Spring 2019

# Benchmarking Scalability of NoSQL Databases for Geospatial Queries

Yuvraj Singh Kanwar
*San Jose State University*

# Benchmarking Scalability of NoSQL Databases for Geospatial Queries

A Project Report

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

Of the Requirements for the Degree

Master of Computer Science

By

Yuvraj Singh Kanwar

May 2019

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

---------------------------------------------------------------------------

Dr. Suneuy Kim

---------------------------------------------------------------------------

Dr. Robert Chun

---------------------------------------------------------------------------

Mr. Ramesh Polisetty

APPROVED FOR THE UNIVERSITY

---------------------------------------------------------------------------

## ACKNOWLEDGMENTS

My deepest gratitude and thanks to my project adviser, Dr. Suneuy Kim, for her support, encouragement, and guidance throughout. Without her efforts and her fabulous ideas for realizing this project, this venture would not be possible. I want to pay my special gratitude to my committee members, Dr. Robert Chun, and Mr. Ramesh Polisetty, for their beneficial suggestions and inputs throughout the research. I would also like to appreciate and thank the faculty of the Department of Computer Science at San Jose State University for the vast plethora of knowledge they imparted to us. The staff of the Department has always put in every effort to maintain classes and sessions to help students like us. Without their support, it would not be a plausible task to fulfill this endeavor

## ABSTRACT

NoSQL databases provide an edge when it comes to dealing with big unstructured data. Flexibility, agility, and scalability offered by NoSQL databases become increasingly essential when dealing with geospatial data. The proliferation of geospatial applications has tremendously increased the variety, velocity, and volume of data that the data stores must manage. Such characteristics of big spatial data surpassed the capability and anticipated use cases of relational databases. Because we can choose from an extensive collection of NoSQL databases these days, it becomes vital for organizations to make an informed decision. NoSQL Database benchmarks provide system architects, who shoulder a considerable burden of selecting the right technology for their data stores, with a vital start point and source of information. The major utility of these benchmarks is reproducing experiments on similar experimental data that can verify and optimize the process of selecting an optimum tool for data management needs in the early phases of the development. The goal of this research is to develop a benchmark that can compare the performance of NoSQL databases for querying complex geospatial data. We have analyzed throughputs, latencies, and runtime of MongoDB and Couchbase to identify the correct fit for our use case. This way we have also demonstrated a systematic process that can be followed to make an optimum choice of datastore. This benchmark can be extended easily to any NoSQL database that supports geospatial querying.

***Index Terms* – Benchmarking, Couchbase, Geospatial Queries, MongoDB, NoSQL Databases, Performance Evaluation, Replication, Scalability**

## Table of Contents

## Table of Figures

## Table of Tables

## 1. Introduction

Relational databases were prevalent for a long time, but in recent times, NoSQL databases have shown tremendous progress, and they are quickly replacing relational databases for big data management and storage in the large enterprise data warehouse. Relational databases face numerous challenges when competing with NoSQL databases due to their fixed schema requirement. In spatial context one is usually found managing big data generated by sensors or satellites. Schema generated by modern sensors/satellites for geodatabases is generally vague. There are many types of geospatial fields such as Point, Line, or Polygon that are generated by Geographic Information Systems (GIS).

In most cases, it causes scalability issues and inefficient performance. Scalability and agility provided by NoSQL databases become a strong reason to avert from using relational databases in big data scenarios [1]. Most modern geospatial applications like Foursquare (MongoDB), SimpleGeo (Cassandra), API for City of Portland (CouchDB), PokemonGO (Couchbase), Where (MongoDB), Google Earth (Google Big Table), Scrabbly (MongoDB) and many more have preferred using NoSQL databases to manage spatial data, because of their capability to deal with large variety, velocity and volume of data [19]. A few years ago, we did not possess any alternative to relational databases, that have always faced a major issue when managing 'Raster' which is essential in GIS [15]. New GIS applications use schema-free NoSQL databases that have been adapted to fix this [15].

We have a large plethora of NoSQL databases to choose when setting up Geospatial datastores. Some of these are MongoDB, Cassandra, Couchbase, Google Big Table, Document DB, etc. NoSQL Database benchmarks provide system architects, who shoulder a huge burden of picking the right technology for their datastores, with vital start point and source of information [2]. It is a current trend to select datastore after understanding and evaluating the needs of applications that will use the data. Through benchmarks, one can make an unbiased choice of database which serves queries from the target applications efficiently. The most important utility of these benchmarks is conducting experiments on trustworthy experimental data that can verify and optimize the process of selecting data store for specific applications in the early phases of their development. So, one should perform benchmarking on data that is close to the use-case. In this project, we developed a performance and scalability benchmark, extended to Yahoo! Cloud Serving Benchmark (YCSB), the most prominent NoSQL database benchmark by big data enterprises. Development of YCSB has promoted many types of research in the benchmarking field [6]. However, not a lot of research has been done to benchmark the spatial capability of NoSQL databases. YCSB currently does not support geospatial workloads. We developed an architecture over the current implementation of YCSB to support spatial workloads. MongoDB and Couchbase were chosen for this research as they are current leaders in NoSQL databases which support Spatial Querying. Both, MongoDB and Couchbase make it easier to work on spatial data and our dataset because they support geoJSON. Also, both MongoDB and Couchbase provide their implementations of Indexes for spatial data. The developed benchmark can be extended easily to any NoSQL database that supports Spatial Querying. Using the developed benchmark, we measure throughput, latencies (minimum, maximum, Average), and measures of spread (upper 95th percentile and upper 99th percentile Latencies) of queries from three different geospatial workloads. We analyzed the performance of chosen NoSQL databases by varying performance factors such as number of nodes, data size, number of points fetched, and database request distribution used. An important factor to consider when scaling is replication and replication strategies [2]. Distributed databases serving the cloud deploy replication mechanisms to provide high availability, high scalability and fault tolerance [3]. We have also studied the impact of replication factor on the scalability of the systems. The investigation is done to analyze the performance impact on scalability for Latest, Uniform, and Zipfian request distributions. The research findings and products of this project are

- A developed benchmark and an architecture that can extend to any number of spatial NoSQL databases
- Geospatial Workload support to YCSB
- Scalability performance analysis for MongoDB and Couchbase for graffiti abatement incidents use case.

The rest of the research is organized as described: Section 2 states our research objective and discusses contributions of this research to develop an architecture for NoSQL geodatabase benchmark. Section 3 elaborates upon some of the researches that have already taken place in the NoSQL database domain and the spatial database domain. It also discusses research areas that have already matured for spatial NoSQL database and the research areas that still need development. Section 4 describes the fundamentals of NoSQL databases and why it is a perfect fit for a spatial data store. We have briefly explained the concepts and features of MongoDB and Couchbase NoSQL databases that are relevant for this research. Section 5 discusses the details of the proposed benchmark architecture and our use case. In this section, we have also discussed some geospatial operations and provided a comparison of how several databases provide these spatial operations. Systematic benchmarking process and phases required to benchmark are explained in detail in this section. In Section 6 we have given a detailed analysis of experiments conducted on MongoDB and Couchbase to benchmark their spatial capability for our use case.

## 2. Project Objective and Contributions

An important advantage of NoSQL databases, deployed as distributed data stores, is high availability by utilizing horizontal scalability. Adding a greater number of nodes to the system and replicating data across these nodes can boost performance because of the greater number of nodes to serve requests. NoSQL databases use eventual consistency protocols which ensure that replicated data will get consistent sometime in the future [3]. There are numerous parameters in NoSQL databases that can affect scalability for large clusters. We can develop a benchmark to evaluate the performance of scalability of NoSQL databases for geospatial applications. If we converse in the context of spatial Informatics, we need databases deployed over multiple data centers because of their sheer data size. Schemas for spatial data is generally vague or complex. We need databases to store images from satellites and another kind of spatial data such as weather info or traffic density; this means large datasets. The main contribution of this research is

- An architecture of developed benchmark that supports spatial workloads.
- Extend the designed architecture to support geospatial workloads in YCSB.
- Demonstrate an efficient process to achieve scalability benchmark analysis.

With the developed benchmark we can solve some very pertinent dilemmas, empirically and analytically, that decision makers require answered early on in their lifecycle. Some important results from the developed benchmark are:

- Throughput (in operations/second) of various NoSQL Databases.
- Average Latency (in µs) of representative geospatial queries and writes.
- Impact on retrieval and latency for fetching the increasing number of geospatial points.
- Upper 99-percentile and upper 95-percentile latency measures of spread (in µs) for reads (separately for various spatial querying operations like near, within and intersect) and writes.
- Throughput (in operations/second) comparison by varying system size and request distributions: Zipfian, Latest, and Uniform.
- Impact on response time is measured for fetching an increasing number of geospatial records.

The developed benchmark can be extended to accommodate different datasets or workloads such as natural disasters, traffic, waterbodies, satellite imageries, City Incidents, etc. The created benchmark can be easily extended to accommodate any number of NoSQL spatial databases by developing just their respective clients.

## 3. Related Work

Creation of YCSB by Brian Cooper has prompted many large enterprises to focus on benchmark result analysis of distributed databases to select an optimal database [6]. Since then it had also promoted many types of research for benchmarking NoSQL databases. Cooper et al. developed a benchmark to demonstrate scalability tradeoffs for various systems, namely PNUTS, Cassandra, HBase and sharded MySQL [6]. Rabl et al. provided detailed analysis and comparison of Cassandra, VoltDB and Redis in their capability to scale and support application performance management tools [7]. They evaluated NoSQL data stores in the context of Enterprise Application Performance Management on virtual infrastructure [7]. Pirzadeh et al. developed range query dominant workloads on HBase, Cassandra and Voldemort [8]. The research by Pokluda et al. gives a detailed comparison of Cassandra and Voldemort for performance and scalability, focusing on failover characteristics for different throughputs [9].

Dede et al. examined the use of Cassandra over Hadoop, considering various features of Cassandra such as data partitioning and data replication which have a massive impact on the performance of Hadoop. They evaluated various replication factors on an eight-node cluster with a single consistency level [3]. As the number of geospatial applications has increased tremendously in recent years, research on geospatial data stores is being done actively. Agarwal et al. did extensive research to compare the performance of SQL databases with NoSQL databases when used in spatial datastores [1]. Baralis et al. also compared in their study the factors responsible for the difference in performance between SQL and NoSQL spatial databases [5]. Their review was able to show how NoSQL databases show better returns and do not cause failures on an increasing number of spatial points fetched. Duan et al. have researched extensively on ways to enhance MongoDB's spatial retrieval [4]. These researches assumed uniform data access pattern and the impact of data access patterns were not studied which is an essential factor to consider as demonstrated in the study by Haughian et al. in [3]. They showed a detailed process to Benchmark effects of replication for Cassandra and MongoDB in their research [3]. They address the impact of sharding and replication factors on NoSQL datastores and their scalability. There has been accelerated growth in the demand for performance evaluation and benchmarking of NoSQL databases. Querying geospatial data is more complicated because it has numerous types of objects and several operations. Most of the performance evaluation studies do not consider under the hood processing being done in NoSQL databases to fetch spatial data. NoSQL databases provide querying capabilities that extend beyond simple CRUD operations and evaluating outputs for multiple types of geoJSON objects in data warehouse require more computation than regular textual data reads. Most benchmark studies focus on intrinsic read, write, scan and update capabilities. In geospatial context, type of object being fetched such as a line, point or polygon can significantly impact performance. In this project, we will see how the kind of query affects the performance differently for different databases. We also did a scalability analysis of the same over the multinode cluster. To our best knowledge, the impact of data access patterns on the scalability of geodatabases is never studied before for spatial querying operations.

## 4. NoSQL Databases

### 4.1 Features of NoSQL databases

Traditional Relational Database Management Systems (RDBMS) do not have the capability required to process the variety and volume of data produced and consumed by organizations with rapid velocity. This has occurred due to the explosion of the rich generated content like web-based content or sensor/satellite data [11]. NoSQL databases are designed to run on clusters of commodity computers and are optimized for horizontal scalability. They support flexible schemas suitable for unstructured and semi-structured data that's why NoSQL databases are a proper fit for hierarchical geospatial data use by rapidly increasing number of GIS applications. CAP theorem delineates that NoSQL Databases can only offer any 2 out of Consistency, Availability and Partition Tolerance. This theorem is vital in the Big Data world, especially for making tradeoffs between the three, based on enterprise-specific use cases [16]. Therefore, benchmarking is most fruitful when done on the desired use case and dataset.

Adding more computing machines helps to scale to cater to a higher number of requests. Another critical factor to consider is that RDBMS follows Atomicity, Consistency, Isolation, and Durability (ACID) properties, which makes it increasingly tough to scale efficiently. NoSQL databases are exceedingly scalable and distributable compared to relational databases which were not designed to run on clusters. Most NoSQL databases are open-source and don't require expensive licenses. They can run on inexpensive hardware, rendering their deployment cost-effective. The main feature generally associated with NoSQL databases is the replication of data to provide continuous availability and location independence.

One more essential characteristic of NoSQL databases is that they do not require a schema. This property provides a huge advantage when dealing with complex and vague spatial data. A database schema is a description of all structures and elements in a database.

## 4.2 Sharding/Partitioning

NoSQL databases offer support for sharding or partitioning of data which involves partitioning database into smaller chunks which can be micromanaged. In the simplest form, it allows users to break a vast database into smaller partitions to spread across multiple instances. All NoSQL Databases delineate strategies for sharding data based on use cases to ensure efficiency for scaled distributed systems. Sharding plays a vital role in benchmarking geospatial databases as sharding data based on spatial properties is complex and is not supported by most databases. So, one must partition data on some other property, the choice of which can impact performance as it affects the distribution of data in a cluster.

## 4.3 Replication

An important feature provided by NoSQL databases to ensure high availability is the replication of data. The Replication Factor (RF) can be customized in NoSQL databases. It reflects several physical nodes that contain a replica of data or the number of times a set of data is replicated in the cluster. A request is valid when a certain number of replicas acknowledges the request. This number is termed the consistency level of the database, and it directly impacts the performance of the data warehouse. Lesser consistency level means lower latency while higher consistency level means the overhead of managing data among several nodes, thus higher latency. With this project, we chose master-slave (MongoDB) and peer-to-peer (Couchbase) replication and their impact on scalability for geodatabases.

## 4.4 MongoDB

MongoDB is a document NoSQL Datastore which provides an abundance of useful operations for geospatial querying. It started supporting spatial data from its version 2.4. Data documents are grouped as collections in the database. MongoDB provides brilliance in set storage, abstracting intricate requests, consistent performance, and supporting spatial retrieval. It provides consistency and partition tolerance in CAP theorem.

It offers horizontal scaling by splitting up data in multiple nodes through shards [4] — Sharded clusters provision deployments with big data sets and high-throughput operations. Sharding allows partitioning a collection within the database to dispense its documents across several mongod instances. Distribution of data and application traffic is done using a shard key in a sharded collection. Choosing an optimum shard key has substantial implications for performance. It can aid or avoid query isolation and increased write capability. MongoDB provides two types of sharding strategies: ranged sharding and hashed sharding. Hash-based sharding comprises calculating a hash value from the key field's value to distribute data among shards. Ranged sharding involves separating documents into ranges based on the shard key values. Query retrieval performance can improve in MongoDB by utilizing Indexes. Fields that frequently appear in queries should be Indexed. Indexes improve read performance. However, there is always an overhead to maintain an Index. Also, write performance can impact because the index update will happen with new writes.

A replica set in Mongo is a cluster of mongod instances that sustain identical data set. It consists of numerous documents bearing nodes and sometimes an arbiter node. One of the data bearing nodes in a replica set is a primary node. It is responsible for chronicling all changes done to its documents in its operation log (oplog), which is used by other nodes or secondary nodes to replicate changes in their documents. If the primary node becomes unavailable for any reason, any qualified secondary node will conduct an election to designate itself as the new primary node. Replication and sharding are commonly deployed together; however, they can be used independently as well. When replication and sharding are deployed together, the replication factor determines the number of replicas of each shard in a cluster. Primary node gets the requests for reads and writes. MongoDB offers supplementary read and writes configurations for replica sets. Write concern determines the level of acknowledgment for write operations necessitated from the database. Read preference determines the members of replica set the drivers should direct the read operations.

For quicker spatial retrieval MongoDB currently offers two indexes: 2D and 2DSphere Index. The 2D index calculates planar surface distances. 2D-sphere index calculates distances for geometries over spherical Earth. Reference coordinate model is presently limited to WGS84 [1]. MongoDB evaluates hash values for coordinate pairs and builds an index on top of these evaluated hashes.

We can store spatial data in MongoDB as geoJSON objects. We can define the geoJSON object by embedding the document with:

- 'type' field that specifies the Object type
- 'coordinates' field that defines the longitude and latitude coordinates. Longitude is listed before latitude.

For, Example

```
location: {type: "Point", coordinates: [-111.896465, 89.456464]}
```

System setup required to benchmark MongoDB for spatial queries is shown in figure 1.

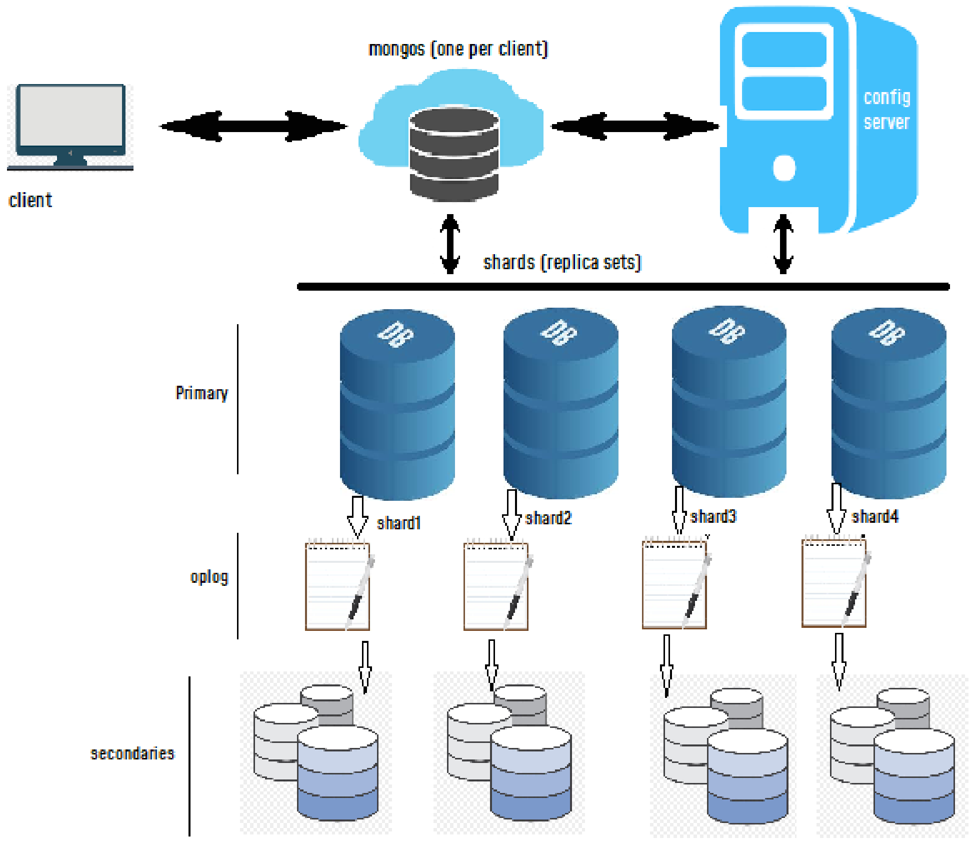


Figure 1. MongoDB architecture

We can use db.collection.createIndex() function to create a 2dsphere or 2d index on the geometry field. Field is specified with a string "2dsphere" or "2d" to set the index type.

```
db.collection.createIndex( { " location"  : "2dsphere" } )
```

MongoDB does not support spatial shard keys. Documentation of mongodb recommends creating a sharded collection by using some other field as shard key. Spatial operations that are offered for sharded collections are: $near, $nearSphere, $geoWithin and $geoIntersect.

TABLE 1. MONGO SPATIAL QUERY OPERATORS

| | |
|---|---|
| $geoIntersects | The operator to fetch geometries intersecting with another geoJSON geometry. A 2dsphere index supports $geoIntersects |
| $geoWithin | The operator to fetch geometries falling within a bounding geoJSON geometry. The 2d and 2dsphere index supports $geoWithin |
| $near | The operator to fetch geometries that exist in proximity to a point geoJSON geometry. The 2d and 2dsphere index supports $near |
| $nearSphere | The operator to fetch geometries that exist in proximity to a point geoJSON geometry over spherical earth. The 2d and 2dsphere index supports $nearSphere |

## 4.5 Couchbase

Couchbase is a peer-to-peer document store, developed over Erlang, a concurrency-oriented language. It provides availability and partition tolerance in CAP theorem. Indexing of 2-dimensional spatial data in Couchbase is based on R-tree. geoCouch was developed to manage spatial indexes and spatial data in Couchbase. It stores spatial data in the form of geoJSON and its query language use JavaScript using map-reduce. Data is stored in Buckets which is equivalent to mongo collections. Buckets can be partitioned into smaller chunks. We can partition documents present in Couchbase buckets into smaller virtual buckets (vBuckets) and specify global secondary Indexes (GSI) on partitions for faster retrieval.

For spatial querying, it offers spatial views that allow querying on datastore using bounding box (Bbox). Spatial views can access the geoJSON documents by computing complex geometries like points, line-strings, or polygons. Spatial views also create an index over arrays containing flat, multidimensional arrays of coordinates and geoJSON objects. We can create a spatial view in Couchbase using the emit() function. It takes key and value parameters as input, where the key must be a multi-dimensional Bbox. Bbox function is comparable to $geoWithin in Mongo. Couchbase provides Full-Text Search through which we can execute queries like $nearSphere in Mongo.

System architecture setup required to benchmark Couchbase for spatial queries is shown in figure 2.

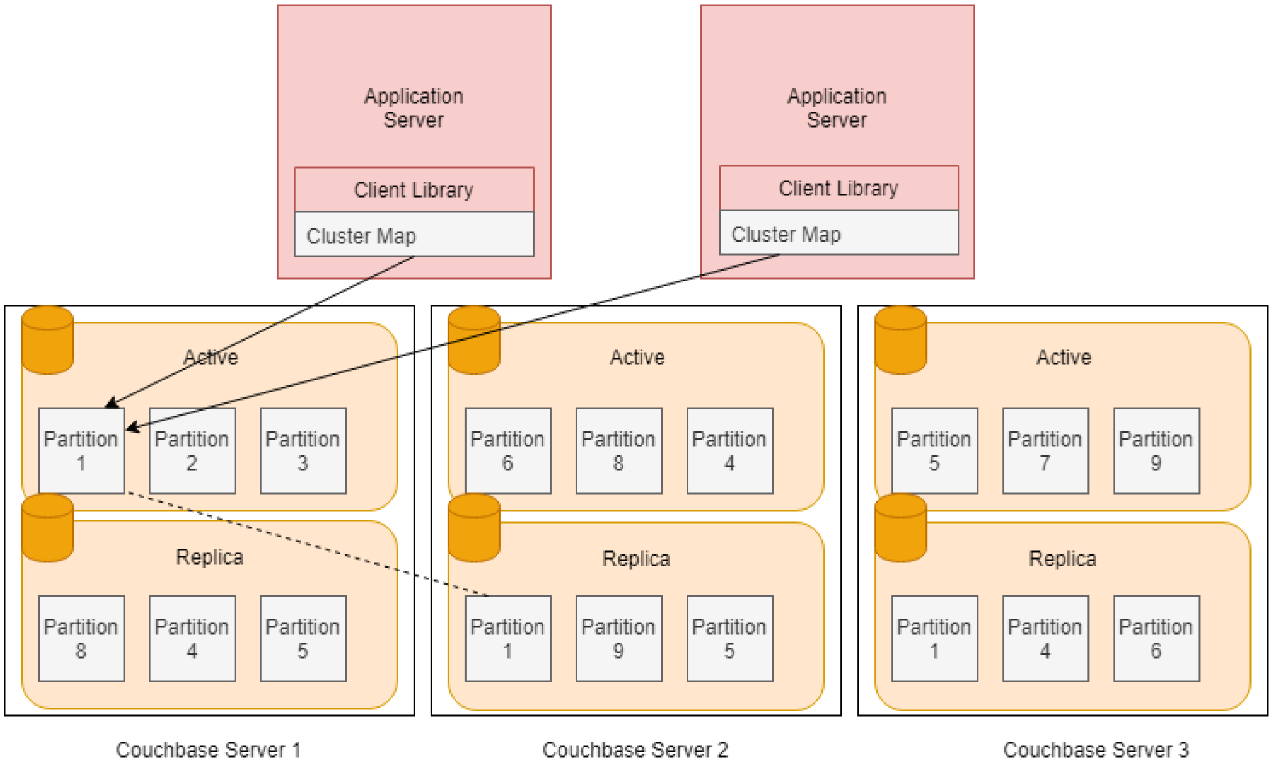


Figure 2. Couchbase architecture

The key for emit() function to define spatial Views can be defined in several ways out of which one is using geoJSON:

1. **Single values:** Key expands to a collapsed range when specified as one value. For example, the key [100, 200] is stored as [[100, 100], [200, 200]].
2. **Ranges:** Ranges can be used as a key to determining a range of values. For example, one can specify the open hours of an office (11:30 to 19:30) as emit([[1130, 1930]], null);

3. **GeoJSON geometry:** We can adopt a geoJSON geometry as a key. However, it must always be the initial element in the array. This prompts Couchbase to use bbox naturally to determine and use as a range. Couchbase supports the below geoJSON objects:
    a. Point
    b. MultiPoint
    c. LineString
    d. MultiLineString
    e. Polygon
    f. MultiPolygon
    g. GeometryCollection

Spatial views can be queried using the RestAPI on the client side.

A simple REST Call to Couchbase looks like:

```
GET /<bucket>/_design/<designDocument>/_spatial/<spatial-view>
```

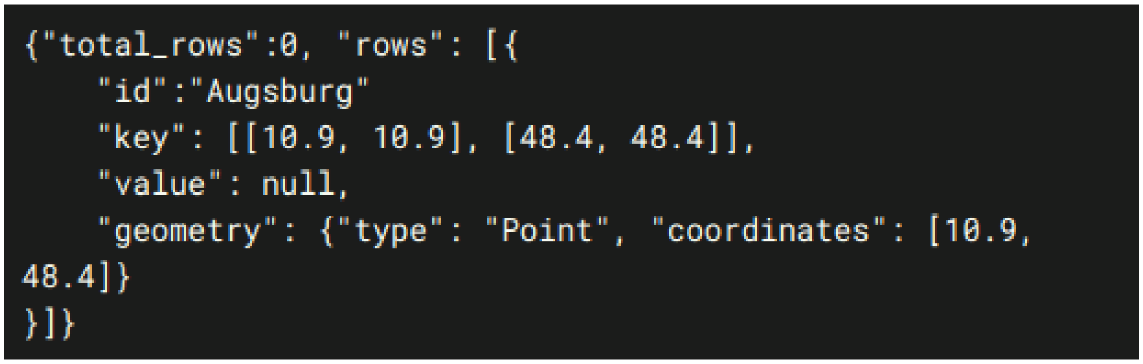
Some Query Parameters that one can use to query spatial data in Couchbase include:

TABLE 2. QUERY PARAMETERS IN COUCHBASE

| Parameters | Description | Type |
|---|---|---|
| start_range | Number of dimensions of the index must match the number of elements | An array of Numbers or null; Optional |
| end_range | Number of dimensions of the index must match the number of elements | An array of Numbers or null; Optional |
| limit | Number of documents that are returned will be limited to a specified amount | Number; Optional |
| bbox | Represents the bbox for spatial query | String; Optional |
| skip | A certain number of records, in the beginning, can be skipped using this parameter | Number; Optional |

| | | |
|---|---|---|
| Stale | Freshness level of documents can be defined with this property. Supported values: false, ok, and update_after | String; Optional |

Standard response as received when API call is made includes, id, key, value, geometry, and total.



```
{"total_rows":0, "rows": [{
    "id":"Augsburg"
    "key": [[10.9, 10.9], [48.4, 48.4]],
    "value": null,
    "geometry": {"type": "Point", "coordinates": [10.9,
48.4]}
}]}
```

Figure 3. The standard response from Couchbase spatial view

## 5. Benchmarking

In this section, we have proposed a benchmark architecture that can be deployed to benchmark a cluster of NoSQL database nodes. We have worked on the use case for graffiti abatements incident initiative, undertaken by the City of Tempe, to identify the best fit among MongoDB and Couchbase for their datastore. We have foresighted the types of spatial operations for the user from the developer's point of view and benchmarked all the operations separately. We have also described how MongoDB and Couchbase provide these operations.

Scalability benchmarking can be done by altering resource capacity when scaling horizontally, i.e., adding or removing more processing nodes to the system. Alternatively, we can also enhance system performance by vertical scaling in which the system capacity is increased by utilizing higher configuration hardware devices. System capacity or request workload can be varied between workload runs to evaluate scalability. We are going to measure throughput, average latency, and percentile latencies. Scalability benchmarking can be achieved by following two approaches [2]:

1. For a given system workload, vary cluster capacity between subsequent runs to measure throughput, average latencies, upper 95th percentile, and upper 99th percentile latencies.
2. For a given system capacity vary workload to the cluster between subsequent runs and then measure trends for throughput, average latencies, upper 95th percentile, and upper 99th percentile latencies.

Throughput is hugely affected by the choice of access distributions, replication factor, and consistency levels. Benchmark can be developed to evaluate the impact of these factors on the system performance. Performance improvement due to replication can be benchmarked by the following approaches:

1. For a given system workload and consistency level, vary the replication factor of a cluster and compare throughput with same-sized baseline non-replicated clusters.
2. For a given system workload and replication factor, analyze the impact of reading and writing operations by varying levels of consistency.

An important factor that can affect scalability performance for the large dataset are Data request distributions:

1. Uniform: Data documents are selected uniformly from the entire dataset with this access pattern. This pattern is used to represent applications in which any documents are selected independent of insertion time or popularity, e.g., blog posts.
2. Zipfian: Data documents are selected as per the popularity of data in datastore and independent of insertion time. Social media applications can be a good example of using Zipfian request distribution because popular users have more connections in social media applications.
3. Latest: In Latest request distribution data items are not selected based on popularity but selected as per the newest insertion time. This data pattern is used to represent apps in which recent insert time documents are desired, e.g., the news is popular at its time of release.

## 5.1 Dataset

It is vital that benchmarking and performance evaluation is done on a real-world dataset to ensure that the benchmark provides more holistic results. Spatial data is complex, and indexes on spatial data can speed up querying for GIS applications. For this project, we chose Graffiti Abatement Incidents dataset of the city of Tempe. It is publicly available on data.gov website for research purposes. City of Tempe public works department has an active graffiti abatement program. Hotspot regions are identified by government agents, police, lawmakers, secret service agencies or town planners to come up with policies and strategies to remove graffiti as soon as possible. This dataset is a perfect representation of an ideal spatial use case. Lawmakers and police might want to query data around a specific point representing center-based querying of data elements around a point. They may also want to fetch all points within a box or a polygon in the city. A town planner might want to remove graffiti on some road in the town and so it might query for Linear Intersect points. There are so many more such user operations which can be benchmarked in this dataset. The dataset is available in various spatial formats like ArcGIS, ESRI, CSV and most importantly, geoJSON which we have used in the project.

Some features of the dataset are listed below:

a. Source: catalog.data.gov
b. City: Tempe
c. Features: 13348
d. URL: https://catalog.data.gov/dataset/graffiti-abatement-incidents

A sample geoJSON document in the dataset is shown below:

```
{
  "type": "Feature",
  "properties": {
        "OBJECTID":1001,
        "INCIDENT_NUMBER":"1/26/2016-EC-1021",
        "LOCATION":"ROW",
        "NOTIFICATION": "Proactive Graffiti",
        "INCIDENT_DATE":"2016-01-26T21:33:24.000Z",
        "TAG_COUNT":0,
        "MONIKER_CLASS":"",
        "SQ_FT":0,
        "PROP_TYPE":"",
        "Waiver": null
  },
  "geometry": {
        "type": "Point",
        "coordinates": [ -111.89212050035984,33.40314749815397]
  }
}
```

## 5.2 Proposed Architecture

YCSB is a de facto benchmark for NoSQL database. However, YCSB currently does not support geospatial workload. To support spatial data and to create a benchmark relevant for spatial database selection and decision making, YCSB should be extended. YCSB populate data by inserting bits to the database using its core library and its drivers generate operations, such as insertion, reading, writing, updating, deleting, and scanning, to stress the target database system. While the basic textual data is well suited for this pattern of use, the situation gets complicated when dealing with hierarchical geospatial data. For Example, spatial operation involves fetching points within a specified distance or fetching geometries intersecting, crossing or overlapping another geometry. These queries are more complex since they require performing convoluted operations such as joins and aggregations on complex spatial data structure in the form of arrays and/or nested objects. We could take inspiration from a well-researched architecture like YCSB to develop our benchmark architecture which supports spatial operations. The design of our benchmark is to mirror the real-world situations that occur at the users' ends.

The proposed architecture of the benchmark is shown in figure 4:

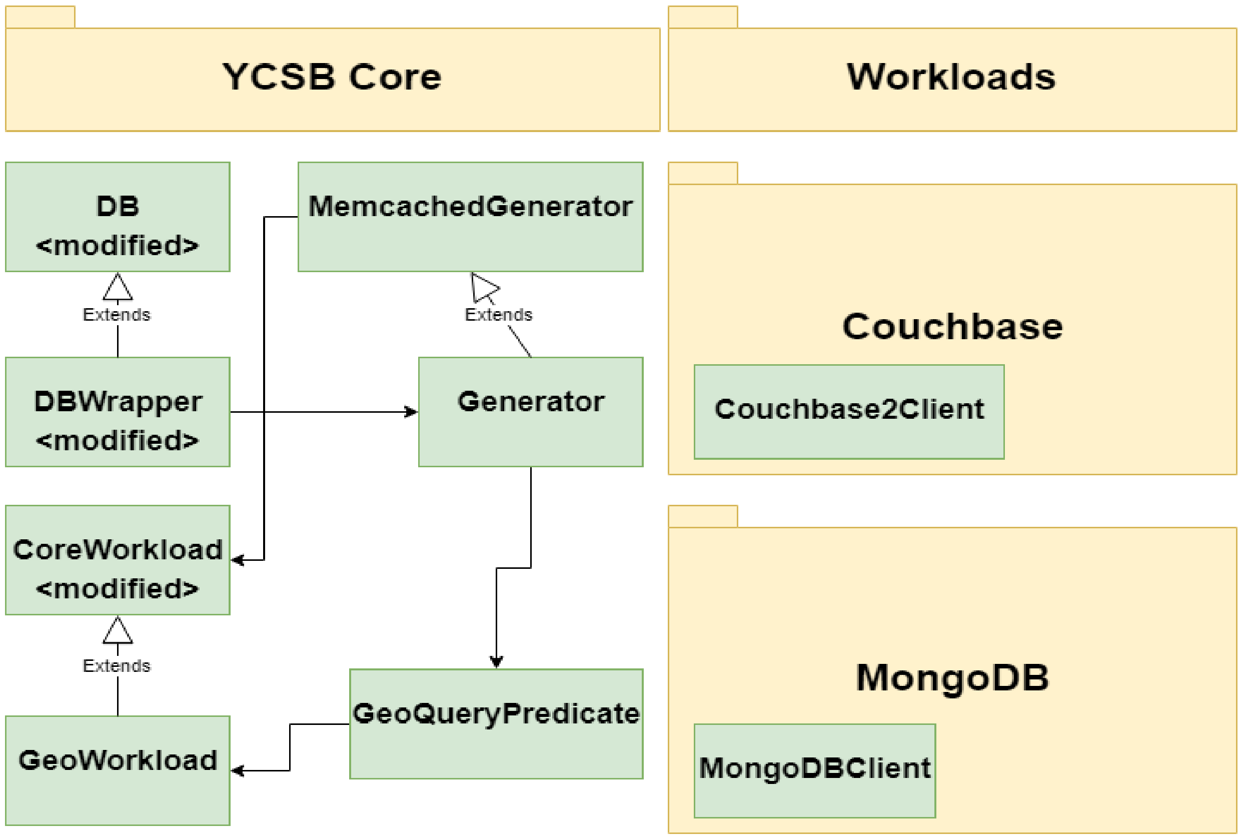


Figure 4. Proposed benchmark architecture

The proposed benchmark architecture consists of a core package and various client packages.

The core package consists of the following software components:

a. DB: This abstract class acts as a layer for accessing a database to be benchmarked. Every thread in clients is provided its instance of whatever DB class is to be used in the stress test. This class contains common Database properties which can be realized in some way from candidate databases.
b. DBWrapper: This class extends the DB abstract class and acts as a wrapper around a database to measure latencies, throughputs, and other benchmark results. Return codes are also evaluated in this class, and error codes are also generated here if some failure happens.
c. CoreWorkload: This is a public class that extends the Workload class of YCSB core package. It represents a set of the client doing simple Read, write, update or delete

operations. The relative ratio among several operations and other properties of the workload are controlled by parameters specified at run time. These parameters can be provided by a configuration text file of key/value pairs in Workload Package.

d. GeoWorkload: This class extends CoreWorkload class. We will specify spatial operations like the near, box, and intersects in this class. The relative proportion of various operations and other properties of the workload are controlled by parameters specified at run time.

e. MemcachedGenerator: The storage-based generator uses Memcached to fetch and store existing documents from databases for creating new random values to be used in benchmarking.

f. Generator: The storage-based generator uses Memcached. It loads data from cache for a benchmark to utilize. This approach allows YCSB to operate with real (or real-looking) JSON documents rather than random bits. It also provides the ability to query rich JSON documents by splitting JSON documents into predicates

g. GeoQueryPredicate: When we want to alter a certain embedded part of the document, we will use an instance of GeoQueryPredicate. It contains definitions for various parts of the dataset.

We developed client packages consisting of two classes, i.e., MongoDBClient and Couchbase2Client. Each class implements binding for the database with the benchmark. We will be executing queries using data generated by the Generator class. This ensures an apples-to-apples comparison between databases.

The workload package contains configurations for the database, which include the proportion of operations, request distribution method, document counts, operation counts, Memcached storage host, and ports, etc., all specified in a text file containing key/value pairs.

## 5.3 Benchmark Phases

The benchmark application that we developed executes in 4 phases as shown in figure 5.

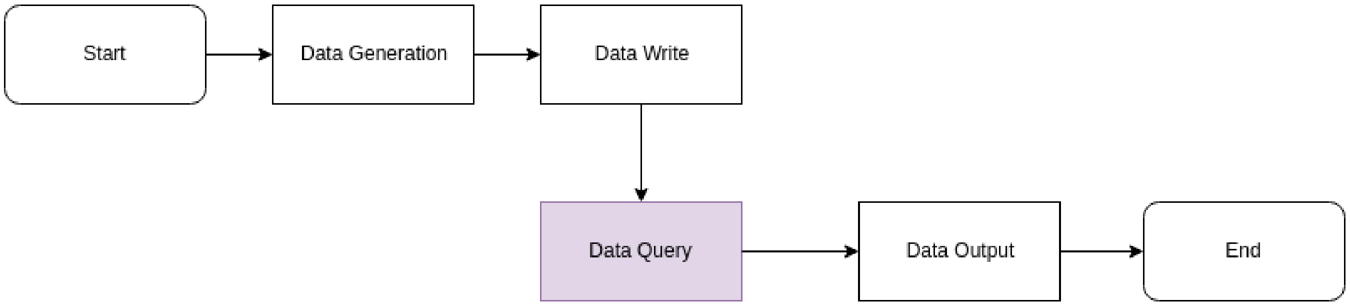


Figure 5. Benchmark process flow

- Data Generation phase–Spatial data is created by the application in the format relevant to datastore under stress.
- Data Write phase –Application generated data will be written to the datastore before stress testing for scalability to ensure the big size of data being benchmarked.
- Data Query phase – Data queries will be executed to stress the system with around 80% Memory usage and time consumed during this phase will be noted for benchmark results. The Benchmarking is done only in this phase.
- Data Output phase – Benchmark results will be processed in this phase to generate reports and project better analysis for decision makers.

## 5.4 Geospatial Queries Under Test

It is vital for a benchmark to provide an apples-to-apples comparison among the candidate databases. In our benchmarking, we identified and used representative geospatial operations commonly supported by both MongoDB and Couchbase. We expect a chosen geospatial operation of MongoDB to return the same result as the corresponding operation of Couchbase for the given input. This section describes the representative geospatial operations of our choice and explains how MongoDB and Couchbase support these operations.

### 5.4.1 Radius Based Queries (GeoNear or Near Query)

A near query or a radius-based query is a geospatial query that defines a geometry that outputs every document confined in a proximity match. Geometry is given as latitude-longitude coordinate pairs. With this spatial operation, the user expects the output of documents in order of near to farther from a specified geometry. This type of querying is supported both by MongoDB as well as Couchbase.

MongoDB postulates this type of query by utilizing 2dsphere indexing. Geospatial query operator $nearSphere describes a geometry over spherical earth, with output documents from nearest to farthest based on distance. Distances are evaluated for nearSphere using spherical geometry [13]. We can choose any of the following indexes based on our dataset.

- Geometry data specified as geoJSON points need a 2dsphere index.
- Geometry data specified as legacy coordinate pairs need a 2d index.

The query in MongoDB that implements radius-based querying is as below:

```
db.incidents.find({
        location: {
                $nearSphere: {
                $geometry: { type: "Point", coordinates: [-2.235143, 53.482358] },
                $maxDistance: 100
        }
}})
```

Couchbase also provides radius-based queries by evaluating geo_distance. The geometries can be fetched by spatial query specifying a longitude-latitude pair and a distance. The geometry of query fixes a circle center whose radius length is the defined distance. Couchbase returns all documents within this circle. A geospatial query in Couchbase is considered fruitful if it references index that achieves geopoint mapping type to field consisting the coordinate pair. This can be done easily on Couchbase UI web-console, or the index can be defined by requesting the Rest API.

The query in Couchbase that returns same documents as above-mentioned MongoDB radius-based query:

```
{
        "from":0,
        "query":{
        "location": {    "lon":-2.235143,
                         "lat":53.482358
                },"distance":"100m",
                "field":"geometry"
        },"sort":[{"by":"geo_distance",
                "field":"geometry",
                "unit":"m",
                "location":{
                        "lon":-2.235143,
                        "lat":53.482358
                }

        }
]}
```

The query has a "sort" document that defines that the outputs should be sorted in terms of geo_distance from defined longitude and latitude coordinates [14].

### 5.4.2 Box-Based Query (GeoBox or Box Query)

The geometry data denoted in a spatial query can be two latitude-longitude coordinate pair. These two pairs can be taken to express top-left and bottom right corners of bbox. Documents are outputted when they refer to geometry confined within the specified bounding box.

MongoDB defines a geospatial query operator $box which states a rectangle for a geospatial within a query ($geoWithin) to output documents that fall within the boundary of the specified rectangle, as per their point location data. When the $box is used with $geoWithin, the output returned are documents based on coordinates. GeoJSON shapes such as line or polygon are not queried by $geoWithin when the $box is used [13]. Box spatial query evaluates distances using planar geometry. Geospatial index is not essential for the applications to use the $box; however, spatial indexes provide better throughput than unindexed equivalents [13]. The $box is supported only using 2d spatial index.

The query in MongoDB that implements box-based querying is following:

```
db.incidents.aggregate([{
        geometry: {
                $geoWithin: {
                        $box: [ [-2.235143, 53.482358], [ 28.955043, 40.991862 ] ]
                }
        }
}])
```

Couchbase also supports box-based queries. The below query consists of top_left property which defines the top left corner of the bounding box. Here we show the use of arrays to define coordinates. It is vital to remember that longitudes precede latitude in the array. The bottom right corner of the rectangular bounding box is defined as a key-value pair. If the location falls within the specified box formed using top-left and bottom right corner, then the document corresponding to location is retrieved by the query. The outputs can be defined to be ordered.

The query in Couchbase that return identical documents to above-mentioned mongo box-based query is following:

```
{
        "from":0,
        "query":{
        "top_left":[-2.235143,53.482358],
        "bottom_right":{
                "lon":28.955043,
                "lat":40.991862
        },
        "field":"geometry"
},      "sort": ["properties.OBJECTID"] }
```

### 5.4.3 Querying on Intersects (GeoIntersect or Intersects Query)

Querying on Intersects involve querying for documents whose spatial data intersects with the specified geoJSON data object. This means that if there is an intersection of location, then the fetched geometry is not-empty.

MongoDB defines $geoIntersects operator that makes use of $geometry operator to specify geoJSON object. We can state multipolygons or polygons by employing the default coordinate reference System (CRS). Spherical geometry is used to evaluate intersects in Mongo. Just like the $box, geospatial Index is not a requirement for the $geoIntersects. However, having a spatial index enhances the performance of the query [13]. Only 2dsphere spatial indexing supports geoIntersects.

The query in mongodb that implements intersect based querying is as follows:

```
db.places.find({ geometry: {
        $geoIntersects: {
                $geometry: {
                        type: "MultiLineString",
                        coordinates: [ [ [100.0, 0.0], [101.0, 1.0]] [ [102.0, 2.0], [103.0, 3.0]]]
}}}})
```

Querying on intersects in Couchbase is achieved by querying spatial views. GeoJSON input documents are JSON documents consisting of geographic regions of several kinds. A geospatial view function can be designed to create a multidimensional view for geoJSON polygons. We can query based on a geometry and a location. One can create a three-dimensional view function. The first two dimensions specify the geometry and third dimension is the location. Queries only filter on a bounding box level.

For example, a spatial development view can be designed in the Couchbase as follows

```
function(doc, meta) {
    if (doc.geometry && doc.properties
    && doc.properties.location) {
    emit (doc.geometry, doc.properties.location);
    }}
```

## 5.5 Setup for the experiments

Experiments were conducted on AWS instances. This section describes the system configuration for our benchmarking.

- Couchbase experiments were conducted using Couchbase Server Edition 5.1.0

- MongoDB experiments were conducted using MongoDB Community Edition version 3.4
- For the cluster setup, we used EC2 m4-large instances with two vCPU, 8 GB of memory and "moderate" network throughput to deploy the Couchbase cluster. All instances were provisioned within the same AWS rack and data-center.

Table 3 summarizes the system configuration used for our benchmarking.

TABLE 3. DEVELOPMENT SETUP

| **Setting** | **Value** |
|---|---|
| OS | Ubuntu 18.04.1 |
| Word Length | 64-bit |
| RAM | 62.6 GB |
| Hard Disk | 2 TB |
| CPU Speed | 2.80 GHz |
| Cores | 8 |
| Ethernet | gigabit |
| Additional Kernel settings | A time disabled |

The type of workloads that we used for tests are the following:

- Workload A consists of 100% of geospatial querying operations. Each workload run targets one of the geospatial queries.
- Workload B consists of 80% of near operations, and 20% write operations.
- Workload C is chosen to simulate a real-world workload. It consists of 30% near queries, 25% box queries, 25% intersect queries, and 20% spatial writes.

One can customize workloads in the benchmark by simply creating a new workload configuration file that consists of a key-value pair, the type of operations. Here, the workload C configuration file is presented as an example.

```
recordcount=1000000
```

```
operationcount=1000000
workload=com.yahoo.ycsb.workloads.geo.GeoWorkload
geo_insert=0.20
geo_update=0
geo_near=0.30
geo_box=0.25
geo_intersect=0.25
geo_scan=0
geo_request_distribution=latest
geo_storage_host=localhost
geo_storage_port=11211
geo_querylimit_min = 10
geo_querylimit_max = 10
geo_offset_min = 10
geo_offset_max = 10
```

## 6. Experiments and Results

We chose the following benchmark execution flow to ensure accurate benchmarking:

1. Load the data in the candidate databases. We created grafittiDB collection for MongoDB and a corresponding bucket in Couchbase. The data is loaded in the databases using external scripts.
2. Execute the load phase for Benchmark. During this phase, Benchmark loads a set of documents from the database and store all its values in internal Memcached to use while benchmarking. In this phase, we also increased the number of records in the database to 1 million records by loading data in the database by making small coordinate changes in the cached data. This increment is to make data size large enough to represent big data.
3. Execute the run phase for Benchmark with each prepared workload one at a time. During this phase, the benchmark generates queries and executes them against the database. The throughput and latencies are measured during this phase.

The benchmarking results are the values of the following performance metrics. The results can be studied and analyzed to make a more effective decision about the selection of NoSQL data store for geospatial applications.

- Total throughput of candidate database for the current workload
- Total run time for the current workload.
- Average latency distribution separate for the various spatial operations.
- Maximum latency and Minimum latency distribution for the various spatial operations.
- Upper 99th percentile and Upper 95th percentile latency measure of spread for several spatial operations.

## 6.1 Experiments on Single Node

### 6.1.1 Workload A

In this experiment, we evaluated the performance of representative spatial queries (near, box and Intersect queries) for the candidate databases. This workload consists of 100% of a specific query operation in a way that we can exclude the impact of a write operation in our performance analysis. This experiment is to compare the performance of different spatial operations for the given database and identify the performance difference between two candidate databases for the given query. Throughput and latencies are the performance metrics used in this experiment. For our experiment we fixed the number of nodes as one for both the databases and executed the benchmark with the following configurations:

Record Count: 1000000 Operation Count: 1000000 Request Distribution: Uniform

Figures 6 and 7 present the experiment results.

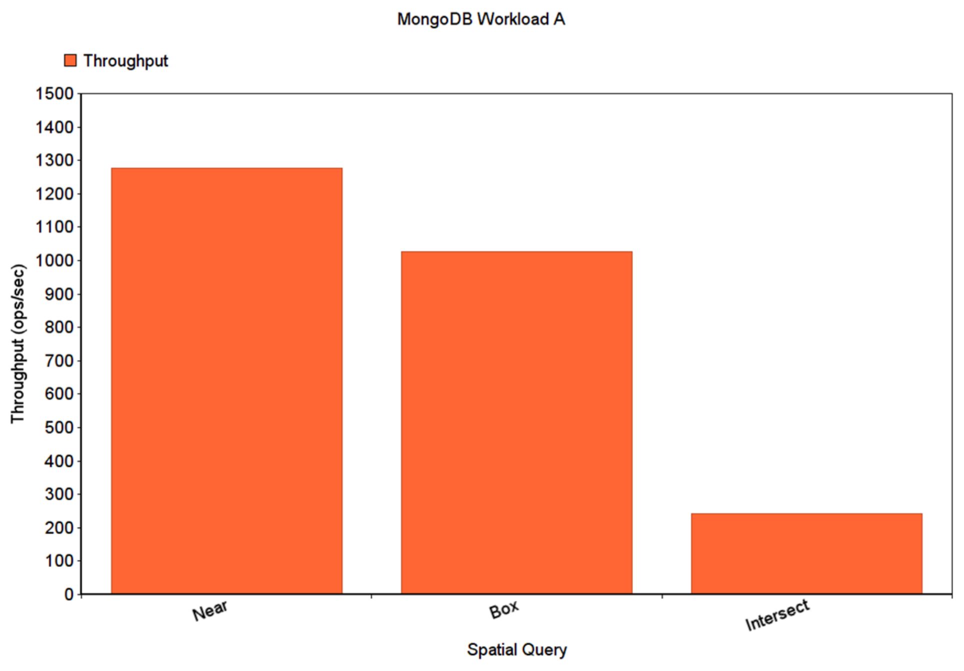


Figure 6. MongoDB workload A throughputs for near, box and intersect

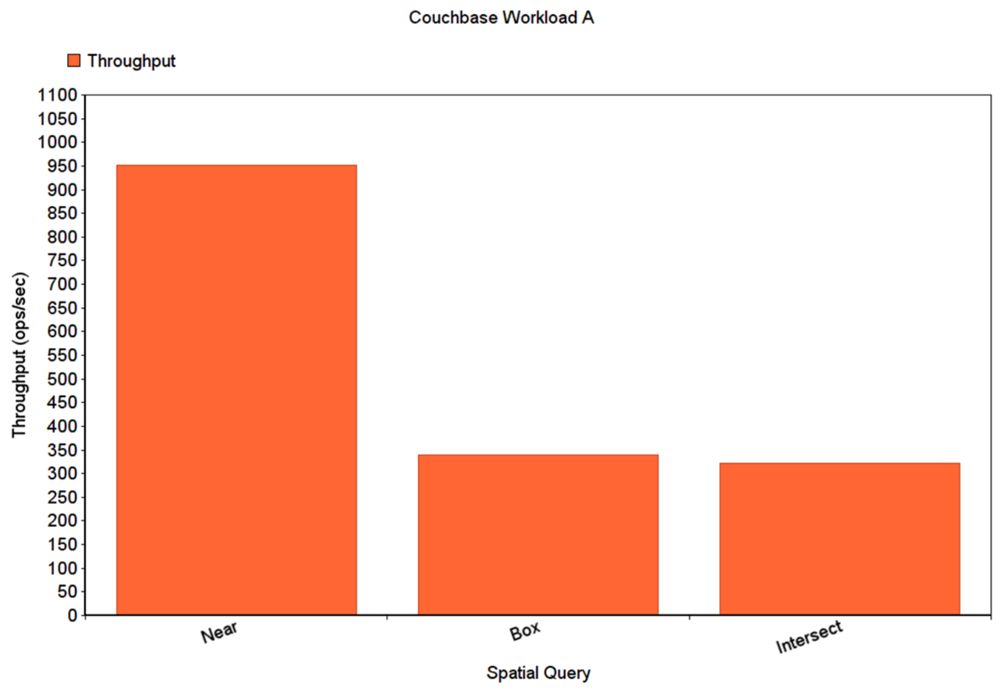


Figure 7. Couchbase workload A throughputs for near, box and intersect

Results in figures 6 and 7 show that MongoDB throughputs of chosen geospatial queries are higher than Couchbase throughputs. The results also show that near query executes considerably faster than the intersect query in both databases. We expected this result because intersects require considerably more calculations by the database as compared to the radius-based queries. Figure 6 also shows that the box query of MongoDB performs better than its intersect query. This difference in throughput is because MongoDB evaluates distances for the box using planar geometries and distances for the intersects using spherical geometries. Thus, we observe that the planar distance calculation requires lesser computation by the database. In MongoDB, the throughput difference between the near and the box queries is 19.5% However, the corresponding throughput difference in Couchbase is 64.4% as shown in figures 6 and 7. Couchbase uses spatial views for box queries and full-text search based spatial index for the near query. The results indicate that the use of full-text search used in the near query provides faster results in Couchbase than spatial-view based box query. It is also interesting to note that Couchbase throughputs for intersect query are higher than MongoDB throughputs. This shows that the R-tree index provides higher throughput for reads than MongoDB's B-tree based spatial index. Also, we observe similar throughputs for box and intersect operations in Couchbase which can be explained because spatial views support both operations. In a single server system, we expect that latency is reciprocal of throughput because parallelism does not come into the picture. The orange bars; representing average latency, of the figures 8 and 9 mirror our expectation. Our performance benchmark also measures upper 95th percentile (represented in green in figures 8 and 9) and upper 99th percentile (represented in blue in figures 8 and 9) latencies. The upper 95th percentile latency and upper 99th percentile latency, for intersects, when compared with box and near operations, is almost three times for MongoDB. This difference is very less when observed for Couchbase in figure 9, where the difference between box and Intersect is just 5%. Again, this is because both box-based queries and Intersect based queries in Couchbase are supported by spatial views.

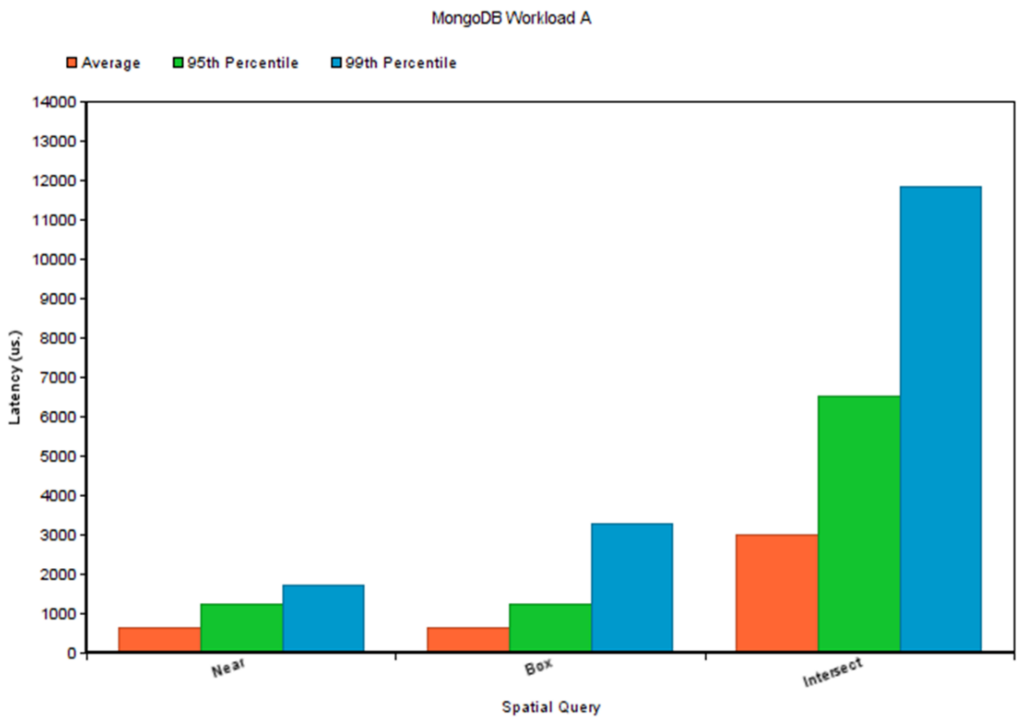


Figure 8. MongoDB workload A latencies for near, box and intersect

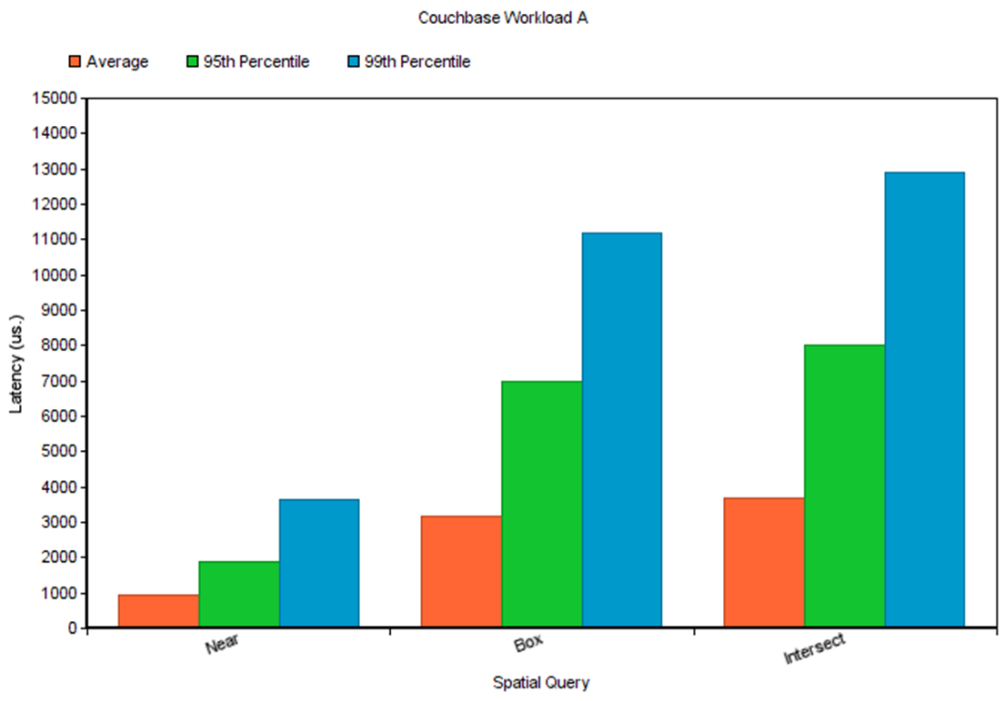


Figure 9. Couchbase workload A latencies for near, box and intersect

### 6.1.2. Workload B

In this experiment, we investigated the performance impact of various types of spatial queries (near, box and intersect queries) when executed with spatial writes. Writes cause the size of workload to increase when executing. Thus, the size of the data becomes variable. This workload consists of 80% of geospatial read operations and 20% of geospatial write operations. For our experiment we fixed the number of nodes as one for both the databases and executed the benchmark with the following configurations:

Record Count: 1000000 Operation Count: 1000000 Request Distribution: Uniform

Figures 10 and 11 present the throughputs measured in this experiment.

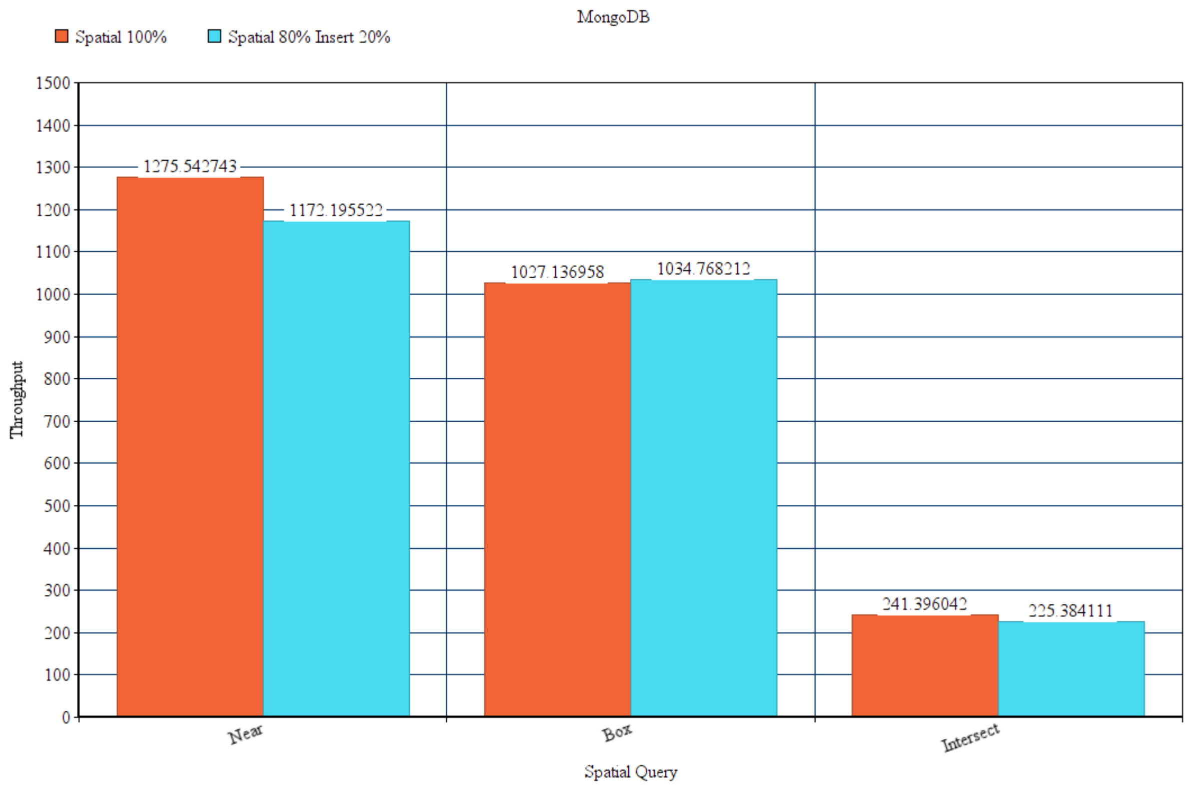


Figure 10. MongoDB effect of spatial writes on querying

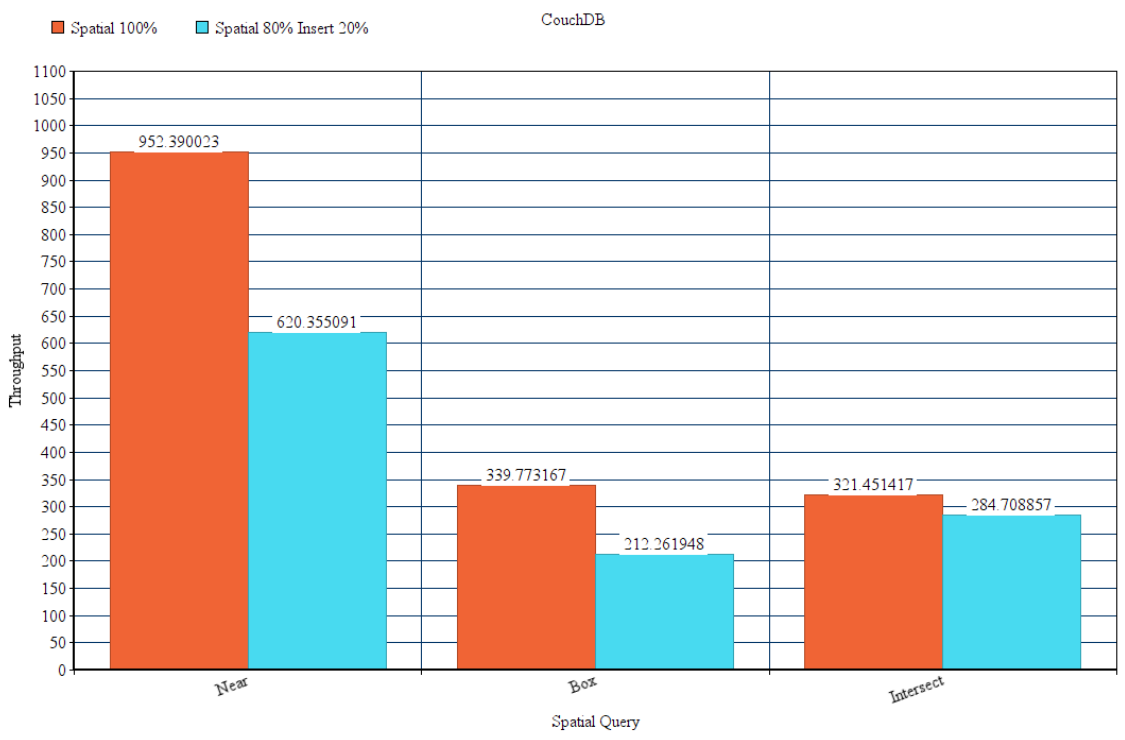


Figure 11. Couchbase effect of spatial writes on querying

In MongoDB, the throughput of near is degraded as compared to workload A (consisting of 100% queries) while that of the box and intersect query remains about the same. The impact of writes is not seen for MongoDB box and intersect queries. The results indicate that spatial read for box and intersect requires more computation as the database size increases by write operations. In Couchbase, the throughput degradation is significant for near, box, and intersect. We observe that the impact of write is more predominant in Couchbase as compared to MongoDB. This can be explained because the R-tree index supports Couchbase which is known to be very efficient for reads, however, it has considerable overhead for write operations due to splitting an overflowing node in the R-tree. Another important factor to consider that lowers throughput is compaction. Couchbase performs compaction of data when new data is written to the buckets to ensure optimum storage and better performance for reads. For compaction, Couchbase generates a new file to write the non-stale information. At the same time, existing database documents stay in place and are continually used for storing information [14]. Compaction, as well as writes/updates, are a CPU intensive operation. If one is running low on

RAM, one might want to reduce compaction because Couchbase may block writes temporarily to make sure that it is not accepting writes faster than it can clean and merge redundant data. Benchmarking experiment was conducted on default auto-compaction. We can control the process of compaction in Couchbase to trigger on off-peak hours.

### 6.2 Experiments on Multiple Nodes

Horizontal scalability benchmarking is done by increasing the number of nodes to serve the same amount of data. Both MongoDB and Couchbase offer replication. We replicated data across multiple nodes and studied the impact of replication factor on the performance. When both sharding and replication are deployed, each shard is replicated in a replica set consisting of primary and secondary. MongoDB shards the documents of a collection by a shard key of choice. Similarly, in Couchbase, a bucket is partitioned by hash, based on the key of choice, and each partition is replicated by the replication factor times across multiple nodes. In this section, we discuss some of the results from the experiments on workload A, workload B, and workload C.

#### 6.2.1 Workload A

Size of replica set (or replication factor) used for our experiments is three. Selection of optimal shard key and sharding strategy is critical for performance. MongoDB currently does not support a geospatial shard key. According to the recommendation from a MongoDB documentation [17], we had to choose a non-spatial field as a shard key to shard geospatial collections. For the experiments, we can use the ObjectID field as a shard key because it has many different values and good cardinality. It is also monotonically increasing which makes it an ideal candidate for hash-based sharding.

Data distribution based on hashed values facilitates more even data distribution. Couchbase Server implements the technique to spread partitions of data around to take benefit of multiple nodes by a structure called vBuckets. They are functional equivalents of database shards. Data is partitioned using the constant hash value of the key into vbuckets which are stored on the data nodes. Couchbase also implements a technique to spread the index load by index partitioning. We can break the index among several nodes with hashing creating several secondary indexes [18].

We executed this workload by specifying 100% geospatial read operations in the configuration file. The performance metrics evaluated are throughput and average latency. For our experiment we varied the number of nodes and request distribution for both the databases and executed the benchmark with the following configurations:

Record Count: 1000000 Operation Count: 1000000

The experiment results are shown in figures 12 and 13.

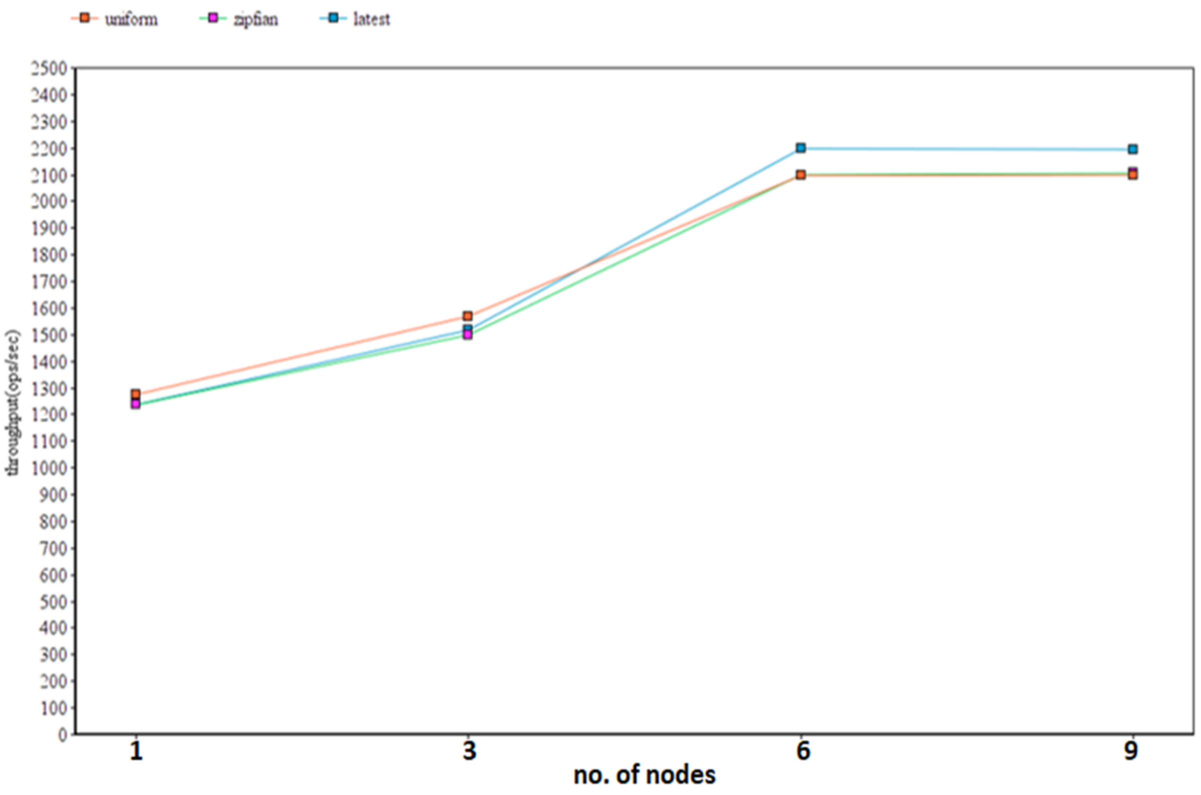


Figure 12. MongoDB workload A scalability and access distribution curve

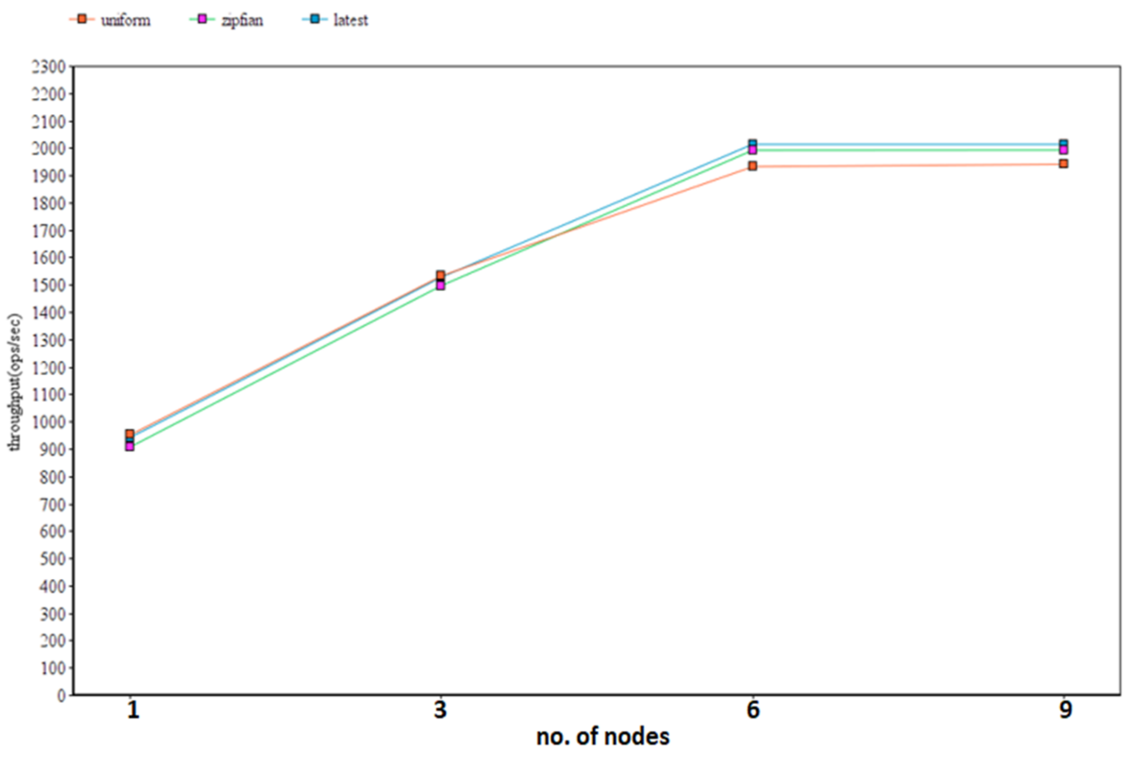


Figure 13. Couchbase workload A scalability and access distribution curve

We observe that the throughput increase as cluster size increases for both MongoDB and Couchbase. When we apply replication to every shard in the multi-node cluster, there are a greater number of shards to serve the requests in parallel. The same trend is observed in Couchbase as well.

It is interesting to note that when the number of nodes increases from one to three in Couchbase, the throughput of MongoDB does not increase as steeply because with MongoDB all reads are sent to the primary node (the node that houses a primary shard). Thus, the degree of parallelism expected by adding nodes is not achieved as effectively as in Couchbase. We used the default read preference of MongoDB which directs all reads to the primary node.

We also observe that when node size is high (i.e., six nodes or nine nodes) the difference between throughputs is less among MongoDB and Couchbase, than what we observed with a fewer node cluster. Thus, Couchbase shows promising performance growth when scaling.

Although MongoDB still shows slightly higher throughputs than Couchbase for 100% spatial read scenario.

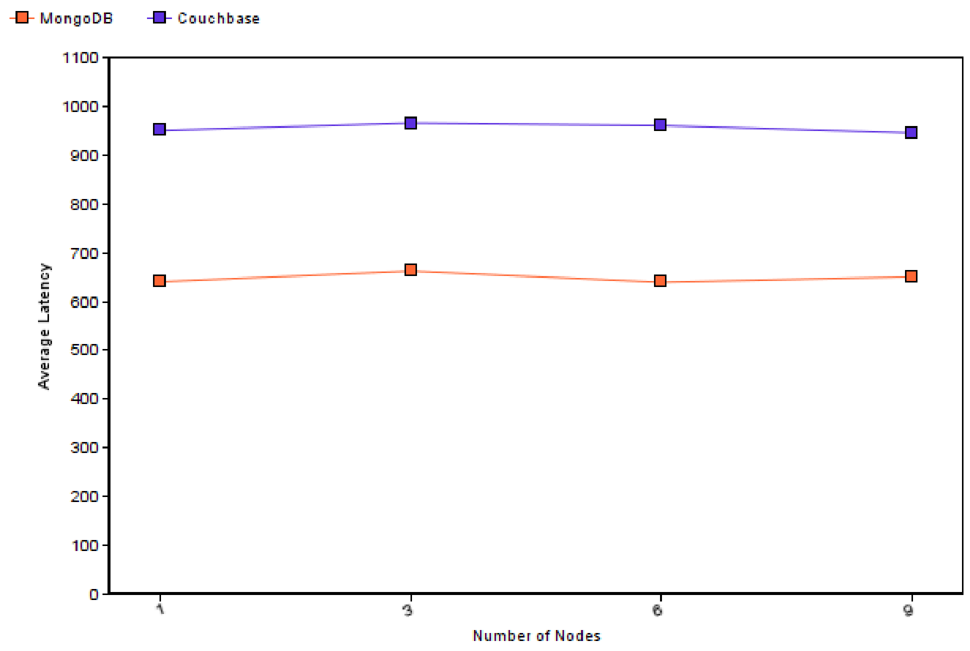


Figure 14. Effect on latency due to replication

We also varied the number of nodes and fixed the uniform request distribution for both the databases and executed the benchmark to observe the average latency distribution. The results of the experiment are shown in figure 14. We can see that when replication is involved, the latency is not reciprocal to throughputs. In contrast to a non-replicated system, we expect that latency is reciprocal of throughput because of the lack of parallelism.

## 6.2.2 Workload B

We desired to examine the result of spatial writes on reads when we scale data stores horizontally. To accomplish this, we used workload in which we specified 80% geospatial read operations and 20% geospatial write operations in the configuration file. The shard configuration and replication used is the same as the one used with workload A, i.e., we applied hashed sharding on ObjectID property, and we use a replication factor of three. For our experiment we varied the number of nodes and request distribution for both the databases and executed the benchmark with the following configurations:

Record Count: 1000000          Operation Count: 1000000

We present the results of this experiment results in figures 15 and 16:

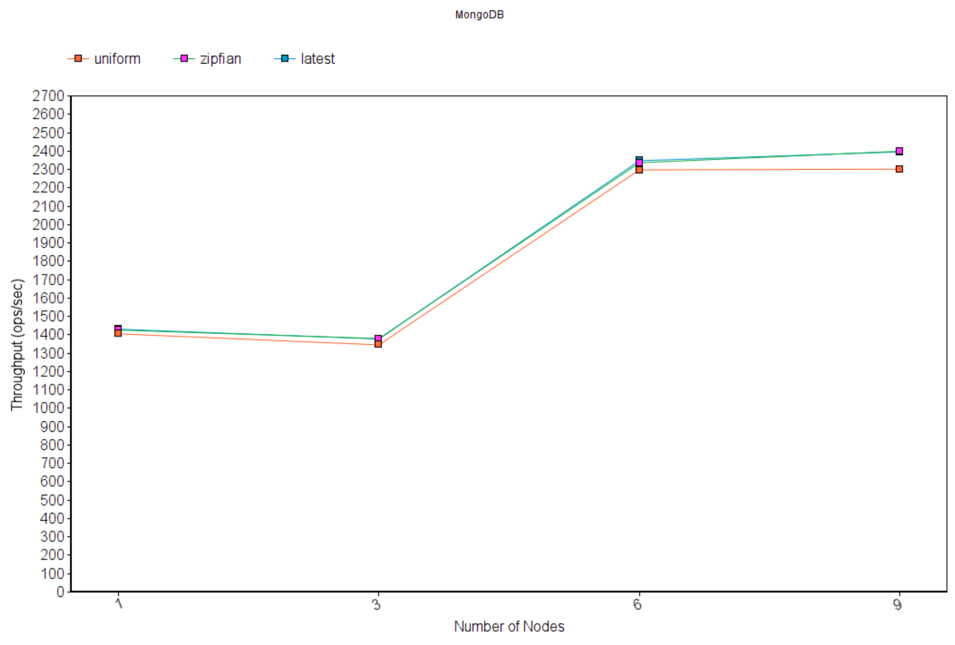


Figure 15. MongoDB workload B scalability and access distribution curve

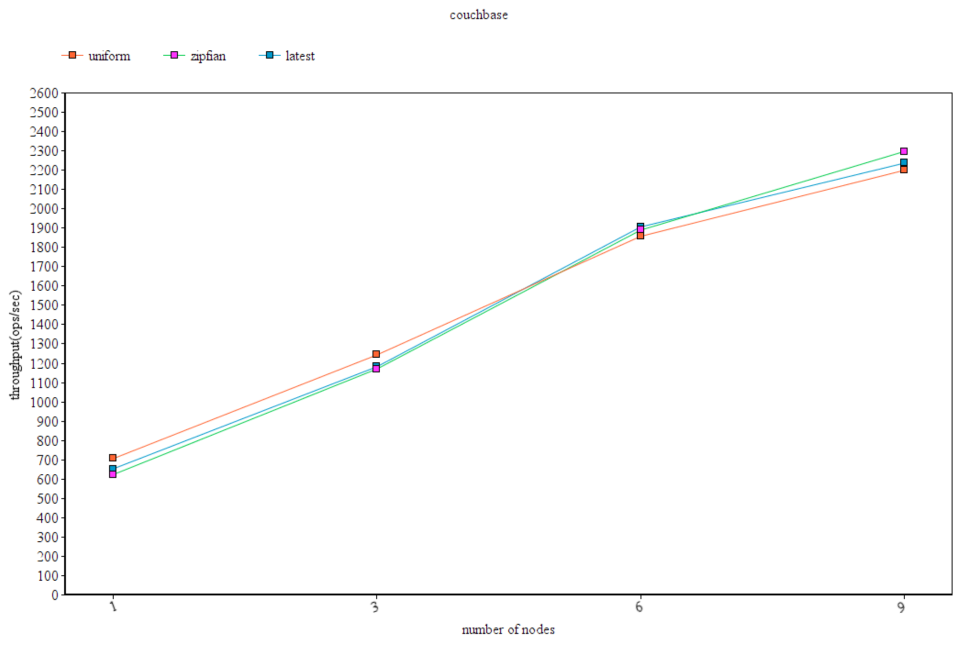


Figure 16. Couchbase workload B scalability and access distribution curve

We observe a throughput drop in MongoDB from cluster sizes one to three. We do not observe this drop with Couchbase because of the master-slave architecture of MongoDB where writes are sent to the primary node only. When we apply the replication factor to every shard in a 3-node cluster, it triggers the primary node to store data on oplog to manage two added secondaries. As the number of nodes is increased further to six node and nine nodes, load for querying and writing to the database is distributed on more shards and thus throughput increases following the regular trend. The Latest and Zipfian distributions slightly outperform the Uniform distribution throughout for MongoDB. We anticipate this because of MongoDB which stores complete data on disk and loads data to RAM when needed. Latest and Zipfian would perform better as accessed data would still be in RAM after few operations.

### 6.2.3 Workload C

Finally, we wanted to benchmark a real-life scenario with the benchmark. To achieve this, we used workload in which we specified 30% near operations, 25% box operation, and 25% intersect operations, and 20% spatial write operations. The shard configuration and replication used is the same as the one used with workload A and B, i.e., we applied hashed sharding on ObjectID property, and we used a replication factor of three. For our experiment we varied the number of nodes and request distribution for both the databases and executed the benchmark with the following configurations:

Record Count: 1000000 Operation Count: 1000000

The experiment results are shown in figures 17 and 18.

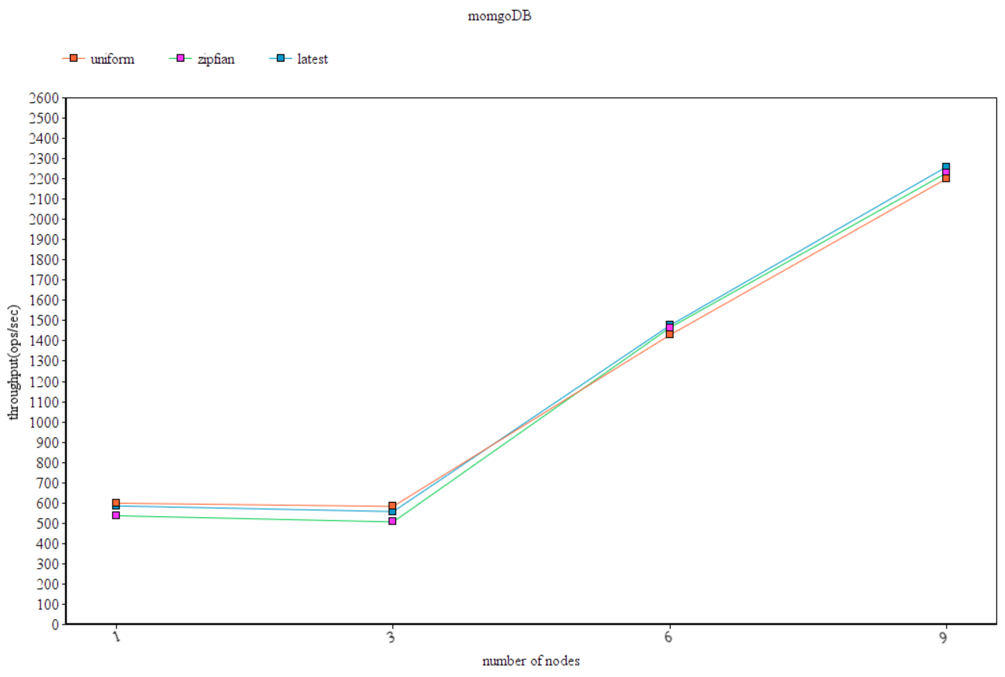


Figure 17. MongoDB workload C scalability and access distribution curve

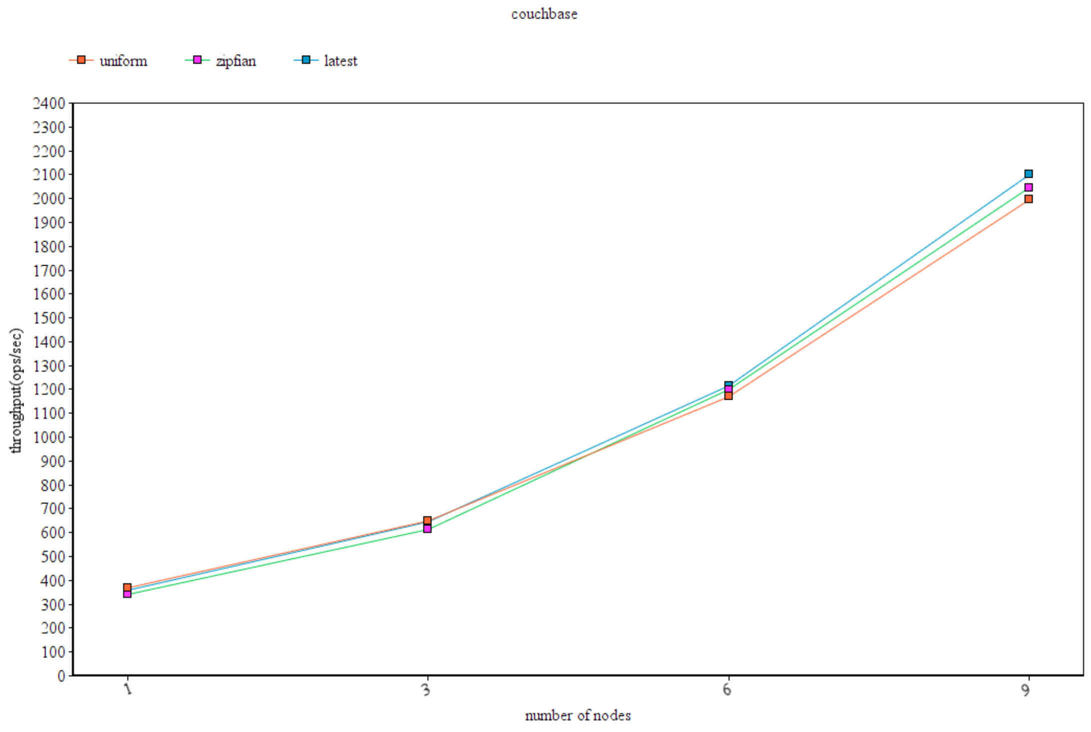


Figure 18. Couchbase workload C scalability and access distribution curve

With this experiment as well, we observe that MongoDB shows slightly higher throughputs than Couchbase, and we observe a throughput drop in MongoDB from cluster sizes one to three which we do not observe in Couchbase. In the master-slave architecture of MongoDB, write operations always direct only to the primary node. When applying the replication factor to every shard in a three-node cluster, MongoDB triggers the primary node to store data on oplog to manage two added secondaries. As the number of nodes is increased further to six nodes and nine nodes, load for querying and writing to the database is distributed on more shards and thus throughput increases following the regular trend. The Latest and Zipfian distributions slightly outperform the uniform distribution for MongoDB as well as for Couchbase as we expected, because complete data is loaded to RAM only when needed and accessed data would still be in RAM after few operations.

## 6.2.4 Query Runtime Analysis

In this experiment, we examined the net run time (in ms.) to fetch a certain number of points from databases.

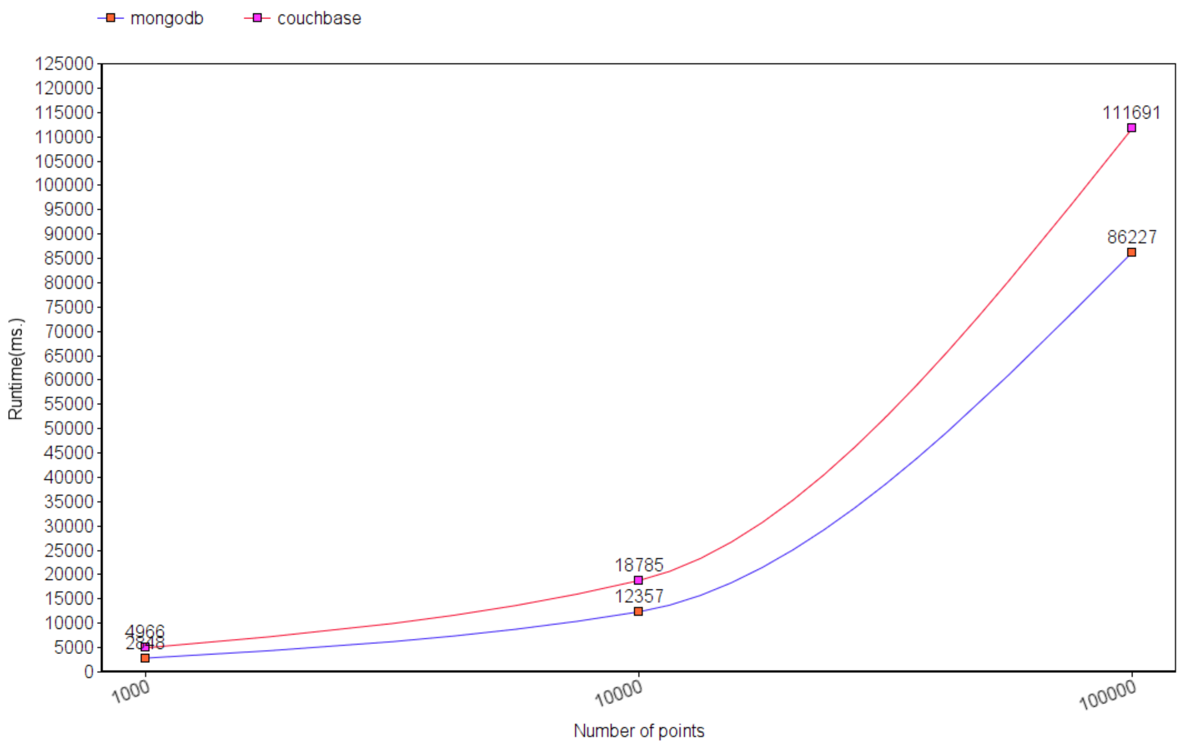


Figure 19. MongoDB vs. Couchbase runtime comparison for variable points

We varied the number of points fetched by the query in a 3-node cluster having hashed sharding deployed on ObjectID property, and we used a replication factor of three. The databases were stressed to examine that queries return without failure when fetching a certain number of points.

From figure 19, we observe a lower latency with MongoDB as compared to Couchbase, as it was observed with the previous experiments as well. Both the system demonstrated a stable retrieval performance, so both are comparable data stores in terms of retrieval efficiency of geospatial queries. Even on increasing further the number of points fetched, the two systems did not give any sign of breakdown. As we can see in figure 19, the runtime is monotonically increasing for both MongoDB and Couchbase.

## 7. Conclusions and Future Work

In this project, we have developed a spatial benchmark, its architecture and integrated it with YCSB to provide geospatial support to it. This benchmark provides decision makers with the tools to perform a comparative analysis of spatial NoSQL data stores. We demonstrated a step-by-step process to benchmark graffiti abatement use case. We have used the benchmark to analyze the performance of MongoDB and Couchbase on the representative geospatial queries.

By following the proposed architecture, we were quickly able to get conclusive results to show which database performs better under specified conditions. We can set and tune parameters to perform a comprehensive scalability analysis of geospatial data stores. The outcomes of the experiments conducted using the developed benchmark helps us to:

- Get a comprehensive qualitative comparison based on features of considered datastores.
- Acquire an empirical comparison of average run times, throughputs, and latencies of candidate datastores based on a large campaign of experiments.
- Show that both Couchbase and MongoDB scale nearly linearly.
- Demonstrate that MongoDB performs slightly better than Couchbase for the spatial operations for our use-case.

We plan to extend the benchmark to support more geospatial data stores. All we need to do is develop clients for respective databases which will interact with the existing benchmark. We can also provide support for more parameters like consistency levels in the benchmark workload configurations to enable more detailed and easier analysis for scalability. With the current, we must set consistency levels at the database level. We also plan to extend experiments with larger datasets on larger Amazon EC2 instances with greater configurations making use of solid-state disks to reflect real-world production deployments better.

## References

[1] Agarwal, Sarthak and Rajan, KS (2017) "Analyzing the performance of NoSQL vs. SQL databases for Spatial and Aggregate queries," Free and Open Source Software for Geospatial (FOSS4G) Conference Proceedings: Vol. 17, Article 4. DOI: https://doi.org/10.7275/R5736P26.

[2] Jörn Kuhlenkamp, Markus Klems, and Oliver Röss. 2014. Benchmarking scalability and elasticity of distributed database systems. Proc. VLDB Endow. 7, 12 (August 2014), 1219-1230. DOI: http://dx.doi.org/10.14778/2732977.2732995.

[3] Haughian G., Osman R., Knottenbelt W.J. (2016) Benchmarking Replication in Cassandra and MongoDB NoSQL Datastores. In: Hartmann S., Ma H. (eds) Database and Expert Systems Applications. DEXA 2016. Lecture Notes in Computer Science, vol 9828. Springer, Cham. DOI: https://doi.org/10.1007/978-3-319-44406-2_12.

[4] Duan, Miaoran, and Gang Chen. "Assessment of MongoDB's spatial retrieval performance." Geoinformatics, 2015 23rd International Conference on. IEEE, 2015.

[5] Baralis, E., Dalla Valle, A., Garza, P., Rossi, C., & Scullino, F. (2017, December). SQL versus NoSQL databases for geospatial applications. In Big Data (Big Data), 2017 IEEE International Conference on (pp. 3388-3397). IEEE.

[6] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., & Sears, R. (2010, June). Benchmarking cloud serving systems with YCSB. In Proceedings of the 1st ACM symposium on Cloud computing (pp. 143-154). ACM.

[7] T. Rabl, S. G ́omez-Villamor, M. Sadoghi, V. Munt ́es-Mulero, H.-A. Jacobsen, and S. Mankovskii. Solving big data challenges for enterprise application performance management. Proceedings of the VLDB Endowment, 5(12):1724–1735, 2012.

[8] P. Pirzadeh, J. Tatemura, and H. Hacigumus. Performance evaluation of range queries in key-value stores. In 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Ph.D. Forum (IPDPSW), pages 1092–1101. IEEE, 2011.

[9] A. Pokluda and W. Sun. Benchmarking failover characteristics of large-scale data storage applications: Cassandra and Voldemort.

[10] E. Dede, B. Sendir, P. Kuzlu, J. Hartog, and M. Govindaraju. An evaluation of Cassandra for Hadoop. In 2013 IEEE Sixth International Conference on Cloud Computing (CLOUD), pages 494–501. IEEE, 2013.

[11] E. Hewitt. Cassandra: the definitive guide. O'Reilly Media Inc., 2010.

[12] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica. Probabilistically bounded staleness for practical partial quorums. Proc. VLDB Endow., 5(8):776–787, Apr. 2012.

[13] "Welcome to the MongoDB Docs", [Online]. Available: https://docs.mongodb.com. [Accessed March 25, 2019].

[14] "Welcome to Couchbase Docs", [Online]. Available: https://docs.couchbase.com/. [Accessed March 25, 2019].

[15] López, Marco & Couturier, Stéphane & Lopez, J. (2016). Integration of NoSQL Databases for Analyzing Spatial Information in Geographic Information System. 2016 8th

International Conference on Computational Intelligence and Communication Networks (CICN). 351-355. 10.1109/CICN.2016.75.

[16] "CAP Theorem and Distributed Database Management Systems", [Online]. Available: https://towardsdatascience.com/cap-theorem-and-distributed-database-management-systems-5c2be977950e. [Accessed March 30, 2019].

[17] "Geospatial Queries", [Online]. Available: https://docs.mongodb.com/manual/geospatial -queries/. [Accessed March 30, 2019]

[18] "Divide and Conquer: Couchbase GSI Index partitioning", [Online]. Available: https:// blog.couchbase.com/couchbase-gsi-index-partitioning/. [Accessed March 30, 2019]

[19] "NoSQL Databases: What Geospatial Users Need to Know", [Online]. Available: https:// www.directionsmag.com/article/2045/. [Accessed March 30, 2019]