Benchmarking the Maximum Throughput with a Geospatial Workload on MongoDB while Scaling Out the System

Emil Cyprian Balitzki
Computer Science (Informatik) B.Sc.
Technische Universität Berlin

***Abstract*—In this paper, we will discuss the need and the efficiency of sharding a cloud-deployed MongoDB database containing the geospatial data. With the increasing number of potential customers, a maximal throughput supported by a given architecture can give applicable insights when deciding on the system's final design. We will ask and answer the research question on how the maximum throughput with geospatial workload changes while scaling out the MongoDB database. With the scope of the benchmark focusing on three scenarios: with zero, two, and three shards, we will compare these scenarios regarding the maximum throughput they can achieve and the latency of requests querying them. In total, we have conducted over 18 benchmarks, which showed the superiority of a not-sharded geospatial MongoDB database while also proving the increased efficiency of a highly sharded database over less sharded one. Lastly, we have discussed the implications of our study for selecting the number of shards when sharding a geospatial MongoDB database.**

***Index Terms*—MongoDB, geospatial data, throughput, benchmark**

I. Introduction

Nowadays, almost every system needs to store the data, and whether the data are the accounts of the users, posted pictures on social media, or news articles, the go-to method is a database, which allows for easy querying of the data to receive necessary insights and information. While one can create their own physical server, with the Cloud being a highly scalable environment, more and more companies turn into setting up their businesses in a distributed fashion. Platforms such as Google Cloud Platform [1] allow for easy deployment of all necessary services, such as web servers, networks, and the databases needed for storing the data. Additionally, within the world, where everything and everyone is connected, storing location data for different places or events is necessary. Clients want to know the locations of the nearest highly rated Chinese restaurants or the direction to the concert of famous pop artists. Thus, many business owners are looking for a way to store their geospatial data, and among many choices, they need to decide which database they will use. MongoDB [2] is one of such options for storing geospatial data, with the support of GeoJSON documents and with the ability to be deployed on the Cloud. Further, together with the growth of the business, more data will be stored, and more clients will want to access their data. After a while, one machine will not be enough to keep all the data. Thus, the maximal throughput becomes a vital resource, where one wants to serve as many clients simultaneously as possible, and the horizontal scaling of servers might be necessary. MongoDB (MDB) supports distributing geospatial data across multiple machines, called sharding, and, as depicted by [1, Sharding], one can see an increase in the performance while querying data kept on multiple machines (shards) at once. However, implementing new shards is not free and will require time, effort and will increase the system's costs, while the system undergoes the change. Developers need to balance the trade-off between the amount of added shards and the costs of adding them. One also needs to decide how many shards are profitable and necessary to increase the maximum throughput of the database. Therefore, in this paper, we will deploy a MongoDB database on the Google Cloud Platform and address the following research question:

> *RQ 1: How does the maximum throughput with a geospatial workload change while scaling out the MongoDB database?*

To manage the scope of the study, we will base our research on three sharding scenarios:

- Scenario 1: MongoDB which was not sharded (0S),
- Scenario 2: MongoDB with two shards (2S) and,
- Scenario 3: MongoDB with three shards (3S).

A benchmark will be executed for each scenario, with the database filled with pre-generated geospatial data. Firstly, we will measure the maximum throughput the MongoDB can sustain using only a single machine (0S). Afterward, we will determine the same quality for the system with two (2S) and three (3S) shards, respectively. As the use case, we will

[1] https://cloud.google.com/docs
[2] https://www.mongodb.com

consider an imaginary restaurant search application, which will store geospatial data containing the locations and details of restaurants worldwide. The database will be increasingly queried using different types of read queries, such as radius search, rating search, or only-restaurants-with-outside-area search. Thus, the number of clients will be increasing, extensively stressing the system until the maximum throughput can be determined. By looking at our benchmark results, we found out that the 0S scenario significantly outperformed other scenarios, while the 3S scenario managed to keep higher throughput and lower latency than the 2S scenario. Finally, we will discuss the implications of our findings on selecting the number of shards when sharding a geospatial MongoDB database.

II. Background

In this section, we will briefly summarise important background concepts, which we will use in our study.

A. MongoDB and Sharding

MongoDB is a free and easy to use and deploy, mostly non-relational database, with optional database schemas. It was developed by *MongoDB Inc* [3] and stores data in flexible, JSON-like files, called documents. MongoDB supports many diverse storage and querying options, such as geospatial data, and is a distributed database at its core, supporting horizontal scaling, called **sharding**. Sharding distributes data across multiple machines (shards) to support deployments with extensive data sets and high throughput operations. All of the components of a sharded database and interactions between them are listed in the official MongoDB documentation [1, Sharding]. Sharded collection is partitioned and distributed across the shards in the cluster and can be held together with non-sharded data collections at the same shard.

B. Geospatial Data

Geospatial data is data, which describes places, objects, events, or other features with a location on or near the surface of the Earth. It combines the location information, usually in the form of coordinates on the Earth, and other attribute information (such as characteristics of the objects, its name or type), with the possibility of additional temporal information (the time or life span at which the location exists). The location may be static in the short-term (e.g., the location of an earthquake event) or dynamic (e.g., a moving vehicle) [2, p. 171]. *Geospatial data* can be used for many applications: transport and mobility, marketing, tracking and storing information [3, p. 2]. One of the options to store such geospatial data are *GeoJSON* objects [4]. Each of the *GeoJSON* objects is a *JSON* file (or document, in context of MongoDB), which contains a type field that specifies the *GeoJSON* object type (such as linestrings, polygon, or point), together with static location coordinates in the form of longitude and latitude pairs. Additional metadata information about the objects, such as their name or identifier, can be included. An example for such *GeoJSON* stored in the MongoDB can be seen below:

```
{
   _id: ObjectId("61f040272bd1a11381d2a281"),
   Name: 'Chinese Restaurant 232',
   location: {
     type: 'Point',
     coordinates: [ -97.2925, 20.3639 ]
   }
}
```

C. Geospatial Queries

The database loaded with geospatial data can be queried with geospatial queries, where the idea behind them is similar to the geospatial data. Geospatial queries contain the location information used to receive the needed geospatial data. Examples of such geospatial queries include getting the closest to the provided coordinate point, getting all data in a specified area, or other operations which use coordinates. Additional filter parameters used in standard queries can also be added to reduce the received data set. While every geospatial database supports different sets of operations, more information about the specifics of MongoDB's geospatial queries can be found in its official documentation [1, Geospatial Queries].

D. Throughput and Latency

Following the [5, p. 50]'s definitions, **throughput** describes the number of requests that a cloud service can handle in parallel, including both a number and some notion of time. This means that the number of successfully completed requests is counted within a given time interval. **Throughput** is typically represented using *requests per second* units. On the other hand, **latency** describes the time necessary to complete an individual request, often presented in milliseconds. The amount of data that can be transmitted decreases when the network latency increases [4], thus making the **throughput** and **latency** correlated.

III. Study Design

In this section, we will discuss the overall architecture of the benchmark, how its components are related to each other, the design of the benchmarking client, and finally, the execution of the benchmark.

A. Benchmark Architecture

In order to understand the results of the benchmark, it is important first to understand its overall architecture. The first component is the System Under Test (SUT), which due to our research question, is clear. MongoDB supports geospatial data and geospatial queries and is one of the most popular open-sourced, NoSQL databases used on the market. Furthermore, like many other Web Service Applications, it uses JSON files to exchange data. Thus making it a perfect database for storing geospatial data and an overall database selection for bigger

[3] https://www.mongodb.com

[4] https://www.comparitech.com/net-admin/latency-vs-throughput/

projects. More information about the MongoDB can be read in the II-A section. In our study, the MongoDB database architecture changes depending on the specific sharding scenario. The second important component is the benchmarking client, which stays the same for every scenario. Starting from the preloading phase until the end of the benchmark a connection with MongoDB is maintained using a client driver. Both of the components are explained in more detail in the subsections below.

B. Sharding Scenarios

The main difference between different sharding scenarios is the number of shards of the MongoDB database. As explained in the II-A, MongoDB can be sharded or, in other words, scaled out horizontally. When talking about a sharded MongoDB deployment, we refer to the set of nodes comprising it as a *sharded cluster*. A sharded cluster consists of several components: config server, which stores the metadata and the configuration settings for the cluster, individual shards containing a subset of the sharded data, and one or more *mongos* router processes, which act as a query router, providing an interface between client applications and the sharded cluster. Multiple replica set members can be deployed for shards and config server to ensure that data is redundant and that the system is highly available. For production configuration, it is recommended to deploy at least three-member replica sets for the config server, and each of the shards [1, Sharded Components]. While benchmarks, in general, should follow the production configurations and real-world scenarios as much as possible, we decided not to use replica sets for our benchmark. The main reason behind that choice was the project's scope, where the simplicity of not including the replication sets required less code and time on our behalf. One of the purposes of the replica sets is to provide redundancy and fault tolerance against the loss of a single database server, which is not strictly necessary for the execution of our benchmark. On the other hand, we need to measure the maximum throughput to answer our research question. We are ending the benchmark when specified latency is exceeded or the connection to the database timeouts (more about the benchmark client's design and its termination rules can be read in IV-C), thus even if the mentioned redundancy nor automated failover is not necessary, the increased data availability would reduce the number of timeouts. When discussing the scenarios where shards are present, all incoming initial geospatial data will be distributed across the shards when loading up the database. In order to decide which shard will contain which data, a *hashed sharding* strategy is used, where for each document, a hash of the shard key field's value is computed, and each shard is then assigned a range based on the hashed shard key values [1, Hashed Sharding]. As mentioned in the documentation, while a range of shard keys may be "close", their hashed values are unlikely to be on the same shard, which will force cross-sharded queries. The cross-sharded nature of the queries when using the hashed sharding was precisely the reason, why we have chosen the mentioned strategy. Finally, with the basis of sharding scenarios explained, we will discuss each of the sharding scenarios in detail.

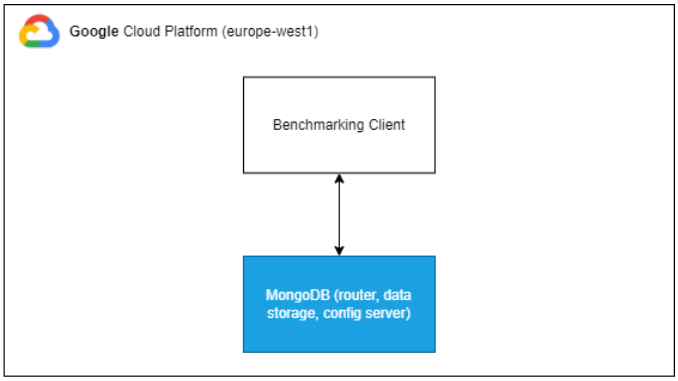

Fig. 1. The architecture of no shards scenario.

1) No Shards Scenario: The first and most simple scenario is MongoDB without sharding. It is a basic case where MongoDB has not been scaled out, and there exists only one machine, which contains all of the necessary components: data storage, client interface, and storage for configuration settings and metadata. When geospatial data is loaded to the database using the client interface, it will not be sharded and stored on a single VM. Since only one machine will be used, queries will not be worked on by multiple machines in parallel, which is the case for other scenarios. Additionally, a second VM will be deployed containing the benchmarking client itself. Above mentioned architecture can be seen in Figure 1.

2) Two Shards Scenario: Second scenario is where MongoDB contains two shards: *shard1* and *shard2*. Both shards are independent and hosted on different VMs. A Config Server is hosted on another virtual machine to store the metadata and configuration settings. Furthermore, on another VM, a mongos router will be set up, and it will receive all incoming queries for both of the shards. As in every scenario, the benchmarking client will also be deployed. Like the first scenario, from the start of the benchmark, the benchmarking client will connect to the mongos router to query it for geospatial data. Two shards scenario can be seen in Figure 2, excluding the more transparent Shard 3.

3) Three Shards Scenario: The last scenario will consist of three shards: *shard1*, *shard2* and *shard3*. While the architecture of this scenario is almost the same as the two shards scenario, the main difference is the addition of the third VM containing one of the shards, which can be seen in the Figure 2. Referring to the official documentation of the MongoDB [1, Sharding], the addition of another shard will distribute the workload across even more shards, allowing each shard to process a subset of cluster operations. Finally, as already said in the Introduction (I), in order to manage the scope of this

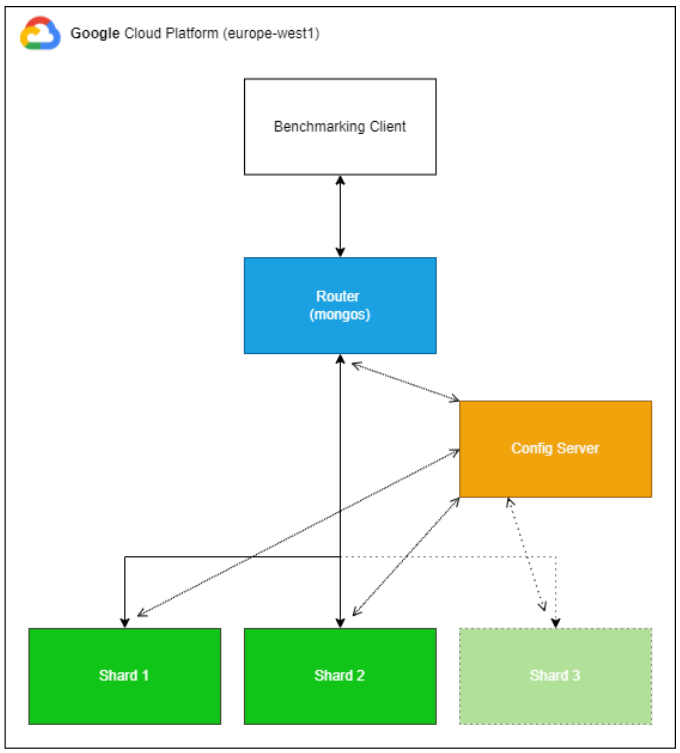

Fig. 2. The architecture of two and three shards scenarios.

study, this will be the last scenario covered in our benchmark, but it should be mentioned that one could further increase the number of shards.

C. Benchmarking Client

Another benchmark component is the benchmarking client, hosted on a separate virtual machine in every scenario. The benchmarking client will always create and maintain a connection to the mongos router, which provides an interface for accessing the data inside of the database. During the first phase, the preload phase, a previously generated dataset, as described in the IV-E, is loaded into the database. After the preload phase, the benchmark phase starts automatically, where client processes will be increasingly created to simulate the increase in the customers' simultaneous connections to the database, thus allowing for the measurement of the maximum throughput. Owing to the nature of the benchmarking phase, the warm-up phase is not implemented and, in fact, not necessary for our benchmark. The database will be gradually warmed up with the increase of the client processes and be ready for measurement before reaching the maximum throughput. During the benchmark phase, all previously created client processes will simultaneously send randomly generated queries, described in more detail in IV-D, and log timestamps of sending the request and arrival of responses. The benchmarking client itself will keep track of the number of total requests sent, the number of requests for which the response time exceeded the selected latency threshold, and if in any of the client processes a timeout exception has arisen. The 98% Percentile rule will be checked every specified time interval, and if more than 2% of the request had violated the latency threshold, the benchmark phase would end. The second option for termination of the benchmark is when a timeout would be detected, and the MongoDB database would not be reachable. Upon the end of the benchmark, the main process will close all client processes and run the clean-up phase, consolidating all log files generated during the benchmark into a single file.

IV. Implementation

In this section, we will describe the deployment and configuration of database components and the implementation details of the benchmarking client itself. Additionally, we will talk about the specifics of dataset generation, together with the characteristics of query generation. Finally, we will also describe the tools, methods, and choices made while designing the benchmark.

A. Cloud Deployment

During cloud service benchmarking, the choice of the Cloud Provider is essential. For this benchmark, we will be using *Google Cloud Platform* (GCP) to host and deploy our virtual environments (VMs). The reason behind the choice is that the GCP is one of the better supported and documented Cloud Providers. Additionally, we were provided with credits for benchmarking on GCP at the start of the project, making it easy to deploy VMs on it. Another option was the Amazon Web Services (AWS) [5], but due to their practice of paying for the usage of internal network traffic, they have been not chosen. The benchmark is designed to be fully deployed automatically, meaning it can be executed on a brand new GCP account. In order to run the benchmark, one needs first to set up a new host VM, and then different steps of the benchmark need to be executed via the Linux terminal. More information and documentation on how to run the benchmark can be read in the GitHub repository [6]. Automatic deployment would not be possible without the use of the architecture-as-code software tool, such as *Terraform* [7]. It allows for easy deployment of different cloud resources via a code. All three sharding scenarios, as well as the benchmarking client, have their own folder containing a *.tf* file, which when run connects the GCP account, provided in the *credentials.json* file in the root directory of the repository. *Terraform* will deploy all necessary VMs, in order to run the benchmark, thus making it fully automatic and repeatable. The same *.tf* files will also destroy and free all deployed resources when the benchmark is over. In order to use the *Terraform*, a *Makefile* file in the root directory needs to be run, which contains commands for deploying specific sharding scenarios and cleaning them up after the benchmark. Lastly, for the benchmark deployment of all of the VMs we have selected the *europe-west1-b* zone inside of the

[5]https://aws.amazon.com/
[6]https://github.com/Corgam/mongo_benchmark
[7]https://www.terraform.io/

europe-west1 GCP region. The zone and the region can be changed manually in the *.tf* files.

B. Virtual Environments and Operating System

For the Virtual Environments themselves, the GCP's *e2-standard* machine series was chosen. It is a standard, general-purpose series, which one could use for production deployment. For all database components, we have used *e2-standard-4*, which proved to be enough to run the benchmark sufficiently long but could be increased in order to give more individual resources to each of the components. For the benchmarking client itself, *e2-standard-8* was necessary due to the increased RAM requirements of it, together with the fact that the benchmarking client's resources should never be a bottleneck for the whole benchmark [5]. Lastly, the host machine executing the benchmark should be run on at least *e2-standard-8* machine due to the memory costs of dataset generation.

Another important decision was the choice of the operating system (OS). After some trials and errors, we have settled upon the latest long-term support (LTS) version of *Ubuntu*, namely *Ubuntu Focal Fossa* [8]. Where the choice of Linux-based OS should not be a surprise, it is important to mention that the minimal version of it was chosen as the image for all components. Namely, the image name specified in the *Terraform* was *ubuntu-os-cloud/ubuntu-minimal-2004-focal-v20211209*. The main advantage of the minimal version of the *Ubuntu* was that it does not come with a large package overhead, meaning it does not have any additional packages and software applications installed from the start. Developers can easily choose and download all necessary packages while not worrying about the performance of the VM due to the vast amount of unused, installed packages. In our benchmark, several *sh* scrips will install all necessary packages, such as *python3-pip* or *mongodb-org*, in order to run the experiment. Finally, it should be noted that during all benchmark runs, the resources of all VMs were monitored to ensure no resource bottleneck.

C. Benchmarking Client

Benchmarking Client simply put is a *Python 3* [9] script run on the *Ubuntu Minimal Focal*. Before running the main script, Python's package installer Pip [10] installs all necessary Python libraries listed in the *requirements.txt* file. Finally, the *client.py* file is run, where the whole code for the benchmarking client is located. The code is divided into a few functions, corresponding to different benchmarking phases. Firstly, the preload phase starts, and a connection to the MongoDB is established, with the use of the *PyMongo*, a MongoDB's client driver for Python, where the previously generated dataset is loaded into the database. Due to the size of the dataset, it is loaded with the use of *insert many* command. The validation step and the insert order will be skipped to speed up the process (the correctness of the data is checked during the generation). Later, the benchmark phase starts. In each of the iterations of the main loop, firstly, a new five client processes are created [11], then the main process is put asleep for 30 seconds until it will run the latency check routine. Each of the client processes will create a connection to the MongoDB database and continuously send generated geospatial queries one after another. For the check routine three process safe variables are initialised with the use of the multiprocessing Manager class, namely: *badLatencies*, *totalRequests* and *ifExceptionsArrised*. These variables will also be passed to all clients' processes, and as mentioned in the Study Design section, they will be used to decide whether the benchmark should end. Benchmark termination will occur if 98% Percentile rule for 100 milliseconds of latency is violated or there happens at least a single timeout (with a generous 500 seconds waiting time) when connecting to the database. When the latency check if failed, the main process will close all client processes and run the clean-up phase.

During the benchmark phase, several log files are created, which will contain all logs created during the benchmark. In addition to the main log file containing metadata for the benchmark run, for each of the worker processes, a single *csv* log file is created. In order to not run into multiprocessing lock issues, each worker access only a single file, and all logs are consolidated into a single file during the clean-up phase. Final results are then automatically uploaded to the host VM with the use of *Python* script and *Secure Copy Protocol* (SCP).

D. Query generation

For our benchmark, we have decided to use a synthetic, closed workload generation, where during the benchmark, each of the client processes continuously sends generated geospatial queries. A new query is sent when the response to the last one is received. Firstly, a simple basic query is created for a randomly selected city from the *cities_above_1000.csv* dataset described in the IV-E. Together with the city location coordinates, a random radius of the query is chosen, with a value between 5000 and 50 000 meters. The idea behind the simple query is to ask for the closest ten restaurants in a given radius and location, as one would firstly do in a real-world restaurants application scenario. The worker will stop on the basic query with the probability of $\frac{1}{5}$, or query the database further. With the probability of $\frac{3}{5}$, an additional random parameter (the type of the cuisine, the rating, the pricing, or if a restaurant has an outside area or not) will be added to the query. An example of an extended query can be seen below.

```
{ "location": {
    "$nearSphere": {
        "$geometry": {
```

[8]https://cloud-images.ubuntu.com/minimal/releases/focal/release/

[9]https://www.python.org/

[10]https://pypi.org/project/pip/

[11]For the first few benchmark runs, we have increased the number of client processes only by one, which was later changed to mentioned five every iteration. The reason behind this change was to decrease the execution time of the benchmark.

```
            "type": "Point",
            "coordinates": COORDINATES
        },
        "$maxDistance": RADIUS }
    },
    "Cuisine": {
        "$eq": "Japanese"
    }
}
```

Finally, instead of adding additional parameters, with the probability of $\frac{2}{5}$, additional ten restaurants will be requested with the same radius and location parameters. The last type of query simulates the *next page* button inside of the application.

E. Dataset generation

For our benchmark, we decided to use a synthetic, pre-generated geospatial dataset, generated by the *generation.py* on the benchmark host VM and not on the benchmarking client, in order to speed up sequential benchmarks. Later, the dataset is uploaded to benchmarking client using *Terraform*. As for the generation itself, it starts with loading up the publicly available *cities_above_1000.csv* dataset [12], containing a list of 139,281 cities from around the world with a population of over 1000 people. Each city record contains its name, population, coordinates, and other data. For each city, based on its population, its radius and number of restaurants inside of it are calculated. This calculation is based on the proportion of the cities' population in comparison to the cities with the smallest and the biggest population in the dataset. In other words, firstly, a value range starting from 1000 people (several smallest cities in the dataset) up to 22 315 474 million people (population of Shanghai, China) is created. Then, cities' population is placed somewhere within the mentioned range, and according to the placement, the radius and restaurant number are calculated. For the smallest cities with a population of 1000 people, the radius is chosen arbitrarily as 1000 meters due to the lack of data online about small cities' surface area, and the number of restaurants is again chosen arbitrarily as 2, for the same reason. On the other hand, there is enough information about Shanghai. Shanghai's surface area is 6340 square kilometers, which translated to the circle area means, that the circle must have 44 927 meters in radius. For the number of restaurants in Shanghai, the number was chosen as 100 000 [13]. Taking Tehran in Iran as the example, it has a population of 7 153 309 million people, and would receive the value of 15 080 meters for the radius and 643 as the number of restaurants inside it. Additionally, it should be mentioned that the population of 1000 people is assumed for the cities with missing populations. The dataset's size can also be increased by tweaking the threshold values for the radius and number of restaurants.

After the calculations, the restaurants are distributed inside of the circle, with the radius being the calculated value and the middle being the coordinates of the selected city [14]. As seen in Figure 3, the distribution favors placing more restaurants closer to the middle of the circle, which is also true for the real-world placement of the restaurants, where more of them are opened closer to the city center. Additionally, it should be mentioned that the distribution is a subject to the *Tissot's indicatrix* distortion [15]. For each of the restaurants, a GeoJSON object is created with key-value pairs for location coordinates (generated with the use of the mentioned distribution), city name, and randomly generated attributes of the restaurant, such as its name, cuisine type, year of opening, rating score, number of reviews, pricing range and if it has an outside area or not. An example for such generated restaurant GeoJSON object can be seen below:

```
{
    'Name': 'Vegan Dinner 939',
    'City': 'Ahuateno',
    'Cuisine': 'Vegan',
    'Opened': 1998,
    'Rating': 4,
    'Reviews': 378,
    'Pricing': '$$',
    'Outside_area': True,
    'location': {
        'type': 'Point',
        'coordinates': [-98.14782132533719,
        21.007412785394337]
    }
}
```

The random generation of the attributes is done with the use of the *random* Python library, either by randomly choosing an option from an array or generating random integers. After all restaurants are generated, they are put together in a single JSON file, encoded using *utf-8* encoding, compressed into the *.gz* format, using the *gzip* [16] Python library, and saved on the disk as a file. Finally, the same pre-generated dataset is used for each sharding scenario.

F. Results Analysis

For the analysis of the results, we have used the convenient *Jupyter Notebooks*[17] with Python as the language of choice. The final results were firstly pre-processed: the header was deleted and the results were sorted by timestamp, then plotted using *Pandas*[18] and *MatPlotLib*[19] Python libraries.

V. Results

In this section, we will present the benchmark results and answer the research question stated in the Introduction.

[12] https://public.opendatasoft.com/explore/dataset/geonames-all-cities-with-a-population-1000

[13] https://news.cgtn.com/news/2019-07-09/Shanghai%2DThe%2Drestaurant%2Dcapital%2Dof%2DChina%2DIbIuOtcIww/index.html

[14] https://stackoverflow.com/questions/43195899/how-to-generate-random-coordinates-within-a-circle-with-specified-radius

[15] https://en.wikipedia.org/wiki/Tissot%27s_indicatrix

[16] https://docs.python.org/3/library/gzip.html

[17] https://jupyter.org/

[18] https://pandas.pydata.org/

[19] urlhttps://matplotlib.org/

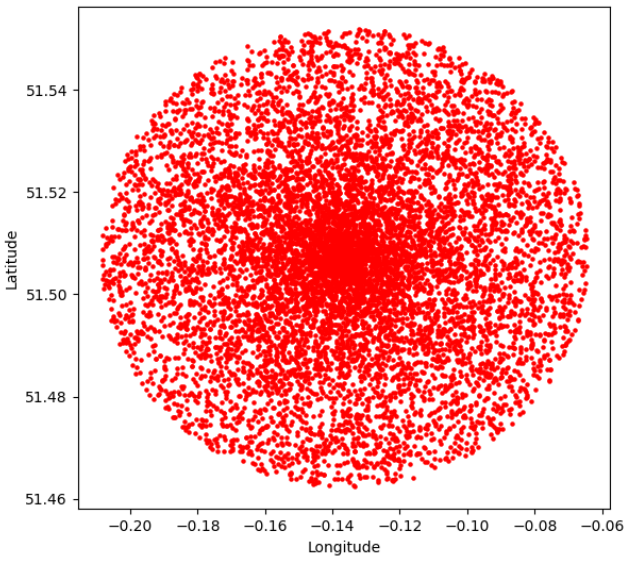

Fig. 3. An example distribution of 10 000 restaurants inside of circle with 5000 meters radius and the coordinates (-0.136439, 51.507359) as the middle.

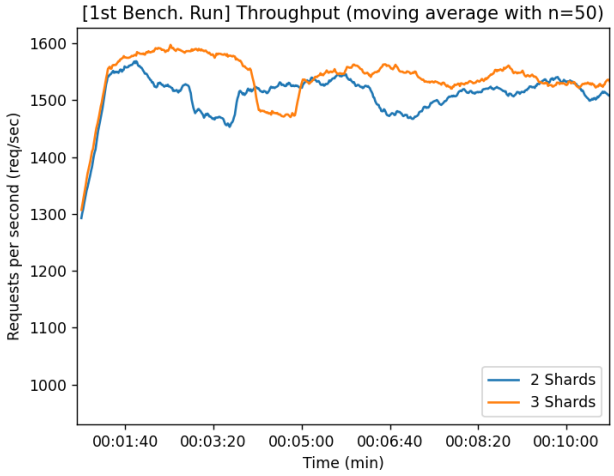

Fig. 4. The throughput for 2S and 3S scenarios (1st Bench. Run) with the increase by 5 client processes every 30 seconds. Data re-sampled into 1 second intervals.

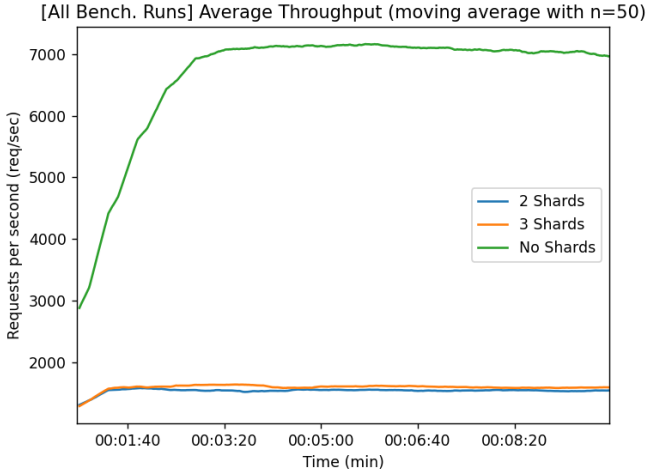

Fig. 5. The average throughput for all scenarios (All Bench. Runs) with the increase by 5 client processes every 30 seconds. Data re-sampled into 1 second intervals.

A. Gradual Throughput Increase

In the first experiment, we gradually added a small number of client processes to see how the throughput would behave for different sharding scenarios. Instead of 5 client processes every 30 seconds, we started by increasing the number of processes by one every 30 seconds. The total amount of client processes was capped at 40, and the latency check routine stayed the same as described in the Implementation section. For each of the sharding scenarios, three benchmarking runs were executed. After a while, the throughput for two and three shard scenarios started to stabilize and stayed relatively the same, around 1600 requests per second. While the same could not be observed for the 0S scenario, further runs will confirm the similarity. For 2S and 3S scenarios, we have also tested longer waiting periods between the increase of the number of client processes, which resulted in the same stabilization of the throughput; thus, we kept the 30 second waiting period for every next run.

B. Throughput Characteristics

After the initial runs, we have increased the number of client processes by 5 every 30 seconds. Similarly to the initial experiments, we have conducted three runs for every sharding scenario. The results for 2S and 3S scenarios from the first benchmark run be seen in the Figure 4. From the 4 Figure, it can be seen that the throughput for the 3S scenario most of the time was, while not by a great amount, higher than the one for the 2S scenario, with two exceptions: around the fourth minute and the ninth minute. Different throughput behaviour from the second and third benchmark runs can be observed in the Figures 6 and 7, where 3S scenario had clearly higher throughput then the 2S scenario. Even if the first benchmark run was not as clear as the next two, the three-shard scenario generally had a higher throughput in every plot. Throughput for the 0S scenario always stayed much higher than other scenarios for all benchmark runs, thus in this paper, we will not include figures for all runs and instead provide an average throughput from all benchmark runs for all sharding scenarios, which can be seen in the Figure 5. Additionally, 0S had the same mentioned throughput stabilization behavior as in other scenarios, which can be seen in Figure 5

C. Maximum Throughput

Finally, to answer the research question stated in the Introduction, we have also calculated the average values from all benchmark runs for the mean, the minimum, the maximum,

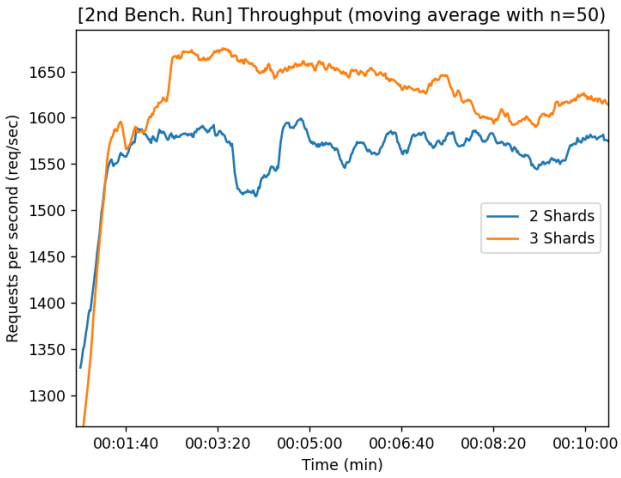

Fig. 6. The throughput for 2S and 3S scenarios (2nd Bench. Run) with the increase by 5 client processes every 30 seconds. Data re-sampled into 1 second intervals.

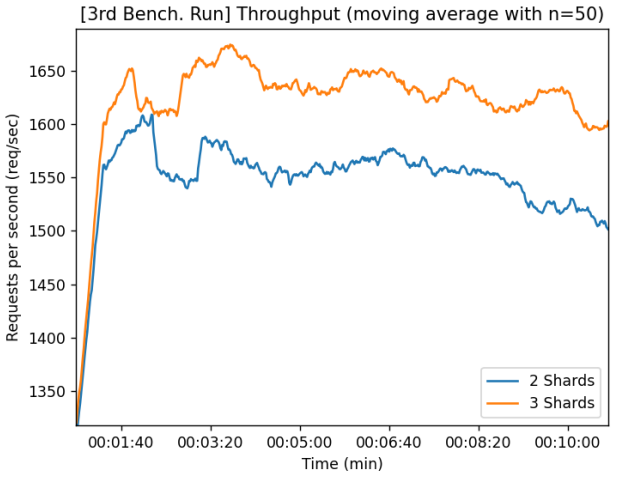

Fig. 7. The throughput for 2S and 3S scenarios (3rd Bench. Run) with the increase by 5 client processes every 30 seconds. Data re-sampled into 1 second intervals.

and the 95% confidence interval, summarized in the table below. These values can be also seen in the Figures 8 and 9. Additionally, it should be mentioned that the first and last few benchmark measurements were disregarded.

	Mean	Minimum	Maximum	95% CI
0S	7003 req/sec	6729 req/sec	7259 req/sec	6998-7008 req/sec
2S	1543 req/sec	1453 req/sec	1609 req/sec	1542-1545 req/sec
3S	1602 req/sec	1470 req/sec	1675 req/sec	1600-1604 req/sec

The total improvement of the mean value for the 3S scenario over the 2S scenario was as slight as 3.85%. Furthermore, there was an improvement of around 354% of the 0S scenario over both other scenarios. The results show the superiority of the

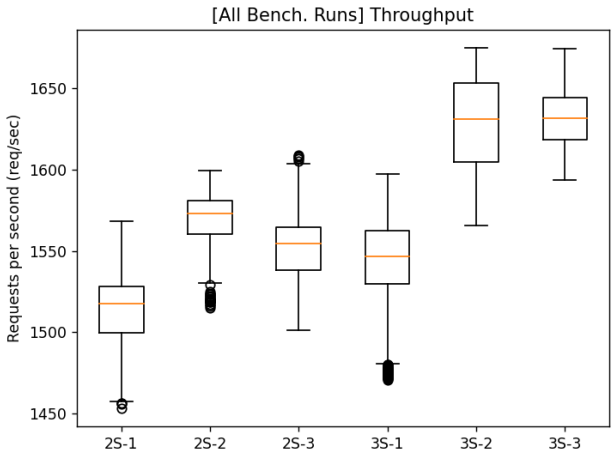

Fig. 8. The throughput for 2S and 3S scenarios (All Bench. Runs) with the increase by 5 client processes every 30 seconds. On *y*-axis, firstly the scenario name and later the benchmark run number.

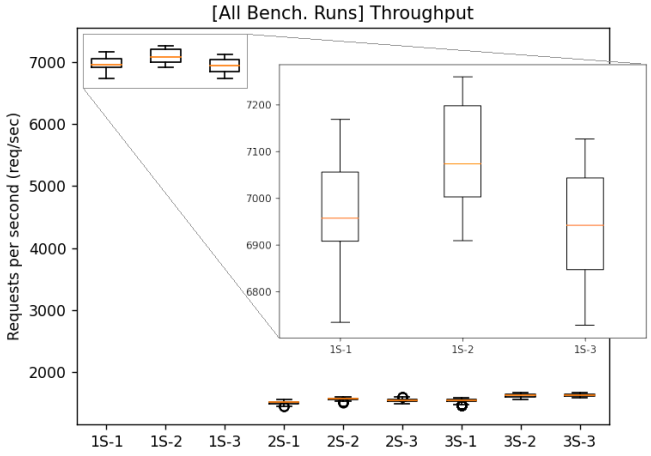

Fig. 9. The throughput for all scenarios (All Bench. Runs) with the increase by 5 client processes every 30 seconds. On *y*-axis, firstly the scenario name and later the benchmark run number. Inside the zoom, 0S scenario values from specific runs.

not-sharded databases in both throughput and the latency over the sharded databases. This statement is valid for the scenarios where the data stored inside the databases are small enough to be kept on a single machine. Additionally, databases with more shards seem to respond faster and support a higher number of queries per second. These differences are further discussed in the VI section.

D. Latency

Another quality we wanted to look at during the benchmark is latency. As could be expected from the previous figures and from the results we measured, which can be seen in the Figure 10, one can see that the 0S scenario had the lowest latency. Furthermore, during the whole benchmark, the 3S scenario had

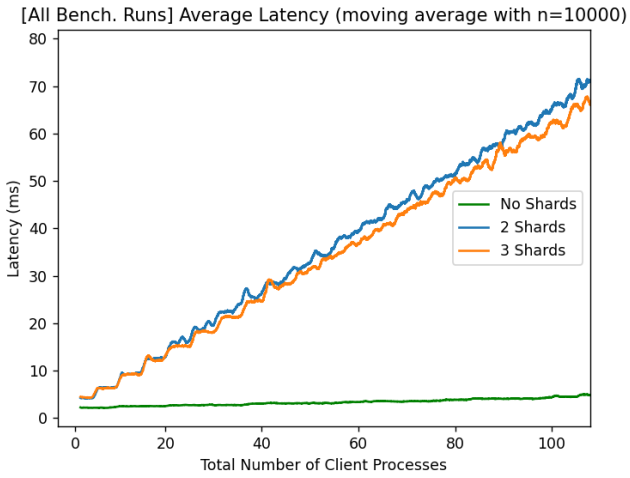

Fig. 10. The average latency for all scenarios (All Bench. Runs) with the increase by 5 client processes every 30 seconds.

lower latency than the 2S scenario, even though the difference was again not that big. Due to the correlation between the latency and the throughput mentioned in the II-D, the overall lower latency of 3S is justified. Finally, the increase in the number of client processes can be seen by the characteristic ”steps” in the latency lines. The overall latency increase during the benchmark is also justified due to the increasing number of client processes and the higher load MongoDB needs to handle after each increase. The previously set latency threshold of 100 milliseconds leading to the termination of the benchmark was never reached, which is valid for all scenarios. All benchmark runs have ended due to the appearance of the connection timeout, even with the generous upper time limit of 500 seconds. The timeouts could be possibly prevented, as discussed in the VI. Finally, due to the lower and less varying latency of the 0S scenario, it can be said that it is the most reliable database architecture from all shown in this study.

VI. Discussion

In this section, we will talk about the implications of our study to the selection of the number of shards when deploying a production-ready MongoDB database containing geospatial data. Following, we will address the threats to the validity of our study, its limitations, and future work which can be done.

A. Study implications to the selection of the number of shards

As already mentioned in the Results section, in our benchmark, the non sharded database was superior to the sharded ones. This is probably due to the communication overhead of the scaled-out systems, where multiple machines need to communicate with each other to fulfill queries. Additionally, network traffic can lead to higher monthly costs, depending on the cloud provider, as mentioned in the IV-A. Thus, when the geospatial data needed to be stored in the database is small enough to be stored on a single machine, it is advised to keep the architecture simple and does not scale out the system. While the scaling out comes with a significant increase in infrastructure costs, when the data becomes too big to be held on a single machine, one should consider sharding the MongoDB database. Horizontal scaling can potentially lead to better efficiency than vertical scaling and positively impact transaction performance due to the reduced load on individual instances, thus making sharding a viable option. When queries are executed on sharded databases, each shard has to process a subset of cluster operations, and all shards execute the operations in parallel, making the response time generally faster. Before sharding the database, the final number of shards needs to be selected; however, it can be changed later. From our results, we can see that a higher amount of shards can lead to higher throughput and smaller latency while increasing the complexity of the infrastructure. As later discussed in the limitations of our study, this property could change with a higher number of shards than we have examined in this paper. Official MongoDB documentation is an obligatory reading before deploying a production-ready environment. Sharded cluster infrastructure requirements and complexity require careful planning, execution, and maintenance since sharding collection cannot be undone [1, Considerations]. The developer needs to fiddle with different parameters when considering the sharding, such as sharding strategy, shard key choice, and understand the operational restrictions for sharded MongoDB databases.

B. Study validity and limitations

Primarily due to the scope of this study, we have simplified many essential choices and possible deployment steps. The most noticeable one is the lack of replicas for different MongoDB components. While, as mentioned in the official MongoDB documentation, it is recommended to deploy at least three replicas for each shard and config server, with the additional possibility to deploy multiple mongos routers, we deliberately did not follow this recommendation. As mentioned in the III-B, one of the purposes of the replica sets is to provide redundancy and fault tolerance against the loss of a single database server, which is not strictly necessary for the execution of our benchmark. Another fundamental reason for replicas sets is to provide increased read capacity as clients can send read operations to different servers, which could influence our benchmark results. Multiple mongos routers could decrease the number of timeouts and provide additional interfaces for benchmarking client connections but would require more time and effort when setting up the benchmark, which was impossible due to our study’s time and credits limitations. Creating multiple benchmarking clients with their separate connections to different mongos routers would be advised when re-implementing the benchmark. Additionally, at least three members replica set should be implemented for the config server and all shards.

Another significant limitation was the query generation and the warm-up phase. While we believe the warm-up phase was unnecessary for our benchmark, more studies could be done in that direction. We have only used a read-only workload

accessing the previously inserted dataset when querying the database. Preferably, as stated in the [5, p. 15], a mix of different workloads such as update-heavy and read-only workloads are typically needed to sufficiently stress the SUT, thus should be used. Changing the workload model would require more effort on our side when implementing the benchmark, which was not possible due to the scope of this study.

Our scenarios included the not sharded MongoDB database, which proved that our pre-generated dataset was not big enough to force sharding of the database. In our use case, the database contained several gigabytes of data (approximately 14.5 million documents), split on all shards equally and should be further increased. Additional study of the database with more than three shards at a single time containing a more extensive dataset could be executed. Such study would provide an overview of the correlation between the increase in the number of shards and its tendency to increase the system's overall throughput. After executing such benchmark, it could be said to which extent it is beneficial to increase the number of shards of the MongoDB database.

When incrementing the number of client processes, a different number of additional processes added every iteration could be set. For our benchmark, we have studied the change in throughput for five clients every 30 seconds, and we briefly saw what happened when the number changed by only a single client at a time. Due to the long benchmark time, we did not settle on a single additional process every 30 seconds. Additional study could be conducted, where the number of processes would increase gradually by one process every 30 seconds until the termination of the benchmark. This change would lead to a smoother throughput increase when plotting the results. We also looked into the increase of the time interval between every iteration of the while loop, which in our understating would not change the benchmark results due to the throughput stabilization after several minutes of running the benchmark. When executing similar benchmarks, one could tinker with different values for time intervals and increase client processes every iteration.

Lastly, it should also be mentioned that for study, we have self-deployed all MongoDB components, while there exists an official cloud-native service *MongoDB Atlas*[20], which guides the deployment of MongoDB databases. When using such a service, one could follow all of the crucial advice when setting up and deploying the database. Naturally, we believe such a service would be used during the production-ready deployment.

VII. Conclusion

In this paper, we have depicted the changes to the maximum throughput with a geospatial workload on the MongoDB database while scaling it out. Our conducted benchmark answered the research question stated in the Introduction section: *"How does the maximum throughput with a geospatial workload change while scaling out the MongoDB database?"*. We have introduced all necessary background definitions needed to understand our study. After explaining the Study Design of our benchmark, we showed and explained all components of the sharded MongoDB architecture. Following the execution of the benchmark, we provided its results and analyzed their implications when choosing the number of shards for a geospatial MongoDB database. The results conveyed the superiority of non sharded databases over the sharded ones with the 354% average throughput mean increase. When comparing the sharded scenarios, the 3S scenario had a slightly better throughput and lower latency than the 2S scenario, with 3S having a 3.85% average throughput mean increase. When sharding a database, the necessity of sharding the database itself should be considered first. If the data is small enough to be stored on a single machine, sharding will not further improve the efficiency of such a database. When sharding the data which needs to be stored on multiple machines, one should decide on the overall number of shards, keeping in mind the increased efficiency of a highly sharded database.

[20]https://www.mongodb.com/atlas/database

References

[1] "Official mongodb documentation," 2022, sections: Sharding: /manual/sharding/, Geospatial Queries: /manual/geospatial-queries/, Sharded Components: /manual/core/sharded-cluster-components/, Hashed Sharding: /manual/sharding/hashed-sharding, Considerations: /manual/sharding/considerations-before-sharding , Accessed Feb. 21, 2022. [Online]. Available: https://docs.mongodb.com/

[2] K. Stock and H. Guesgen, "Chapter 10 - geospatial reasoning with open data," in *Automating Open Source Intelligence*, R. Layton and P. A. Watters, Eds. Boston: Syngress, 2016, pp. 171–204. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128029169000105

[3] S. Jozefowicz, M. Stone, and E. Aravopoulou, "Geospatial data in the uk," *The Bottom Line*, 2019. [Online]. Available: https://www.emerald.com/insight/content/doi/10.1108/BL-09-2019-0115/full/html

[4] H. Butler, M. Daly, A. Doyle, S. Gillies, T. Schaub, and S. Hagen, "The GeoJSON Format," RFC 7946, Aug. 2016. [Online]. Available: https://www.rfc-editor.org/info/rfc7946

[5] D. Bermbach, E. Wittern, and S. Tai, *Cloud service benchmarking*. Springer, 2017. [Online]. Available: https://link.springer.com/book/10.1007%2F978-3-319-55483-9