoDB (Query II)

```
db.Stores.aggregate([
  {
    $geoNear: {
      near: { type: "Point", coordinates: [ <<user_long>>, <<user_lat>> ] },
      spherical: true,
      maxDistance: <<max_distance>> * 1000,
      distanceField: "distance"
    }
  },
  {
    $lookup: {
      from: "Products",
      localField: "id",
      foreignField: "store_id",
      as: "products"
    }
  },
  {
    $unwind: {
      path: "$products"
    }
  },
  {
    $match: {
      "products.price": { $gte: <<min_price>>, $lte: <<max_price>> },
      "products.calories": { $gte: <<min_calories>>, $lte: <<max_calories>> }
    }
  },
  {
    $addFields: {
      "products.distance": "$distance"
    }
  },
  {
    $project: {
      _id: 0,
      "products.id": 1,
      "products.name": 1,
      "products.price": 1,
      "products.calories": 1,
      "products.distance": { $multiply: ["$products.distance", 0.001] }
    }
  },
  {
    $replaceRoot: {
      newRoot: "$products"
    }
  }
])
```

Figure 6: MongoDB Query II

The $geoNear stage performs the geospatial search based on the provided location information. Similar to query I, the $geoNear stage takes advantage of MongoDB's geospatial capabilities and can optimize the query execution using the 2dsphere index on the “location” field.

The $lookup stage performs a join operation between the Stores and Products collections. It links the documents from both collections by matching the "id" field in the Stores collection with the "store_id" field in the Products collection. The lookup stage can take advantage of the index on the “store_id” field on the Products collection, further optimizing the query execution by efficiently matching the documents.

The $unwind stage deconstructs the resulting "products" array from the $lookup stage. This allows for subsequent filtering and processing of individual product documents.

The $match stage filters the documents based on the specified price and calorie ranges. It ensures that only products with prices and amounts of calories falling within the specified price and calorie ranges are included in the result set.

The $addFields stage adds a new field called "distance" to each product document. The value of this field is the distance calculated from the $geoNear stage, representing the proximity between the store and the specified location.

The $project stage reshapes the documents in the pipeline, including or excluding specific fields. It selects the "id", "name", "price", "calories", and "distance" fields to be included in the output. The "distance" field is calculated by multiplying the "distance" value obtained from the $geoNear stage by 0.001 to convert it from meters to kilometers.

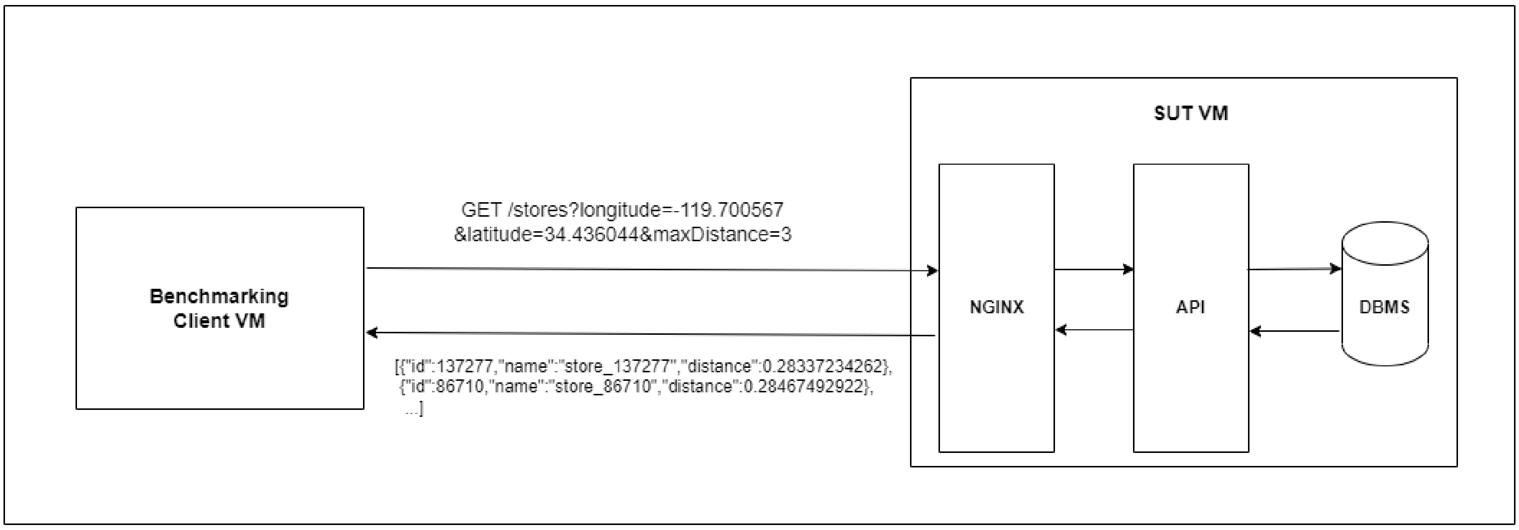
The $replaceRoot stage promotes the "products" field as the new root of the document, discarding the unnecessary parent fields. This results in a cleaner representation of the product documents in the final result set.

VI. Setup

In this section, we will provide a detailed description of the setup employed for conducting the Radius Query Performance Benchmarking. As we have already mentioned in Section II, one of the key aspects of this study is to be fully reproducible, which ensures that the experiment can be accurately replicated by other researchers or interested parties. To achieve this, we leveraged the power of Terraform and Ansible for infrastructure provisioning and configuration, Docker and Docker Compose for deployment, and k6 for load-testing.

The setup consists of two main components: the "Benchmarking Client VM" and the "SUT VM". The "SUT VM" serves as the system under test and comprises of an NGINX reverse proxy, an API server, and the selected spatial database. The "Benchmarking Client VM" is responsible for executing load-testing scripts (k6 scripts) and copying the results to the Ansible Control Node machine.

In this setup, when a load-testing script generates an HTTP request, it is received by NGINX, which is responsible for forwarding the requests to the API server. Upon receiving a request, the API server initiates an asynchronous query to the chosen spatial database for processing. Once the spatial database completes processing the request, it responds to the API server with the result set. The API server then converts this result set into a JSON array and forwards it to NGINX. Finally, NGINX returns the JSON array response to the client, completing the request-response cycle.



```
GET /stores?longitude=-119.700567
&latitude=34.436044&maxDistance=3
```

```
[{"id":137277,"name":"store_137277","distance":0.28337234262},
 {"id":86710,"name":"store_86710","distance":0.28467492922},
 ...]
```

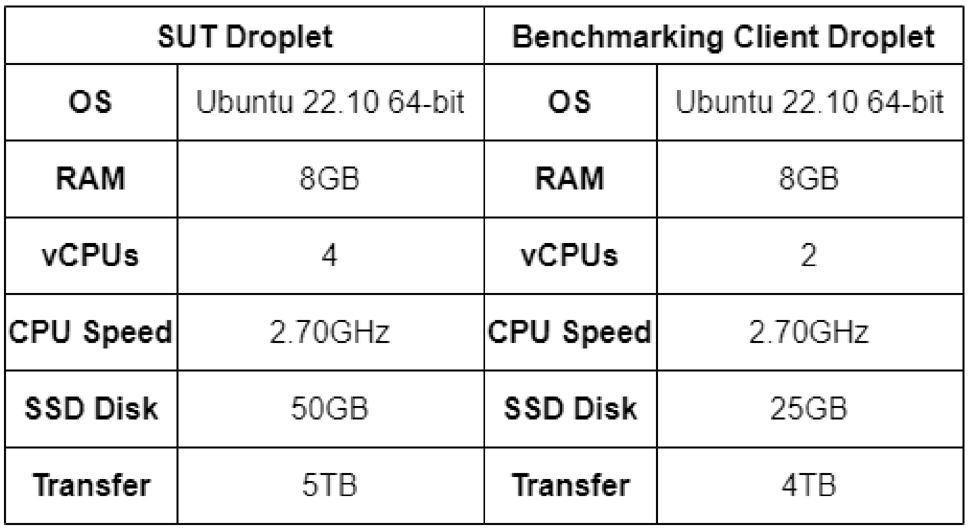
Figure 7: Setup High Level Overview

A. Infrastructure

To automate our infrastructure provisioning and configuration tasks, we used two powerful Infrastructure as Code (IaC) tools: Terraform and Ansible. While Terraform excels at creating and managing infrastructure resources, Ansible's strengths lie in configuring and deploying software applications and services on that infrastructure. The IaC scripts required to reproduce the benchmarking results can be found at https://github.com/Radius-Query-Performance-Benchmarking/IaC, along with step-by-step instructions on how to conduct the benchmark in the README.md file.

To reproduce the Radius Query Performance Benchmarking you only need a machine operating on Ubuntu 22.10 x64 to be used as the Ansible Control Node and a DigitalOcean account. We chose DigitalOcean as our cloud hosting provider because of their generous free trial and their really good documentation.

For the benchmarking environment setup, we require two DigitalOcean droplets—one for the system under test (SUT) and another for the benchmarking client. To ensure optimal performance and resource availability, we have chosen dedicated CPU droplets. These droplets guarantee that the virtual machine will have exclusive access to the allocated resources at all times. Furthermore, both droplets reside in the Frankfurt region, specifically in the same datacenter (FRA1). This choice is based on the DigitalOcean documentation, which states that resources created in the same datacenter will be members of the same VPC Network. This is crucial to minimize network latency and ensure efficient communication between the droplets.



SUT Droplet		Benchmarking Client Droplet	
OS	Ubuntu 22.10 64-bit	**OS**	Ubuntu 22.10 64-bit
RAM	8GB	**RAM**	8GB
vCPUs	4	**vCPUs**	2
CPU Speed	2.70GHz	**CPU Speed**	2.70GHz
SSD Disk	50GB	**SSD Disk**	25GB
Transfer	5TB	**Transfer**	4TB

Table 1: Droplet Specification

B. System Under Test Deployment

To automate the deployment of the NGINX reverse proxy, the APIs, and the spatial databases, we used Docker and Docker Compose due to their ability to provide a consistent and reproducible environment. This decision is supported by a study [15] that investigated whether it is safe to dockerize database benchmarks. Although the study concluded that Docker can have an impact on the results of such experiments, the effect for the configuration most similar to ours, which involved running 10 threads with the system under test (SUT) dockerized, was found to be only 1.84% on Reads. This level of impact is considered acceptable for our specific use case, as our primary objective is to compare the relative performance of PostGIS and MongoDB for the radius query. The docker images used in the Radius Query Performance Benchmarking are hosted on Docker Hub and can be found at https://hub.docker.com/u/papajohn77. Also, the Dockerfiles used to build the docker images, the source code of the APIs, and the docker-compose files used for the deployment are available at https://github.com/orgs/Radius-Query-Performance-Benchmarking/repositories.

C. API Configuration

The API configuration is equivalent for each spatial database. Both APIs are implemented using .NET Minimal APIs, utilizing asynchronous endpoints and connection pooling to minimize the API's performance effect on the results.

PostGIS	Default	MongoDB	Default
Minimum Pool Size	0	MinConnectionPoolSize	0
Maximum Pool Size	100	MaxConnectionPoolSize	100
Connection Idle Lifetime	300s	MaxConnectionIdleTime	600s
Connection Lifetime	0 (disabled)	MaxConnectionLifeTime	1800s
Timeout	15s	ConnectTimeout	30s

Table 2: Common Connection Pool Parameters

The following connection string parameters have been provided to PostGIS, to configure the common connection pool parameters between the two spatial databases with the same value. When comparing the values between the two systems, the larger value has been chosen, coincidentally matching MongoDB's default value for all three parameters:

- Connection Idle Lifetime=600
- Connection Lifetime=3600
- Timeout=30

D. Load Testing

k6 was chosen as our load-testing tool due to its adoption of a "tests as code" approach, which is important for reproducibility. With load tests expressed as code, we can easily version control and share them, replicating scenarios accurately across environments. The k6 scripts used to conduct the load tests for the benchmark can be found at https://github.com/Radius-Query-Performance-Benchmarking/k6.

We used a different load-testing dataset for each endpoint, dataset, and number of virtual users combination. The load-testing datasets are publicly available at https://zenodo.org/record/8081573. The load-testing datasets are used as parameters in the requests by the load-testing scripts to prevent server-side caching. The radius used for these queries ranged from 1 to 5 kilometers. To ensure that the queries will return results for the parameters provided by the produced load-testing datasets, we geocoded the unique city and state combinations for each of the datasets, and we used the Faker library to generate latitude-longitude pairs around the centers of the geocoded locations. The Jupyter Notebook used to produce the load-testing datasets can be found at https://github.com/Radius-Query-Performance-Benchmarking/Dataset.

For the /stores endpoint, which utilizes Query I, 5000 requests are made, while for the /products endpoint, which utilizes Query II, 2000 requests are made. To account for the significant performance difference between "cold" and "hot" runs, we considered the first 10% of requests as warmup and excluded them when measuring performance. During the initial (cold) run, the processing time is usually significantly longer as the relevant data needs to be loaded from persistent storage, and the query is parsed and compiled. Subsequent (hot) runs benefit from the data being readily available in buffers, resulting in faster execution [14].

VII. Experimental Results

Repetition is key in benchmarking to ensure accurate and reliable results. By conducting multiple runs and repeating experiments sufficiently, random variations and interference can be effectively addressed. This is particularly important when dealing with dockerization, where there is an extra layer of indirection. Overall, repetition plays a vital role in benchmarking by minimizing errors and providing more reliable data [14, 15, 16]. In the Radius Query Performance Benchmarking, for each spatial database and dataset combination, we load tested 3 times the /stores endpoint utilizing Query I and the /products endpoint utilizing Query II. To eliminate any potential influence from previous runs, we completely pruned everything on Docker before each load-testing iteration. The report includes the boxplot graph of the iteration that has the most overlap in confidence intervals with the other two iterations, for each spatial database and dataset combination. The scripts used to produce the boxplot graphs and compute the confidence intervals can be found at https://github.com/Radius-Query-Performance-Benchmarking/Analyze_Results, along with step-by-step instructions in the README.md file.

The lack of variation measures, such as variance or confidence intervals, in many papers within the field, raises concerns about the validity of their results. This deviates from standard scientific practices and increases the risk of presenting misleading results [14, 16]. In the Radius Query Performance Benchmarking, we report our results using the 95% confidence intervals for the mean HTTP request duration.

We evaluated the performance of the two spatial databases on the radius query, in terms of the HTTP request duration, with respect to two main factors: the number of virtual users and the number of Stores records stored in the database. In this section, we will present the boxplot graphs for the full dataset with respect to the number of virtual users. The rest of the boxplot graphs will be available in the Appendix.

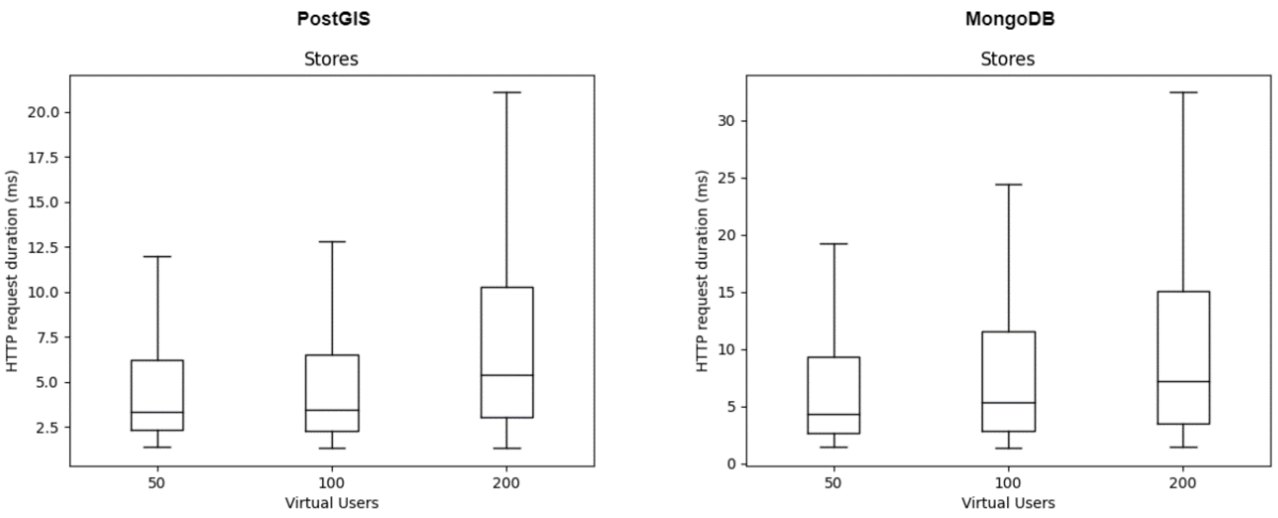


Figure 8: /stores endpoint HTTP request duration with respect to the number of virtual users. Number of Stores records = 145.000

In Figure 8, the boxplots illustrate the change in the HTTP request duration for the /stores endpoint utilizing Query I with respect to the number of virtual users, using the dataset with 145.000 Stores records, for PostGIS and MongoDB respectively. It is evident that PostGIS consistently outperforms MongoDB in Query I in terms of efficiency, regardless of the number of virtual users. For PostGIS, the mean HTTP request duration for the /stores endpoint with 50 virtual users falls within the 95% confidence interval of (5.29, 5.66) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (7.65, 8.29) milliseconds. When the number of virtual users increases to 100, for PostGIS the mean HTTP request duration for the /stores endpoint falls within the 95% confidence interval of (5.68, 6.12) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (9.72, 10.58) milliseconds. Finally, when the number of virtual users increases to 200, for PostGIS the mean HTTP request duration for the /stores endpoint falls within the 95% confidence interval of (9.54, 10.50) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (13.76, 15.17) milliseconds.

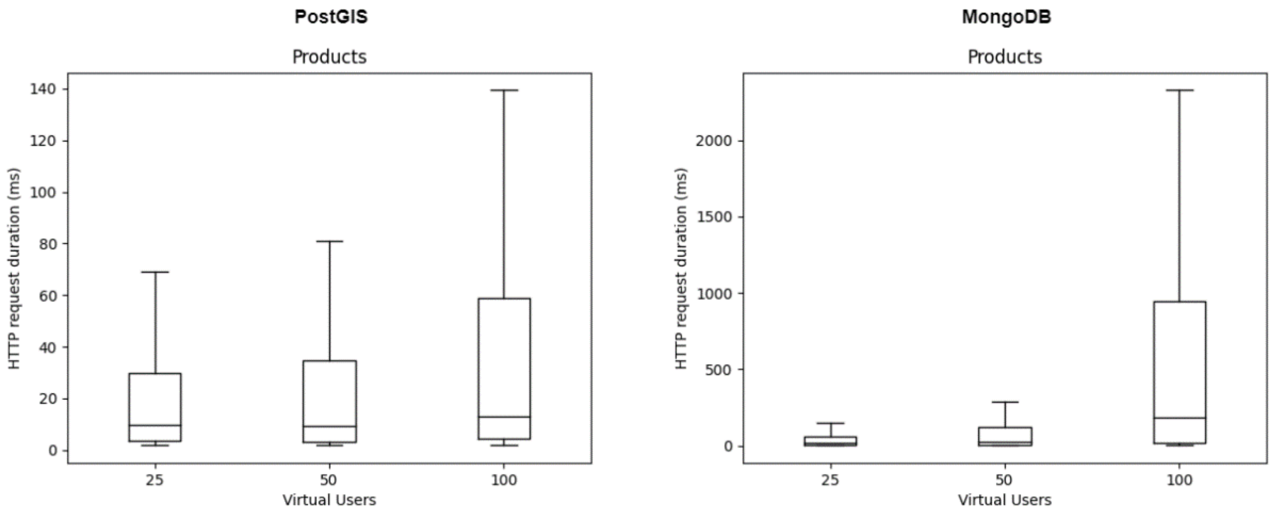


Figure 9: /products endpoint HTTP request duration with respect to the number of virtual users. Number of Stores records = 145.000

In Figure 9, the boxplots illustrate the change in the HTTP request duration for the /products endpoint utilizing Query II with respect to the number of virtual users, using the dataset with 145.000 Stores records and their corresponding Products, for PostGIS and MongoDB respectively. It is evident that PostGIS consistently outperforms MongoDB in Query II in terms of efficiency, regardless of the number of virtual users. For PostGIS, the mean HTTP request duration for the /products endpoint with 25 virtual users falls within the 95% confidence interval of (24.35, 28.95) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (51.78, 63.69) milliseconds. When the number of virtual users increases to 50, for PostGIS the mean HTTP request duration for the /products endpoint falls within the 95% confidence interval of (29.22, 35.64) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (112.24, 139.83) milliseconds. Finally, when the number of virtual users increases to 100, for PostGIS the mean HTTP request duration for the /products endpoint falls within the 95% confidence interval of (67.09, 85.53) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (742.62, 894.35) milliseconds.

The results of the benchmark clearly demonstrate the performance superiority of PostGIS over MongoDB in both queries, regardless of the dataset and number of virtual users considered. Although PostGIS performs better in Query I, the performance difference is not substantial enough to be deemed a decisive factor on its own. It is important to take into account other factors as discussed in [1, 17] before making a final decision on the spatial database selection. On the other hand, when analyzing Query II, there is a notable and significant performance difference between PostGIS and MongoDB which can be considered a decisive factor in favor of choosing PostGIS over MongoDB.

References

[1] B. Huang et al., in *Comprehensive Geographic Information Systems*, Amsterdam: Elsevier, 2018, pp. 8-65

[2] P. A. Longley, G. M. F. Coaut, M. D. J. Coaut, and R. D. W. Coaut, in *Geographical Information Systems and science: Paul A. Longley*, England: Wiley, 2005, pp. 4

[3] Pászto, in *Spationomy*, Springer International Publishing, 2020, pp. 5

[4] A. Brimicombe and C. Li, in *Location-based services and geo-information engineering*, Chichester, UK: Wiley-Blackwell, 2009, pp. 2

[5] D. Bartoszewski, A. Piorkowski, and M. Lupa, "The comparison of processing efficiency of spatial data for PostGIS and MongoDB databases," *Beyond Databases, Architectures and Structures. Paving the Road to Smart Data Processing and Analysis*, pp. 291–302, 2019. doi:10.1007/978-3-030-19093-4_22

[6] A. Makris, K. Tserpes, G. Spiliopoulos, D. Zissis, and D. Anagnostopoulos, "MongoDB VS postgresql: A comparative study on performance aspects," *GeoInformatica*, vol. 25, no. 2, pp. 243–268, 2020. doi:10.1007/s10707-020-00407-w

[7] E. Baralis, A. Dalla Valle, P. Garza, C. Rossi, and F. Scullino, "SQL versus NOSQL databases for geospatial applications," *2017 IEEE International Conference on Big Data (Big Data)*, 2017. doi:10.1109/bigdata.2017.8258324

[8] C. Rossi, M. H. Heyi, and F. Scullino, "A service oriented cloud-based architecture for mobile geolocated emergency services," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 11, 2016. doi:10.1002/cpe.4051

[9] S. Ray, B. Simion, and A. Demke Brown, "Jackpine: A benchmark to evaluate spatial database performance," *2011 IEEE 27th International Conference on Data Engineering*, 2011. doi:10.1109/icde.2011.5767929

[10] J.-G. Lee and M. Kang, "Geospatial Big Data: Challenges and opportunities," *Big Data Research*, vol. 2, no. 2, pp. 74–81, 2015. doi:10.1016/j.bdr.2015.01.003

[11] M. Choy, J.-G. Lee, G. Gweon, and D. Kim, "Glaucus: Exploiting the wisdom of crowds for location-based queries in Mobile Environments," *Proceedings of the International AAAI Conference on Web and Social Media*, vol. 8, no. 1, pp. 61–70, 2014. doi:10.1609/icwsm.v8i1.14534

[12] T. Taipalus, *Database management system performance comparisons: A systematic survey.*, Jan. 2023. doi:10.48550/arXiv.2301.01095

[13] L. M. Weber *et al.*, "Essential guidelines for computational method benchmarking," *Genome Biology*, vol. 20, no. 1, 2019. doi:10.1186/s13059-019-1738-8

[14] M. Raasveldt, P. Holanda, T. Gubner, and H. Mühleisen, "Fair benchmarking considered difficult," *Proceedings of the Workshop on Testing Database Systems*, 2018. doi:10.1145/3209950.3209955

[15] M. Grambow, J. Hasenburg, T. Pfandzelter, and D. Bermbach, "Is it safe to dockerize my database benchmark?," *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, 2019. doi:10.1145/3297280.3297545

[16] T. Kalibera and R. Jones, "Rigorous benchmarking in reasonable time," *Proceedings of the 2013 international symposium on memory management*, 2013. doi:10.1145/2464157.2464160

[17] K. Sahatqija, J. Ajdari, X. Zenuni, B. Raufi, and F. Ismaili, "Comparison between relational and NOSQL databases," *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2018. doi:10.23919/mipro.2018.8400041

[18] J. Gray, in *The benchmark handbook: For database and transaction processing systems*, San Francisco, Cal.: Kaufmann, 1993, pp. 1–17

[19] Y. S. Kanwar, *Benchmarking scalability of NoSQL databases for geospatial queries*, 2019. doi:10.31979/etd.azm5-7asx

[20] D. Sullivan, in *NoSQL for mere mortals®*, Hoboken etc.: Addison-Wesley, 2015, pp. 190–197

[21] H. Garcia-Molina, J. D. Ullman, and J. Widom, in *Database systems: The complete book*, Pearson Education Ltd, 2009, pp. 557–607

Appendix

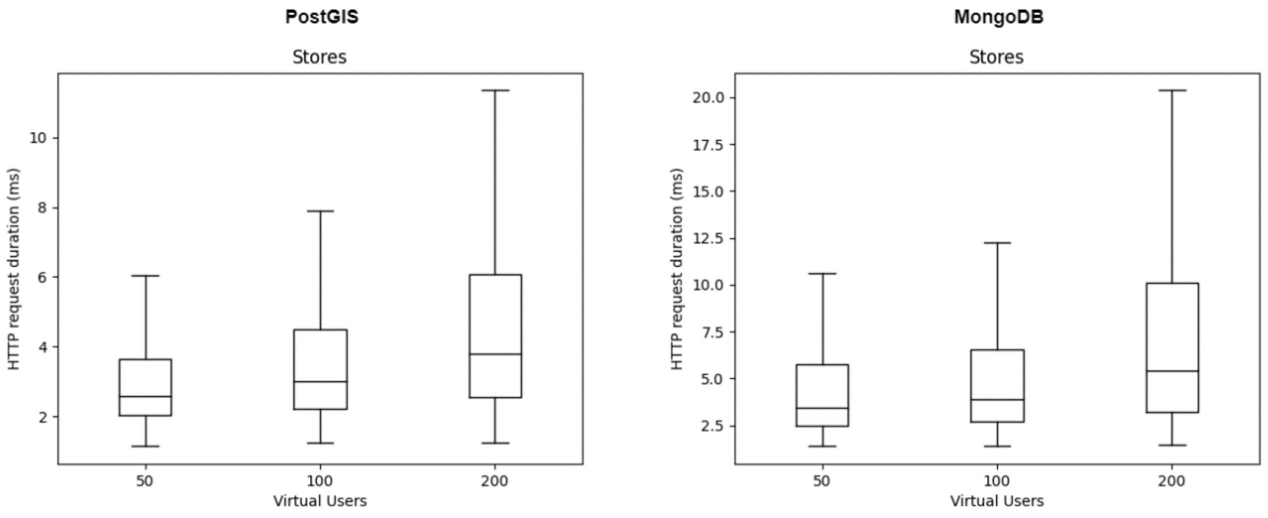


Figure 10: /stores endpoint HTTP request duration with respect to the number of virtual users. Number of Stores records = 50.000

In Figure 10, the boxplots illustrate the change in the HTTP request duration for the /stores endpoint utilizing Query I with respect to the number of virtual users, using the dataset with 50.000 Stores records, for PostGIS and MongoDB respectively. It is evident that PostGIS consistently outperforms MongoDB in Query I in terms of efficiency, regardless of the number of virtual users. For PostGIS, the mean HTTP request duration for the /stores endpoint with 50 virtual users falls within the 95% confidence interval of (3.16, 3.29) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (4.73, 4.98) milliseconds. When the number of virtual users increases to 100, for PostGIS the mean HTTP request duration for the /stores endpoint falls within the 95% confidence interval of (3.80, 3.97) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (5.43, 5.74) milliseconds. Finally, when the number of virtual users increases to 200, for PostGIS the mean HTTP request duration for the /stores endpoint falls within the 95% confidence interval of (5.25, 5.59) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (9.18, 10.03) milliseconds.

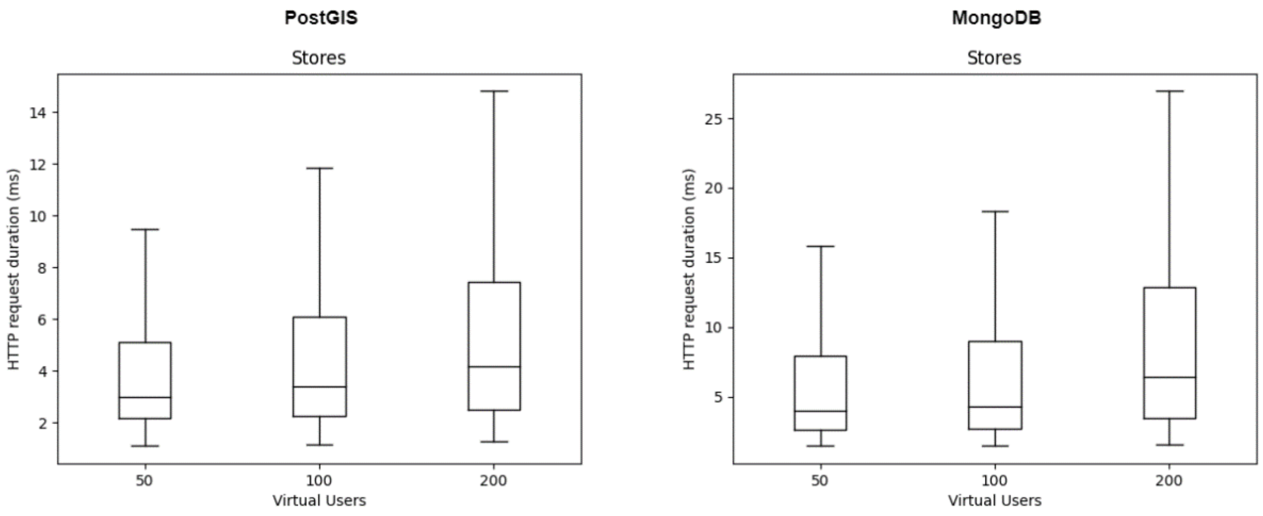


Figure 11: /stores endpoint HTTP request duration with respect to the number of virtual users. Number of Stores records = 100.000

In Figure 11, the boxplots illustrate the change in the HTTP request duration for the /stores endpoint utilizing Query I with respect to the number of virtual users, using the dataset with 100.000 Stores records, for PostGIS and MongoDB respectively. It is evident that PostGIS consistently outperforms MongoDB in Query I in terms of efficiency, regardless of the number of virtual users. For PostGIS, the mean HTTP request duration for the /stores endpoint with 50 virtual users falls within the 95% confidence interval of (4.31, 4.56) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (6.50, 6.99) milliseconds. When the number of virtual users increases to 100, for PostGIS the mean HTTP request duration for the /stores endpoint falls within the 95% confidence interval of (5.19, 5.59) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (7.78, 8.50) milliseconds. Finally, when the number of virtual users increases to 200, for PostGIS the mean HTTP request duration for the /stores endpoint falls within the 95% confidence interval of (6.52, 7.03) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (11.09, 12.08) milliseconds.

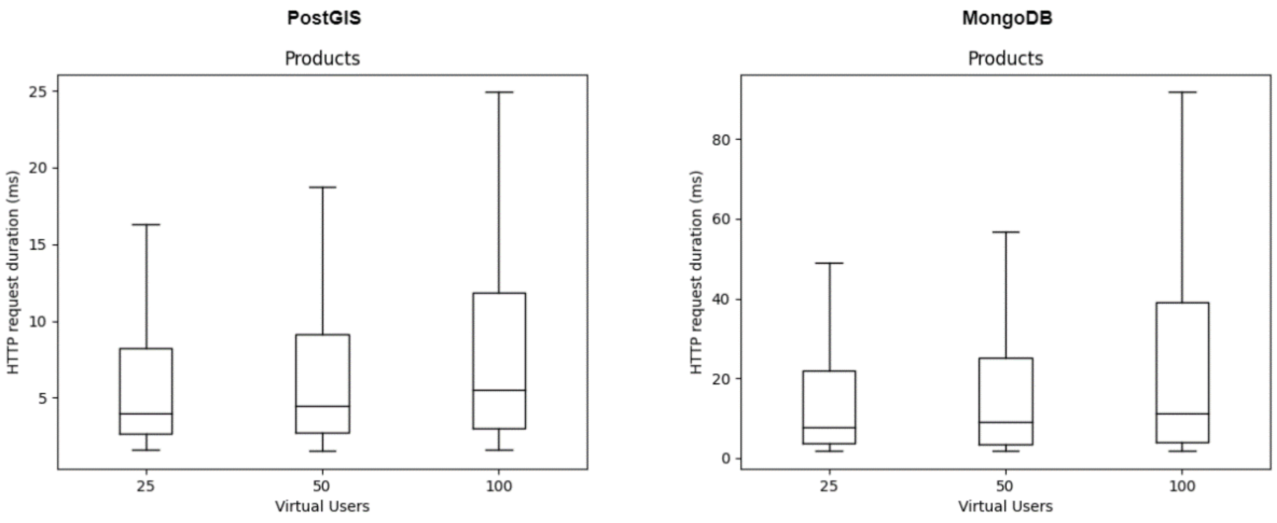


Figure 12: /products endpoint HTTP request duration with respect to the number of virtual users.
Number of Stores records = 50.000

In Figure 12, the boxplots illustrate the change in the HTTP request duration for the /products endpoint utilizing Query II with respect to the number of virtual users, using the dataset with 50.000 Stores records and their corresponding Products, for PostGIS and MongoDB respectively. It is evident that PostGIS consistently outperforms MongoDB in Query II in terms of efficiency, regardless of the number of virtual users. For PostGIS, the mean HTTP request duration for the /products endpoint with 25 virtual users falls within the 95% confidence interval of (6.75, 7.64) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (17.40, 20.33) milliseconds. When the number of virtual users increases to 50, for PostGIS the mean HTTP request duration for the /products endpoint falls within the 95% confidence interval of (7.40, 8.28) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (20.01, 23.28) milliseconds. Finally, when the number of virtual users increases to 100, for PostGIS the mean HTTP request duration for the /products endpoint falls within the 95% confidence interval of (10.70, 12.49) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (37.51, 46.25) milliseconds.

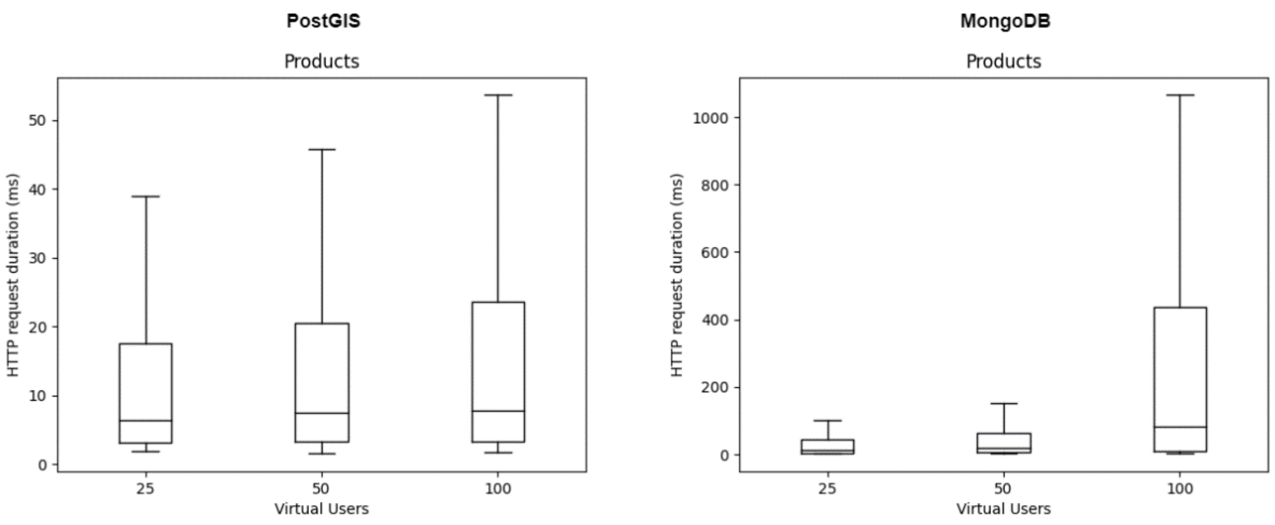


Figure 13: /products endpoint HTTP request duration with respect to the number of virtual users. Number of Stores records = 100.000

In Figure 13, the boxplots illustrate the change in the HTTP request duration for the /products endpoint utilizing Query II with respect to the number of virtual users, using the dataset with 100.000 Stores records and their corresponding Products, for PostGIS and MongoDB respectively. It is evident that PostGIS consistently outperforms MongoDB in Query II in terms of efficiency, regardless of the number of virtual users. For PostGIS, the mean HTTP request duration for the /products endpoint with 25 virtual users falls within the 95% confidence interval of (15.26, 18.45) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (37.79, 46.49) milliseconds. When the number of virtual users increases to 50, for PostGIS the mean HTTP request duration for the /products endpoint falls within the 95% confidence interval of (17.97, 21.58) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (65.31, 82.30) milliseconds. Finally, when the number of virtual users increases to 100, for PostGIS the mean HTTP request duration for the /products endpoint falls within the 95% confidence interval of (25.05, 30.54) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (364.08, 448.44) milliseconds.

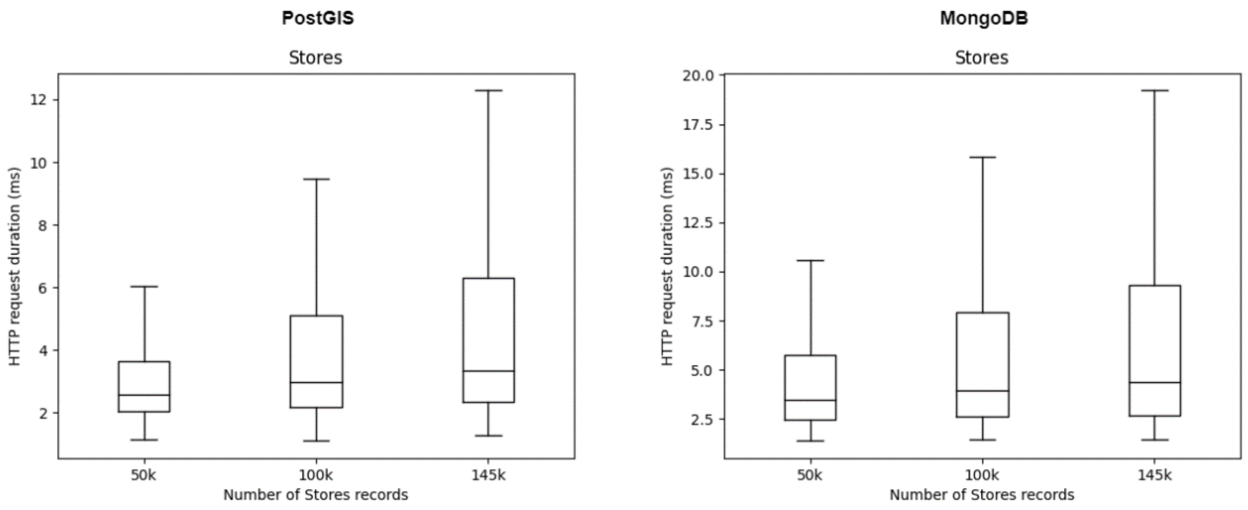


Figure 14: /stores endpoint HTTP request duration with respect to the number of Stores records. Number of virtual users = 50

In Figure 14, the boxplots illustrate the change in the HTTP request duration for the /stores endpoint utilizing Query I with respect to the number of Stores records, using 50 virtual users, for PostGIS and MongoDB respectively. It is evident that PostGIS consistently outperforms MongoDB in Query I in terms of efficiency, regardless of the number of Stores records. For PostGIS, the mean HTTP request duration for the /stores endpoint with 50.000 Stores records falls within the 95% confidence interval of (3.16, 3.29) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (4.73, 4.98) milliseconds. When the number of Stores records increases to 100.000, for PostGIS the mean HTTP request duration for the /stores endpoint falls within the 95% confidence interval of (4.31, 4.56) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (6.50, 6.99) milliseconds. Finally, when the number of Stores records increases to 145.000, for PostGIS the mean HTTP request duration for the /stores endpoint falls within the 95% confidence interval of (5.34, 5.72) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (7.65, 8.29) milliseconds.

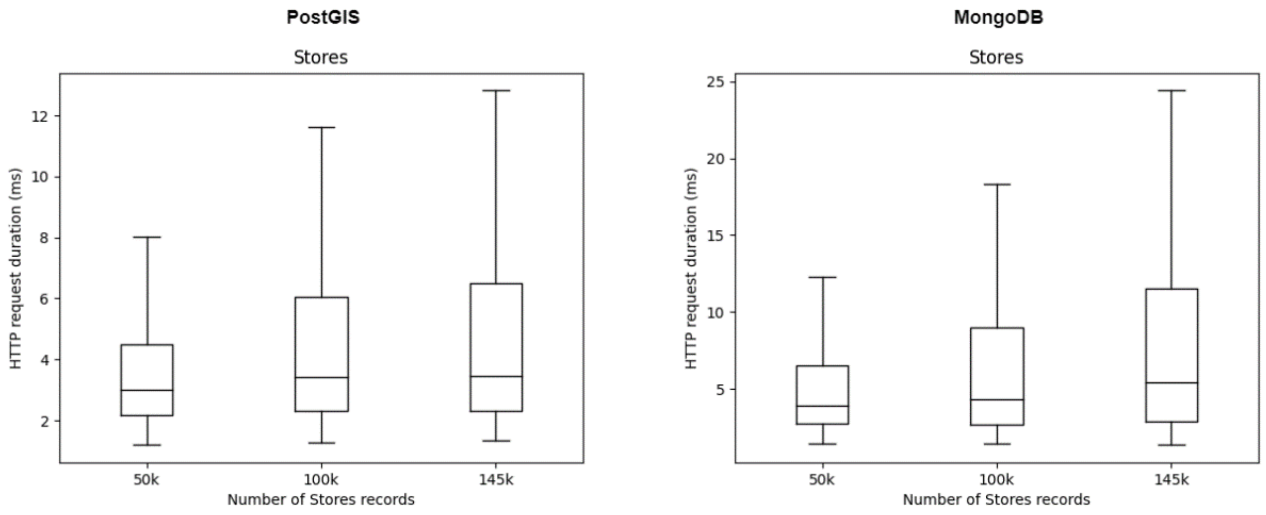


Figure 15: /stores endpoint HTTP request duration with respect to the number of Stores records. Number of virtual users = 100

In Figure 15, the boxplots illustrate the change in the HTTP request duration for the /stores endpoint utilizing Query I with respect to the number of Stores records, using 100 virtual users, for PostGIS and MongoDB respectively. It is evident that PostGIS consistently outperforms MongoDB in Query I in terms of efficiency, regardless of the number of Stores records. For PostGIS, the mean HTTP request duration for the /stores endpoint with 50.000 Stores records falls within the 95% confidence interval of (3.729, 3.89) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (5.43, 5.74) milliseconds. When the number of Stores records increases to 100.000, for PostGIS the mean HTTP request duration for the /stores endpoint falls within the 95% confidence interval of (5.18, 5.56) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (7.78, 8.50) milliseconds. Finally, when the number of Stores records increases to 145.000, for PostGIS the mean HTTP request duration for the /stores endpoint falls within the 95% confidence interval of (5.68, 6.12) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (9.72, 10.58) milliseconds.

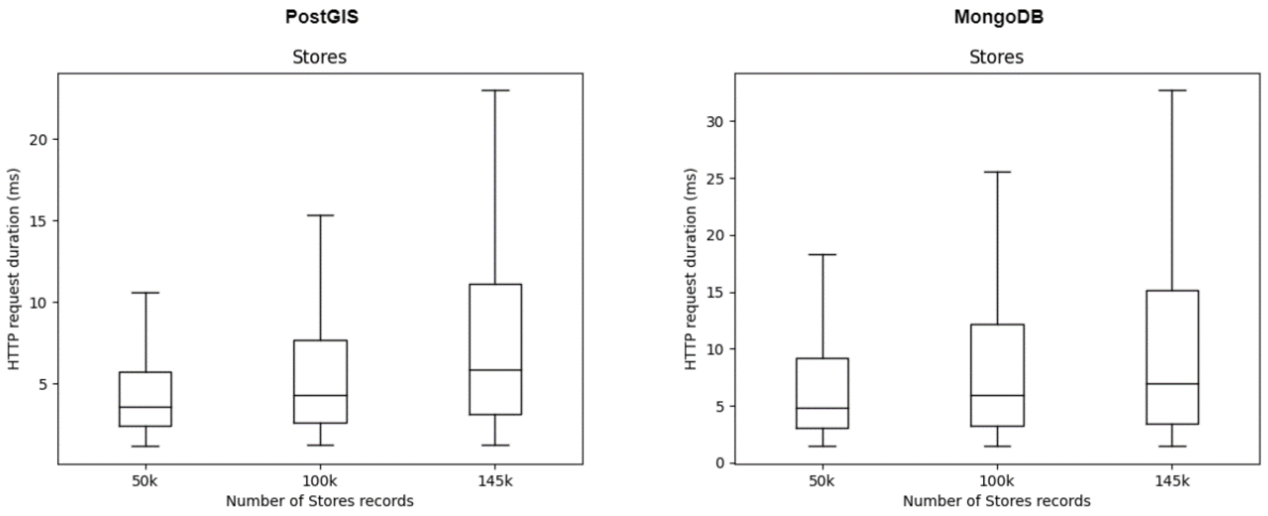


Figure 16: /stores endpoint HTTP request duration with respect to the number of Stores records. Number of virtual users = 200

In Figure 16, the boxplots illustrate the change in the HTTP request duration for the /stores endpoint utilizing Query I with respect to the number of Stores records, using 200 virtual users, for PostGIS and MongoDB respectively. It is evident that PostGIS consistently outperforms MongoDB in Query I in terms of efficiency, regardless of the number of Stores records. For PostGIS, the mean HTTP request duration for the /stores endpoint with 50.000 Stores records falls within the 95% confidence interval of (5.03, 5.35) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (8.40, 9.18) milliseconds. When the number of Stores records increases to 100.000, for PostGIS the mean HTTP request duration for the /stores endpoint falls within the 95% confidence interval of (6.70, 7.20) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (10.74, 11.73) milliseconds. Finally, when the number of Stores records increases to 145.000, for PostGIS the mean HTTP request duration for the /stores endpoint falls within the 95% confidence interval of (10.00, 10.98) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (14.04, 15.58) milliseconds.

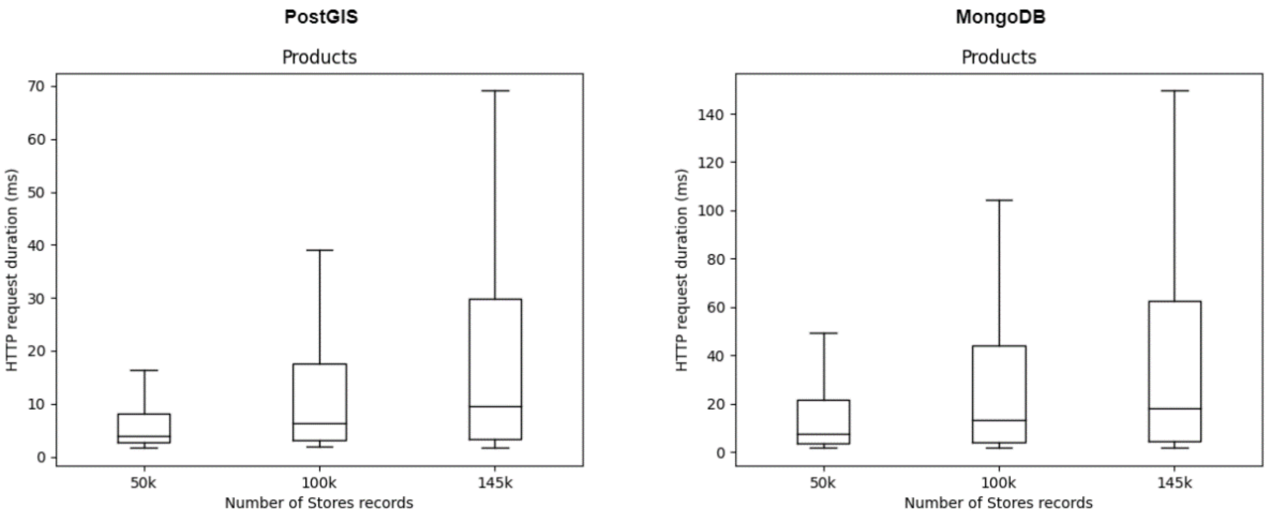


Figure 17: /products endpoint HTTP request duration with respect to the number of Stores records. Number of virtual users = 25

In Figure 17, the boxplots illustrate the change in the HTTP request duration for the /products endpoint utilizing Query II with respect to the number of Stores records and their corresponding Products, using 25 virtual users, for PostGIS and MongoDB respectively. It is evident that PostGIS consistently outperforms MongoDB in Query II in terms of efficiency, regardless of the number of Stores records and their corresponding Products. For PostGIS, the mean HTTP request duration for the /products endpoint with 50.000 Stores records falls within the 95% confidence interval of (6.75, 7.64) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (17.20, 20.22) milliseconds. When the number of Stores records increases to 100.000, for PostGIS the mean HTTP request duration for the /products endpoint falls within the 95% confidence interval of (15.26, 18.45) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (38.89, 47.89) milliseconds. Finally, when the number of Stores records increases to 145.000, for PostGIS the mean HTTP request duration for the /products endpoint falls within the 95% confidence interval of (24.35, 28.95) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (51.78, 63.69) milliseconds.

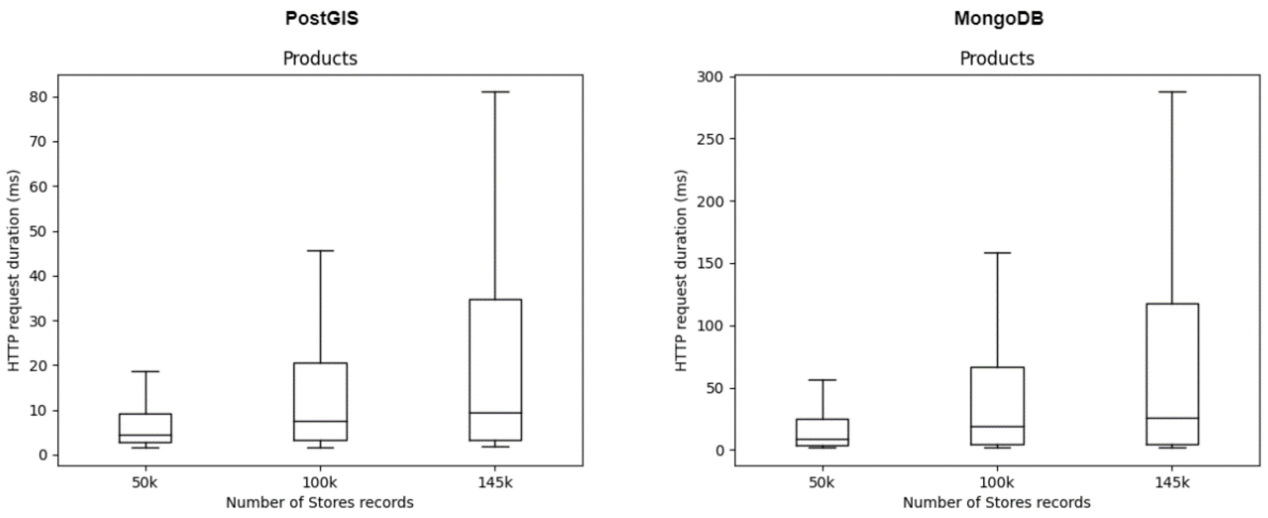


Figure 18: /products endpoint HTTP request duration with respect to the number of Stores records. Number of virtual users = 50

In Figure 18, the boxplots illustrate the change in the HTTP request duration for the /products endpoint utilizing Query II with respect to the number of Stores records and their corresponding Products, using 50 virtual users, for PostGIS and MongoDB respectively. It is evident that PostGIS consistently outperforms MongoDB in Query II in terms of efficiency, regardless of the number of Stores records and their corresponding Products. For PostGIS, the mean HTTP request duration for the /products endpoint with 50.000 Stores records falls within the 95% confidence interval of (7.40, 8.28) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (20.01, 23.28) milliseconds. When the number of Stores records increases to 100.000, for PostGIS the mean HTTP request duration for the /products endpoint falls within the 95% confidence interval of (17.97, 21.58) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (63.99, 81.25) milliseconds. Finally, when the number of Stores records increases to 145.000, for PostGIS the mean HTTP request duration for the /products endpoint falls within the 95% confidence interval of (29.22, 35.64) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (109.42, 136.23) milliseconds.

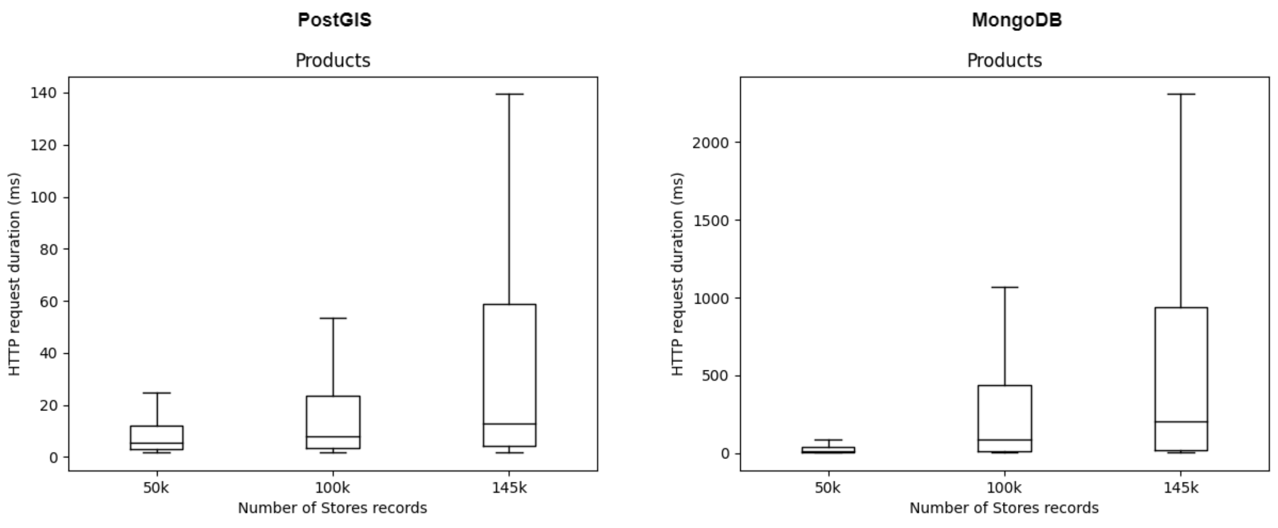


Figure 19: /products endpoint HTTP request duration with respect to the number of Stores records. Number of virtual users = 100

In Figure 19, the boxplots illustrate the change in the HTTP request duration for the /products endpoint utilizing Query II with respect to the number of Stores records and their corresponding Products, using 100 virtual users, for PostGIS and MongoDB respectively. It is evident that PostGIS consistently outperforms MongoDB in Query II in terms of efficiency, regardless of the number of Stores records and their corresponding Products. For PostGIS, the mean HTTP request duration for the /products endpoint with 50.000 Stores records falls within the 95% confidence interval of (10.70, 12.49) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (37.45, 46.45) milliseconds. When the number of Stores records increases to 100.000, for PostGIS the mean HTTP request duration for the /products endpoint falls within the 95% confidence interval of (25.05, 30.54) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (364.08, 448.44) milliseconds. Finally, when the number of Stores records increases to 145.000, for PostGIS the mean HTTP request duration for the /products endpoint falls within the 95% confidence interval of (67.09, 85.53) milliseconds, whereas, for the same scenario the mean HTTP request duration for MongoDB falls within the 95% confidence interval of (751.75, 903.64) milliseconds.