2-28-2023

GeoYCSB: A Benchmark Framework for the Performance and Scalability Evaluation of Geospatial NoSQL Databases

Suneuy Kim
San Jose State University, suneuy.kim@sjsu.edu

Yvonne Hoang
San Jose State University

Tsz Ting Yu
San Jose State University

Yuvraj Singh Kanwar
San Jose State University

Recommended Citation

Suneuy Kim, Yvonne Hoang, Tsz Ting Yu, and Yuvraj Singh Kanwar. "GeoYCSB: A Benchmark Framework for the Performance and Scalability Evaluation of Geospatial NoSQL Databases" *Big Data Research* (2023). https://doi.org/10.1016/j.bdr.2023.100368

Big Data Research 31 (2023) 100368

Contents lists available at ScienceDirect

Big Data Research

journal homepage: www.elsevier.com/locate/bdr

GeoYCSB: A Benchmark Framework for the Performance and Scalability Evaluation of Geospatial NoSQL Databases

Suneuy Kim *, Yvonne Hoang, Tsz Ting Yu, Yuvraj Singh Kanwar

Department of Computer Science, San José State University, San Jose, CA 95192, USA

ARTICLE INFO

Article history:
Received 1 June 2020
Received in revised form 4 October 2022
Accepted 3 January 2023
Available online 11 January 2023

Keywords:
Geospatial Big Data
NoSQL
Benchmark
MongoDB
Couchbase
Apache Accumulo

ABSTRACT

The proliferation of geospatial applications has tremendously increased the variety, velocity, and volume of spatial data that data stores have to manage. Traditional relational databases reveal limitations in handling such big geospatial data, mainly due to their rigid schema requirements and limited scalability. Numerous NoSQL databases have emerged and actively serve as alternative data stores for big spatial data.

This study presents a framework, called GeoYCSB, developed for benchmarking NoSQL databases with geospatial workloads. To develop GeoYCSB, we extend YCSB, a de facto benchmark framework for NoSQL systems, by integrating into its design architecture the new components necessary to support geospatial workloads. GeoYCSB supports both microbenchmarks and macrobenchmarks and facilitates the use of real datasets in both. It is extensible to evaluate any NoSQL database, provided they support spatial queries, using geospatial workloads performed on datasets of any geometric complexity. We use GeoYCSB to benchmark two leading document stores, MongoDB and Couchbase, and present the experimental results and analysis. Finally, we demonstrate the extensibility of GeoYCSB by including a new dataset consisting of complex geometries and using it to benchmark a system with a wide variety of geospatial queries: Apache Accumulo, a wide-column store, with the GeoMesa framework applied on top.

1. Introduction

Geospatial data – data associated with a location on Earth – is created and captured in large volumes from numerous sources every day. "A significant portion of big data is geospatial data, and the size of such data is growing rapidly at least by 20% every year" [1]. The rapidly increasing number of location-based services and applications are developed to deliver the full benefits of geospatial data to industry, business, and the general public. Traditionally, spatial data is housed in relational databases. However, relational databases encounter challenges when managing big geospatial data, mainly due to their rigid schema requirements and limited scalability. Therefore, there has been a constant search for innovative data management solutions for geospatial data.

NoSQL (Not-Only-SQL) databases came into the picture as alternative data stores that exchange some of the guarantees and functionality of relational databases for higher performance, especially when working with big data. NoSQL databases provide a flexible data model that accommodates big data for the needs of modern applications. NoSQL databases are designed to run on clusters of commodity computers and are optimized for horizontal scalability, which is an essential system requirement for big data management. Recently, NoSQL databases for geospatial data has emerged as an active area of research [2–7], and a considerable number of geospatial applications are now operating with NoSQL databases in the back-end. Foursquare (MongoDB), SimpleGeo (Cassandra), PokemonGO (Couchbase), Google Earth (Google Big Table), and Scrabbly (MongoDB) are just a few example applications using NoSQL databases to manage spatial data.

Identifying the most performant data store for an application entails evaluating the performance and scalability of different systems for the application use cases. Benchmarking is an evaluation methodology to examine a system with respect to performance metrics (e.g., throughput and latency), system parameters (e.g., the number of CPU cores, RAM size, and disk size), and workload parameters (e.g., query density in the workload mix and distribution of data access) in a cost-effective manner. A well-tuned benchmark allows developers to make an unbiased decision on an efficient and scalable database for a given application in the early phases of its development. Benchmarks can also be used to discover the performance bottlenecks of systems under diverse workloads.

* Corresponding author.
E-mail address: suneuy.kim@sjsu.edu (S. Kim).

https://doi.org/10.1016/j.bdr.2023.100368

In this study, we develop a benchmark framework, GeoYCSB, for the performance and scalability evaluation of NoSQL databases for geospatial workloads. We extend YCSB (Yahoo! Cloud Serving Benchmark) [8] to design GeoYCSB. YCSB is the most prominent NoSQL database benchmark used by big data enterprises. Development of YCSB has promoted active research in the NoSQL benchmarking field [9–11]. However, YCSB currently does not support geospatial workloads. To our knowledge, there is no prior work using YCSB for geospatial workloads. The task of extending YCSB to support spatial workloads is not just about implementing a new workload. We develop an architecture over the current implementation of YCSB by integrating new components and then implement geospatial workloads using them. GeoYCSB supports both microbenchmarks and macrobenchmarks and facilitates the use of real datasets in both. A microbenchmark is to test the efficiency of basic spatial operations in isolation. A macrobenchmark consists of a series of logically related operations that describe an application's use case and thus reflect the access pattern typical of the application.

This paper extends our previous publication [12] which presents the development of GeoYCSB and its microbenchmarks for geospatial workloads. The specific extension to the former publication is the development of GeoYCSB macrobenchmarks, which allow NoSQL databases to be tested for more realistic use cases relevant to the application. This paper now presents GeoYCSB as a comprehensive benchmark framework that can support both microbenchmarks and macrobenchmarks. Another extension is our demonstration of the extensibility of GeoYCSB by including a new dataset consisting of complex geometries and using it to benchmark a system with a wide variety of geospatial queries.

Using GeoYCSB, we conduct benchmark experiments and present the analysis of their results. The purpose of the presented experiments is twofold - (1) to demonstrate the capability of GeoYCSB to evaluate and compare the performance of NoSQL systems. We evaluate two leading document stores, MongoDB and Couchbase, and make apples-to-apples comparisons. These document stores were chosen for this study because both databases support geospatial queries, GeoJSON data types, and spatial indexes. Also, both support comparable sharding and replication strategies, which allows us to conduct comparisons of MongoDB and Couchbase at equivalent levels. (2) to demonstrate GeoYCSB's extensibility for new datasets, various NoSQL data models, and complex geospatial operations. We test MongoDB with new datasets. Then, we choose Accumulo, a wide-column data store, as a different data store and evaluate it with workloads consisting of complex geospatial queries. The use of Accumulo is for the extensibility test, and thus, we did not conduct a performance comparison between Accumulo and another NoSQL system.

Our contributions are the following.

- We develop GeoYCSB, an extensible benchmark framework that can handle geospatial workloads, by integrating new components into the design architecture of YCSB.
- We develop a microbenchmark consisting of geospatial queries that are parameterized by a real dataset, specifically, the Graffiti Abatement Incidents dataset of the city of Tempe. Microbenchmarking experiments are conducted to evaluate the performance and scalability of MongoDB and Couchbase under geospatial workloads.
- We develop a macrobenchmark based on four common use cases for graffiti abatement applications. These use cases comprise of geospatial queries that are also parameterized by real datasets, namely, the Graffiti Abatement Incidents, Building Footprints, and Tempe Public Schools datasets. Macrobenchmarking experiments are conducted to evaluate the performance and scalability of MongoDB in the context of the applications' use cases.
- We examine the impact of a tunable consistency level on the performance and scalability of multiple clusters by deploying both replication and sharding.
- We present and analyze experimental results to identify factors that affect the performance of geospatial queries and systems.
- We demonstrate the extensibility of GeoYCSB by including a new dataset, specifically, a combination of the Japan's Counties dataset and the Japan's Routes dataset, consisting of complex geometries that we then use to benchmark a system with a wide variety of geospatial queries: Apache Accumulo, a wide-column store, with the GeoMesa framework applied on top.

The remainder of this paper is organized as follows. We describe the main features of MongoDB and Couchbase, focusing on geospatial supports, sharding, and replication in Section 2. GeoYCSB is introduced in Section 3. GeoYCSB microbenchmark and macrobenchmark are presented in Sections 4 and 5, respectively, along with the benchmarking experiments and our analysis. In Section 6, we demonstrate the extensibility of GeoYCSB. Related work is reviewed in Section 7. Finally, we present conclusions and future work in Section 8.

2. NoSQL databases under performance comparison test

This section presents the main features of MongoDB and Couchbase, focusing on their geospatial supports and distributed data management using sharding and replication.

2.1. MongoDB

A MongoDB database is a set of collections, each of which consists of JSON-like documents. A shard key for a collection is a field or two that MongoDB uses to partition data. A chunk is a continuous range of shard key values, and documents are mapped to chunks according to their shard key values. MongoDB stores a chunk (i.e., documents mapped to the same chunk) in an available server. A chunk should split if its size grows beyond a specified threshold. The balancer migrates chunks around, evenly distributing them among nodes in the cluster. MongoDB ensures that documents with the same shard key values stay together in the same chunk to minimize the amount of probing for a given query. Hashed sharding and range-based sharding are representative sharding strategies offered by MongoDB. The hashed sharding strategy uses hashed shard key values while the range-based sharding strategy uses shard key values as they are for the chunk ranges. The shard key greatly affects the performance and efficiency of sharding. It is hard to change the shard key after sharding is deployed, and thus, one should carefully choose it. A shard key must be indexed, and the shard key index performs a crucial role in improving query performance as the data size grows. MongoDB config servers store the list of chunks on every shard and the ranges that define the chunks. A mongos instance, which is usually per application server, routes application requests to correct shards by consulting config servers.

MongoDB replication provides redundancy and increases data availability. MongoDB replicates data in a replica set. A replica set consists of one primary server and multiple secondary servers that replicate the primary's data by replaying the operation log (oplog) of the primary. The primary can serve both read and write operations while secondaries only serve read operations. Therefore, if the primary goes down, the cluster becomes unavailable for writes until a new primary is elected among the secondaries. Reading from a replica can introduce data inconsistency if the recent write

is not copied to the replica for the read operation. By default, read operations are sent to the primary. An application can specify a read preference to send read operations to secondaries. An application can also specify read and write concerns for read and write operations, respectively, to set the consistency level of the operation. The consistency level is tunable so that applications can trade off consistency for latency.

MongoDB stores geospatial data as GeoJSON objects or legacy coordinate pairs. A geospatial query uses one of the geospatial selectors $near/$nearSphere, $geoWithin, or $geoIntersects, along with a geometry specifier such as $box or $geometry, to specify target data to manipulate [13]. MongoDB offers 2dsphere and 2d indexes for geospatial queries. The 2dsphere index is for spherical earth maps and supports both GeoJSON objects and legacy coordinate pairs. For a 2dsphere index, MongoDB partitions the earth's surface and structures them in a B-tree [14]. The 2d index is for flat maps and supports legacy coordinate pairs. To create a 2d index, MongoDB evaluates hash values for coordinate pairs and builds an index on top of these evaluated hashes.

2.2. Couchbase

As a descendant of Membase and Apache CouchDB, Couchbase is both a key-value store and a document store. The caching layer of Couchbase follows the Membase distributed key-value data model. Its persistence layer largely inherits the capabilities from CouchDB, a JSON document store, for storage, indexing, and querying. A key-value data store treats their values as opaque BLOBs, supporting fast key-value lookups. Document stores are aware of the internal structure of documents, which allows indexing and querying the contents of the documents. Logically related documents are stored in a bucket that corresponds to a collection in MongoDB.

Couchbase provides two different ways of querying geospatial data: Full-Text Search (FTS) [15] and Spatial Views [16]. Couchbase N1QL (a SQL-like-language of Couchbase) may emulate geospatial queries. However, it is currently not supported by the geospatial index and can cause significant query performance degradation, especially in dealing with complex geospatial queries. Therefore, we do not address N1QL in this study. Also, in Couchbase Server 6.0, spatial views are no longer supported. This decision indicates that geospatial queries and indexes provided by FTS are a more efficient choice than those of spatial views. Our experimental results support this presumption. We find that with a workload consisting of 10% within queries, the use of FTS for the within query results in higher throughput (875 ops/sec) than that of a spatial view (340 ops/sec), with 88% difference.

Couchbase FTS uses Bleve, an open-source search and indexing library written in Go, for indexing of documents. Bleve supports various types of queries, including geospatial queries. It stores indexes in a single key-value store table where it handles index keys and values as byte arrays. A spatial view defines a MapReduce function in JavaScript. A spatial view produces a spatial index based on the spatial content of JSON documents of the bucket to which the view belongs. Spatial views store spatial indexes in R-trees. In both the FTS and spatial view approaches, a spatial index can be built upon a geospatial field of any GeoJSON type, including points, multi-points, linestrings, polygons, and geometry collections.

Couchbase also supports sharding and replication. Each bucket is logically partitioned into 1,024 vBuckets (virtual buckets). Like MongoDB chunks, Couchbase evenly distributes vBuckets across nodes in the cluster. The hashed document ID is used to assign a document into a vBucket. An application server maintains a cluster map, which stores the bindings of vBuckets to nodes in the cluster. vBucket is the unit of sharding and replication in Couchbase.

Data replication in Couchbase is mainly for resilience. When a node fails, another node carrying its replica helps the recovery of the failed node. Couchbase supports up to three replicas, making up one active data and up to three replicas. An application reads and writes from active nodes. Each write operation can specify a consistency level using the optional parameter replicated_to. Couchbase has to wait for acknowledges from the specified number of replicas to notify the client that the write is successful. Replicas cannot serve writes and serve reads when active data is not available before failover takes place. Couchbase ensures that active data and its replicas are distributed over different nodes.

3. GeoYCSB

This section describes the design architecture of GeoYCSB, the representative geospatial queries used to define GeoYCSB benchmark workloads, and the primary performance metric used in the benchmarking experiments.

3.1. Design architecture of GeoYCSB

YCSB is an open-source extensible benchmark framework for cloud serving systems [8]. It comes with a workload generator, known as the YCSB client, and a set of core workloads to evaluate the performance of various database systems running on clusters. To implement a new workload, one can modify the parameter file to tune the specific workload and/or write a new workload Java class. While YCSB has served as a de facto benchmark framework for NoSQL database systems, it currently does not support geospatial features.

The task of extending YCSB to support spatial workloads is not just about implementing a new workload. Another layer needs to be added to the YCSB design architecture to define a base class representing a geospatial database interface. The client class that implements binding to a specific spatial NoSQL database can extend this class to implement the common geospatial operations in their dialect. Then, a new geospatial workload class, which describes an experiment scenario with the spatial operations defined in the base class, is implemented. In this way, GeoYCSB can preserve the extensibility of YCSB while supporting geospatial workloads. Also, the geospatial queries that comprise the workloads should be parameterized with spatial data. YCSB perceives data as key-value pairs and populates data in the database by generating random bits for the values. The queries, which comprise workloads, will read and write a value as opaque flat data without being aware of its internal structure. This pattern of perceiving data does not work for geospatial queries. To be answered, geospatial queries inherently rely on the content of values. For example, spatial operations involve fetching points within a specified distance or fetching geometries intersecting, crossing, or overlapping another geometry. Without awareness of the value contents, executing such operations as well as measuring corresponding execution times will be meaningless. We adopt the idea of using real data for benchmark operations from [17] and load real spatial data to an in-memory data structure to feed geospatial queries with real spatial parameters.

Fig. 1 depicts the design architecture of GeoYCSB in UML (Unified Modeling Language). In Fig. 1, GeoDB, GeoDBWrapper, GeoWorkload, ParameterGenerator, DataFilter and SpatialDBClient are the new components we added to YCSB to make up GeoYCSB. For brevity purposes, we describe these classes only. The description of the complete set of GeoYCSB classes can be found in [12].

- The GeoDB class represents a geospatial database interface layer. It abstracts representative geospatial operations includ-

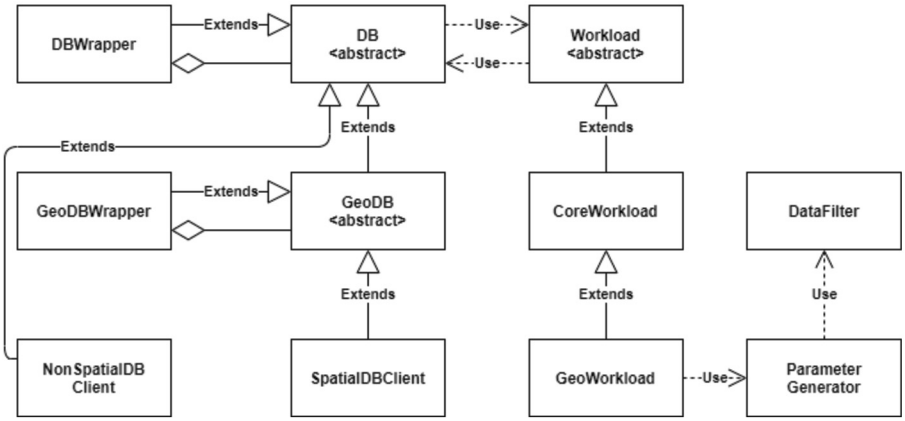

Fig. 1. GeoYCSB Design Architecture.

ing nearness, within, and intersection operations, hiding their implementation details from the YCSB client.
- The GeoDBWrapper class provides a geospatial database interface layer with the capability to measure performance metrics.
- The GeoWorkload class implements one experiment scenario with geospatial workloads. It is also in charge of setting up the connection to the in-memory data store and parsing the workload configuration file for the particular query mix. It also holds some metadata about the dataset(s) in use, relevant to the geospatial workloads under test.
- The ParameterGenerator class defines an in-memory data structure that houses spatial data loaded from the database and also defines associated accessors. The Graffiti Abatement Incidents, Building Footprints, and Tempe Public Schools data of the city of Tempe are loaded to the in-memory data structure for use as the parameters of geospatial queries. Its other responsibilities consist of synthesizing additional geospatial data based on the workload parameters, populating the in-memory data store with geospatial data from the database, and preparing parameter values for the workload by fetching them from the in-memory data store. Any in-memory key-value data store can be used without disturbing the functionality of GeoYCSB. In this study, we use Memcached.
- The DataFilter class specifies fields and embedded documents to be loaded into the in-memory store of the ParameterGenerator.
- The SpatialDBClient class represents the spatial NoSQL database binding for GeoYCSB. The class overrides the geospatial operations defined in the GeoDB class in terms of the dialect of the given NoSQL database.

3.2. Geospatial benchmark workloads

GeoYCSB benchmark workloads are defined by the representative geospatial queries such as nearness, within, and intersection.

- The nearness query takes a point in terms of latitude and longitude and returns documents in the order of their distance from the specified point. The query sorts the documents from nearest to farthest with respect to the distance from the specified point.
- The within query specifies a bounding box and finds documents for which their spatial data is completely contained in the bounding box.
- The intersection query takes a location of any GeoJSON type and returns documents for which their spatial data intersects with the specified GeoJSON location. Unlike the within query, this query returns a document for which their spatial data merely passes through or overlaps the specified GeoJSON location.

The GeoYCSB framework preserves the extensibility of YCSB in a way that a new geospatial workload can be implemented. Also, the client implementing the binding for any spatial NoSQL database can be integrated into the framework. To use other datasets, some modifications are necessary, the majority of which taking place in the ParameterGenerator class. We detail the particulars of these extensions in Section 6.

3.3. Performance metric

Throughput is the primary performance metric used in this study. Specifically, throughput in this study's context means the maximum throughput that a system can achieve for the given workload and system configuration. The arrival rate of requests is increased until the system is stably saturated for the given workload and system configuration, and then the throughput is measured. The number of GeoYCSB threads and operation count to saturate a cluster varies depending on each experiment's system configuration, such as the number of nodes in the cluster. We calibrated them for each experiment.

4. GeoYCSB microbenchmark

The GeoYCSB microbenchmark comprises of both data-loading queries and representative geospatial queries, namely, nearness, within, and intersection. For an apples-to-apples comparison in our experiments, the geospatial queries of MongoDB and Couchbase are written according to the common interface defined in the base type, the GeoDB class. These queries are also written so that a MongoDB query and the corresponding Couchbase query with the same goal return the same result for the same input. This section presents the dataset and workloads used for the microbenchmarking experiments conducted in this study, followed by the results of the experiments and our analysis.

4.1. Dataset

Creating read and write operations for benchmarking any given database system requires understanding the structure of the dataset. In the microbenchmarking experiments, we use the Graffiti Abatement Incidents dataset of the city of Tempe [18]. The city of Tempe Public Works department has an active graffiti abatement program. Hotspot regions are identified by government agents, police, and lawmakers, enabling town planners to devise policies and strategies to remove graffiti promptly. The original Graffiti Abatement Incidents dataset has about 13,000 entries. We synthetically

generate geospatial data according to the schema of this dataset, increasing its size to about 4GB with 10 million documents.

4.2. Microbenchmark workloads

Three workload mixes, Workloads A, B, and C are used for microbenchmarking. The type and density of queries can be easily tuned by modifying the configuration file.

- Workload A consists of 100% specific geospatial read (nearness, within, or intersection) operations. This workload serves to isolate the impact of write operations from the system performance.
- Workload B consists of 80% specific geospatial read (in this case, nearness only) operations and 20% geospatial write operations.
- Workload C consists of 30% nearness operations, 25% within operations, 25% intersection operations, and 20% geospatial write operations. A more complex ratio for query density is chosen to simulate a real workload.

In real scenarios of serving applications, the data access distribution may vary across applications. YCSB comes with several built-in random distributions, including Uniform, Zipfian, and Latest. GeoYCSB leverages these built-in distributions and fetches parameters for the geospatial queries from the ParameterGenerator according to the specified distribution.

4.3. Microbenchmarking experiments and result analysis

4.3.1. Experimental setup

Microbenchmarking experiments are conducted on a single node system first, and then on multiple-node clusters for horizontal scalability evaluation. We use Couchbase Server Edition 5.1.0 and MongoDB Community Edition version 3.4 in the experiments. For the single node and cluster setup, we use AWS EC2 m4.large instances with two vCPU, 8 GB of memory, and 16 GB of storage. All instances are provisioned within the same AWS rack and data center.

4.3.2. Performance evaluation of a single node system

The goal of the following experiments is to analyze how efficiently systems support individual geospatial queries for the given workload and system parameters. We also aim to study any performance impacts caused by a query's type and density in a mixed workload setting. We use a uniform data access distribution for all single node experiments.

Workload A In this experiment, we evaluate the performance of MongoDB and Couchbase under Workload A. Workload A consists of 100% specific geospatial read (nearness, within, or intersection) operations in a way that we can analyze the results without the impact of write operations.

In Workload A of Couchbase, nearness and within queries use FTS and are supported by a FTS-based spatial index. It is not straightforward to use FTS for intersection queries. Therefore, we used a spatial view for the intersection queries. The results are presented in Fig. 2.

We observe that the throughput of intersection queries is substantially lower than that of nearness and within queries in both MongoDB and Couchbase. We consider this result is due to the different levels of calculation complexity that the queries involve. Intersection queries require considerably more calculations than nearness and within queries. The results also show that the throughput difference between nearness and within queries of MongoDB is 21.3% while the corresponding throughput difference

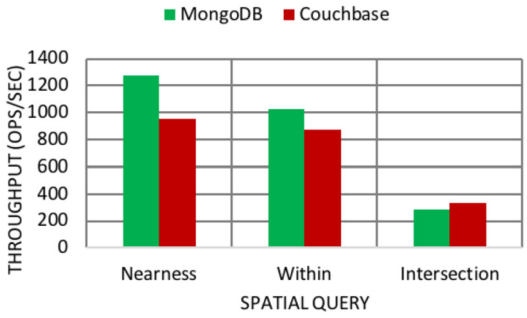

Fig. 2. Throughputs of Individual Geospatial Queries from MongoDB and Couchbase under Workload A.

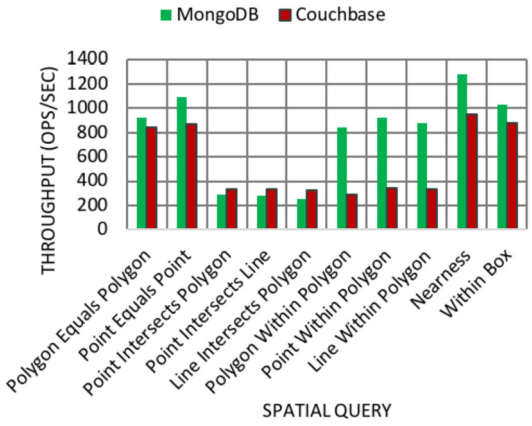

Fig. 3. Throughputs of Topological Relationship Functions defined by OGC.

in Couchbase is only 8.5%. In Couchbase benchmarking, because both nearness and within queries are supported by a FTS-based spatial index, they result in a similar performance. In MongoDB benchmarking, we use the geometry specifier $box for the $geoWithin operator to make it comparable to the box-based within query of Couchbase. Since $box is only supported by 2d indexes, within queries use the 2d index while nearness queries use the 2dsphere index. The use of different types of indexes can further contribute to the throughput difference between the nearness and within queries of MongoDB. It is interesting to note that the throughput of the intersection queries of Couchbase is higher than that of MongoDB. We consider that the R-tree spatial index of Couchbase, which can accommodate multi-dimensional data, works better than the B-Tree spatial index of MongoDB for complex geospatial queries such as intersection queries.

Workload A is extended to test topological relationship functions defined by the Open Geospatial Consortium (OGC) [19]. The results are presented in Fig. 3. This experiment demonstrates that various geospatial functions can be described in terms of nearness, within, and intersection queries to make up GeoYCSB workloads.

Workload B In this experiment, we study the impact of geospatial write operations on the throughput of the systems. The insert operation is chosen as a representative write operation. Workload B consists of 80% of nearness queries and 20% of geospatial insert operations. Both reads and writes are counted to calculate throughput. The results are presented in Fig. 4.

In MongoDB, the throughput measured under Workload B does not degrade significantly as compared to the throughput measured under Workload A for all the geospatial queries tested. However, the throughput degradation under Workload B is notable for nearness and within queries in Couchbase. There are a couple of factors to consider for further performance optimization. With 20% insert

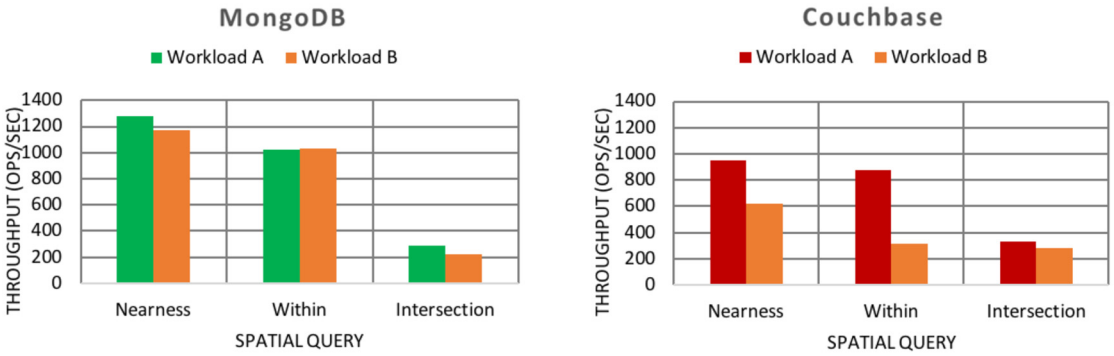

Fig. 4. Throughput Comparisons between Workloads A and B for Individual Geospatial Operations in MongoDB and Couchbase.

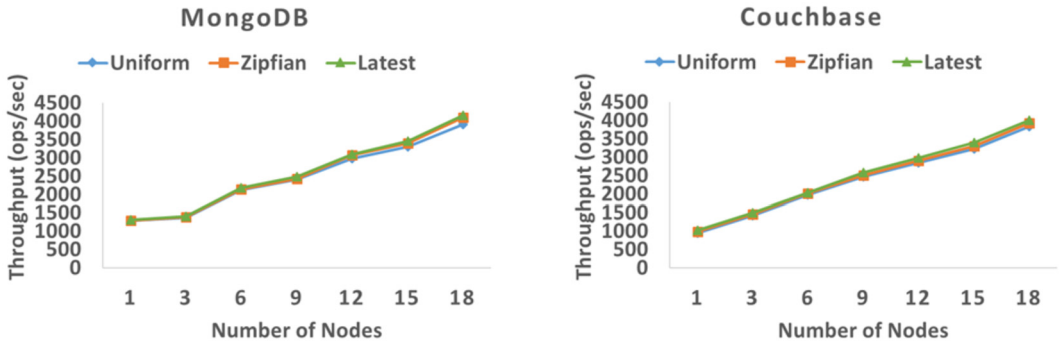

Fig. 5. Throughputs from MongoDB and Couchbase under Workload A with Various Data Access Distributions.

operations, the database size grows as insertions are in progress. Therefore, the sensitivity of the FTS-based nearness and within queries to the database size can be examined. Also, the overhead of the insertion algorithm of Bleve can be further investigated.

4.3.3. Horizontal scalability evaluation of a multi-node cluster

In the following scalability benchmarking experiments, the number of nodes in the cluster is varied while workload parameters are fixed. The throughput is measured for the given number of nodes. The ultimate goal of our scalability benchmarking is to find the impact of sharding and replication on the scalability of MongoDB and Couchbase running on a cluster of multiple nodes.

We set the replication factor to 3 for both systems so that for every primary copy (or active copy) within the cluster, there exists two replicas of the same data. The replication is configured in a way that the primary nodes of MongoDB and the active nodes of Couchbase serve both reads and writes. A consistency level specifies how many replicas should acknowledge for the system to consider the given operation successful. Therefore, with a higher consistency level, the system has to wait longer to complete a given operation. The client may consider the system is not available for the operation if the waiting time exceeds a specified threshold. We study the impact of the consistency level on the throughput by comparing throughputs measured with a low consistency level to those with a high consistency level.

MongoDB currently does not support a geospatial shard key. We follow the recommendation from the MongoDB documentation [13] and choose a non-spatial field as a shard key, namely, the ObjectID. Due to the monotonically increasing feature of ObjectIDs, the hash-based sharding strategy is adopted to eliminate the maximum chuck problem (that is, all new inserts are routed to one chunk) and thus evenly distribute data across nodes. Couchbase uses the hash value of document ID, which is a synonym for ObjectID, to assign documents to vBuckets. This configuration of sharding and replication of MongoDB and Couchbase are comparable.

We also test the impact of data access patterns, namely, Uniform, Zipfian, and Latest, on the scalability and performance of these systems.

Workload A With 100% read (nearness) operations, the results in Fig. 5 show that the throughput increases as more nodes are added. With well-tuned sharding, requests are evenly distributed across multiple nodes, and thus, the system horizontally scales. An interesting phenomenon observed from the results is that there are diminishing returns to scale in MongoDB. Adding three more nodes to make a six node cluster increases throughput by 56% while adding another three nodes to make a nine node cluster increases the throughput by only 13%. From a nine to 12 and 12 to 15 node cluster, we can observe a similar trend of a major throughput increase followed by the diminishing return of a minor throughput increase. Couchbase scales almost linearly with the number of nodes. The results also show that the impact of data access distribution on performance and scalability is not significant. This indicates that a system can be robust in handling different data access patterns from various applications if sharding and replication are properly deployed.

Workload B The goal of this experiment is to examine the impact of the consistency level on throughput. An application can tune the level of consistency to increase the consistency of data at the cost of higher latency for individual operations accessing replicated data. For both MongoDB and Couchbase, the read consistency level is set to default which makes a read considered successful as soon as it gets data from a primary (or an active) node.

In this experiment, we focus on the impact of the write consistency level of MongoDB and Couchbase on the throughput. For MongoDB, w:1 and w:all are chosen to represent a low consistency level and a high consistency level, respectively. A write operation with w:1 is successful when the primary node responsible for the write acknowledges. With w:all, a write is successful when both the primary and all involved replicas, that is, 2 replicas, ac-

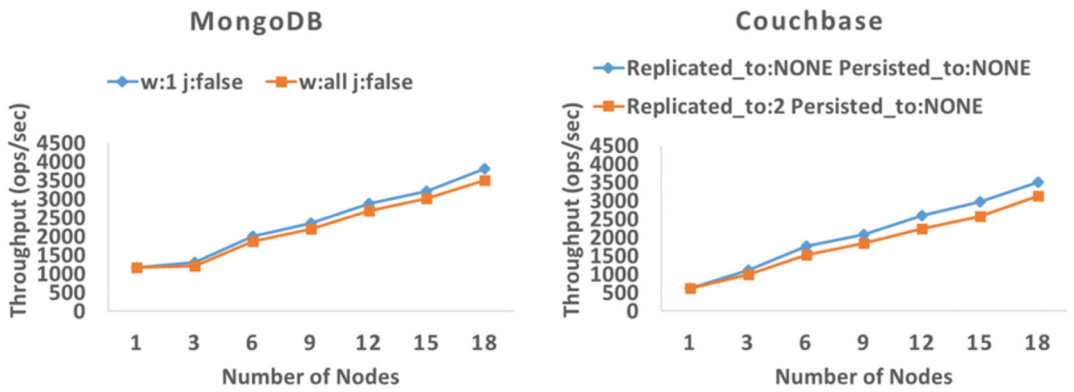

Fig. 6. Throughputs from MongoDB and Couchbase under Workload B.

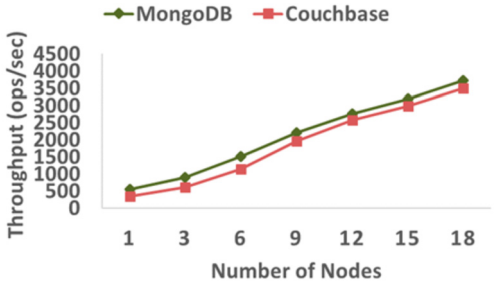

Fig. 7. Horizontal Scalability Comparison between MongoDB and Couchbase under Workload C.

knowledge it. The Couchbase write consistency level corresponding to w:1 is replicated_to: NONE, which makes a write operation successful when the active node responsible for it acknowledges without replicating the write to any replica. Replicated_to:2 corresponds to w:all, which makes a write operation successful when the write is done in the active node as well as its 2 replicas. By default, journaling is disabled in MongoDB, and the persist_to parameter is set to NONE in Couchbase. In this way, a write is acknowledged when the data is written in memory for both systems.

The results of these experiments are presented in Fig. 6. In both MongoDB and Couchbase, a higher consistency level lowers the throughput. In MongoDB, the percentage difference between the throughput with a low consistency level and the throughput with a high consistency level lies in the range between 6.5% and 8.3%. In Couchbase, the corresponding percentage difference lies between 11% to 15%. This experiment was conducted on a cluster of healthy nodes. In practice, if a replica is down, an operation with a high consistency level may result in an intolerably long latency, and the client may perceive that the system is not available for the operation. Our results confirm that the consistency level is an important factor that affects the query performance when data are replicated.

Workload C The goal of this experiment is to study the scalability of MongoDB and Couchbase under a workload with a more complex ratio for query density. Specifically, the workload consists of 30% nearness operations, 25% within operations, 25% intersection operations, and 20% spatial write operations. It is desirable to find out real workload mixes from applications using the actual query logs of a system. However, generally, query logs are not available for benchmarking due to many reasons, including privacy concerns. One of the essential features of any benchmark framework is its ability to easily tune the workload mix to simulate various real workloads. The results of this experiment are presented in Fig. 7. MongoDB and Couchbase scale well under Workload C, with the differences in performance between the two reaching as high as 37%. When these systems scale to a cluster size of 18 nodes, the performance difference between MongoDB and Couchbase shrinks to as low as 6%.

5. GeoYCSB macrobenchmark

The GeoYCSB macrobenchmark comprises of four common use cases for graffiti abatement applications. This section presents the datasets and workloads used for the macrobenchmarking experiments, followed by the results of the experiments and our analysis. We chose MongoDB for the macrobenchmarking experiments considering its spatial support and scalability tested in the microbenchmarking experiments and its popularity [20].

5.1. Dataset

We use two additional datasets with the Graffiti Abatement Incidents dataset in the macrobenchmarking experiments: Building Footprints [21] and Tempe Public Schools. For the Tempe Public Schools dataset, we went to each public school district page from the Tempe School website [22], found their list of schools and compiled them all into one list as long as they are within the boundaries of the city of Tempe. These two datasets provide more context and purpose to our geospatial queries. For example, finding graffiti around a school is meaningful and more realistic than searching for graffiti around a randomly selected point that may translate to an inaccessible location, such as an airport runway. The random location may have a low priority for cleaning, such as an abandoned bridge that sees little foot traffic. The building dataset contains a record of every building in the city of Tempe for a total of 55,000 original documents, while the public school dataset contains 34 public schools from 3 different school districts. These datasets are expanded with the graffiti dataset to layer with the synthesized data points. In addition to datasets, we also layer a grid of neighborhoods, defined as a 1-mile square block, across the entire range to demarcate meaningful units in which graffiti can cluster.

For macrobenchmarking experiments, we adopted a synthesis method that would maintain the distribution of data and geospatial meaningfulness between the datasets.

The method is to replicate the original dataset along the formation of a grid: each cell in the grid is the size of a minimum bounding box surrounding the city of Tempe, Phoenix. The first cell is located at the upper left corner of the grid and contains documents of the original dataset. Then, these documents are duplicated and their coordinates shifted into the next cell over, with little to no overlap between the cells. The ultimate result is a large dataset of a synthetic city with no two documents possessing the same geometrical coordinates while still maintaining the original

distribution of the dataset. We replicate the original Graffiti Abatement Incidents dataset over a 9x9 grid for a total of roughly 1 million documents. The Building Footprints dataset and the Tempe Public Schools dataset are synthesized in the same manner. The resultant building and school collections contain roughly 4.5 million and 3000 documents, respectively. The total size of the synthesized data is about 4GB. These three datasets may be layered like maps to perform queries with meaningful and realistic uses, as we discuss in the next section.

5.2. Macrobenchmark workloads

We develop four common use cases for graffiti abatement applications to simulate realistic workloads. Three pertain to geospatial queries, including common GIS operations, while the fourth is a write-intensive use case.

Use Case 1: This use case is to perform a series of point-of-interest searches using nearness operations. The graffiti abatement team using the application may wish to find the school with the most graffiti, the point-of-interests, in a 500-meter distance to clean. All schools are checked in random order and the graffiti belonging to the school with the highest amount are returned.

Use Case 2: This use case is to perform basic data analysis using a spatial join performed by within operations. Instead of searching by school, the graffiti abatement team may want to find the neighborhood, defined as a 1-square-mile block, with the highest graffiti density. Neighborhoods are spatial joined with the incident collection and the graffiti counts of the neighborhoods are recorded. The graffiti belonging to the neighborhood with the highest amount is returned.

Use Case 3: This use case involves two spatial join operations performed by intersection operations. Spatial joins are done between the layer of neighborhoods and the building collection as well as the layer of neighborhoods and the graffiti collection. The graffiti abatement team may initiate this use case to find graffiti within the neighborhoods with the highest building density. For each neighborhood, an intersect operation is performed to find all buildings within the neighborhood, and the coverage of buildings in the neighborhood is tallied. Then, the top five densest neighborhoods are spatially joined with the incident collection, and their graffiti are retrieved.

Use Case 4: This use case involves a write operation that removes graffiti found from Use Case 1.

Each use case scenario is implemented in the MongoDB query language and used to constitute macrobenchmark workloads.

5.3. Macrobenchmarking experiments and result analysis

5.3.1. Experimental setup

We experienced that the macrobenchmarking experiments entail more system resources than the microbenchmarking experiments. There is significant performance degradation when the working set exceeds both the WiredTiger cache and the file system cache. Queries from macrobenchmark workloads need a larger working set than those from microbenchmark workloads. For instance, if a query involves both the graffiti collection and the building collection, which are sharded and also maintain a 2dsphere index, its working set size will likely increase. Therefore, we scale the AWS EC2 instances to m4.xlarge with four vCPUs, 16GB of memory, and 16GB of storage to keep the working set size within memory and avoid evictions.

In macrobenchmarking experiments, we define throughput as the number of completed use case executions per second. Also, the specifier $geometry is used for the $geoWithin operator to describe the boundary of searches in terms of a GeoJSON Polygon. (In microbenchmarking experiments, the specifier $box is used for $geoWithin for comparability with Couchbase.)

5.3.2. Impact of data size on throughput in a single node system

In this experiment, we study the impact of data size on the performance of a single node system. As we have synthesized data according to a grid formation, the data size is represented in terms of the expansion of that grid—a polynomial increase of n^2 or nxn in size, where n is the number of rows in the grid. For example, 1x1 would refer to the original dataset of 13,000 graffiti incident documents, while 2x2 would refer to that amount multiplied by four. As the data size increases, a given use case requires more queries to achieve its goal. For example, with the data size 1x1, Use Case 1 is required to search through 34 schools, and this number increases to 136 with the data size 2x2.

The results are presented in Fig. 8. Noticeably, there is a large throughput gap under the data size 1x1, where Use Case 3 significantly underperforms compared to Use Cases 1 and 2. The performance difference is due to the disparity between the number of geospatial operations involved in each use case and the performance of the operations themselves. Use Case 1 is comprised of nearness queries that probe schools. Use Case 2 and Use Case 3 mainly involve within and intersection queries, respectively, and probe neighborhoods, of which there are twice as many when compared to the number of schools. Nearness also outperforms both within and intersection, as seen in the results of the microbenchmarking experiment under Workload A. In addition, Use Case 3 has two parts: first, to find the number of buildings inside each neighborhood along with their areas, and second, to sum and sort the results of the first step before performing additional intersection operations on the densest neighborhoods, retrieving the overlapping graffiti as the final result. At low data sizes, where the number of queries involved in the first step is small, this additional work is significant.

As the data size and, consequently, the number of queries sent to the system grows, the single node system saturates, resulting in extremely low throughputs regardless of the type of use cases. With the system fixed at a single node, this increase stresses the system, and therefore, the queries comprising the use cases experience longer latency. This is reflected in the latency graph in Fig. 8. Continuing to increase the data size further — and thus, the number of queries to serve — on a saturated single node system will only result in higher latency while the throughput is unable to improve.

In the following section, we present the scalability of the MongoDB sharded cluster in supporting these use cases running over the largest data size 9x9 under our consideration.

5.3.3. Horizontal scalability evaluation in a cluster of multiple nodes

In this experiment, we study the horizontal scalability of the system against Use Cases 1, 2, and 3. The most demanding data size 9x9 under our consideration is chosen.

To focus on examining the impact of sharding on the system's scalability, we do not replicate the shards. Replicating shards not only requires additional storage space but also uses system resources to maintain secondary servers, and thus the throughput can be affected. Since MongoDB enforces that each shard is implemented as a replica set, we use a replica set of one member, which will serve as the primary shard, and direct all queries to the primary shard. We keep mongos and the config server on two separate machines to separate the use of resources consumed by the mongod processes and mongos maintaining connections to all servers. A bottleneck of mongos resources would affect the throughput of the entire system.

The results presented in Fig. 9 show that sharding horizontally scales the system, and thus the throughputs of all use cases in-

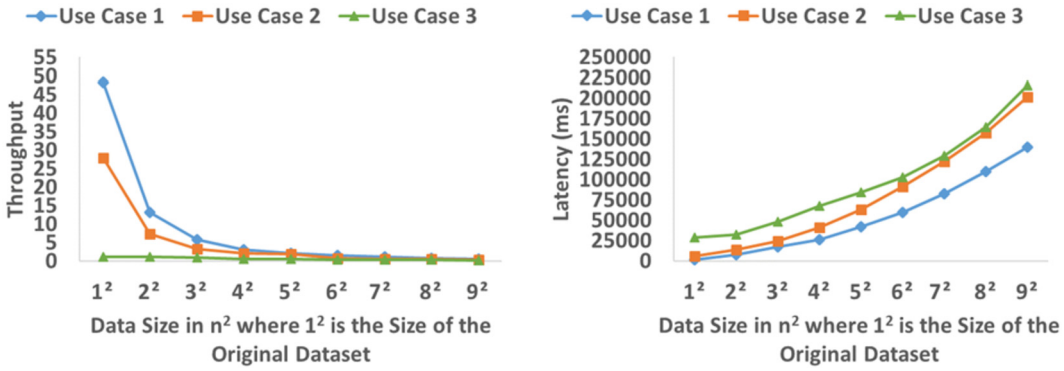

Fig. 8. Impact of Data Size on Throughput (use case executions/sec) and Latency.

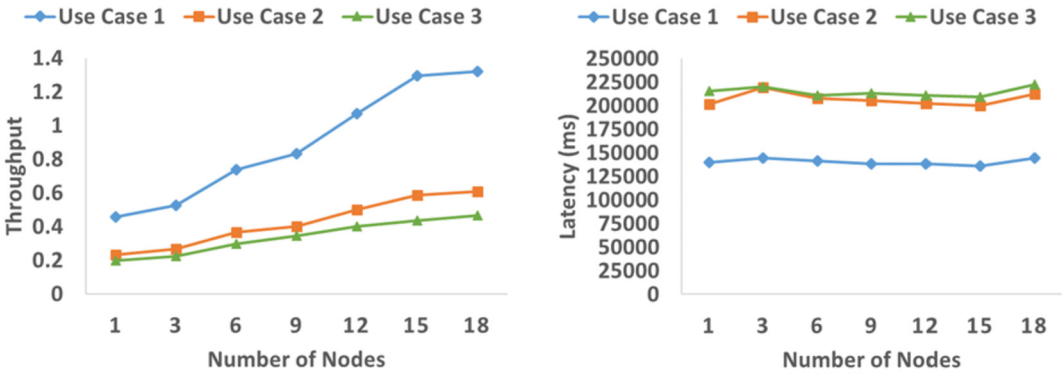

Fig. 9. Horizontal Scalability Evaluation with Various Use Cases.

crease as more nodes are added to the cluster. The throughput of Use Case 1 significantly improves, but the throughput improvement of Use Cases 2 and 3 is minor. Also, for any given number of nodes, it is observed that the throughput of Use Case 1 is higher than those of Use Cases 2 and 3 while the throughputs of Use Cases 2 and 3 are more closely aligned.

Use Case 1's throughput difference can be contributed to its nearness operations. Due to using points in the query parameter, the computation involved with nearness operations is significantly less than within and intersection, which use polygons. Polygons require a substantially higher amount of computation. We can thus see that Use Case 1 comprising of nearness operations performs and scales better than Use Cases 2 or 3.

The alignment of Use Cases 2 and 3 can be explained similarly. In our macrobenchmarking experiments, within and intersection operations function similarly because both of them use a polygon for the $geometry specifier and a 2dsphere index. That explains the similar behaviors of Use Cases 2 and 3 which rely on within and intersection operations, respectively. However, Use Case 3 performs some additional queries to retrieve the target graffiti. Therefore, the throughput of Use Case 3 is still lower than that of Use Case 2.

As shown in Fig. 9, with well-tuned sharding, the latency remains stable while the throughput increases as more nodes are added to the system.

5.3.4. Impact of write consistency level on the performance of write-intensive use cases

This experiment is to examine the impact of write consistency level on the performance of write-intensive use cases running over replicated data. The graffiti collection is sharded into the given number of nodes and each shard is replicated in a replica set of three members. We set up this experiment in a way that primary shards are held on their own AWS instance while the two secondary shards share an AWS instance. The write concern w:1 is used to represent weak write consistency level while w:all is used for strong write consistency level. Then, we measure the runtime in milliseconds of write operations involved in Use Case 4—specifically, deleting roughly 400 graffiti documents found by a single Use Case 1. (Use Case 1 finds the school with the most surrounding graffiti and returns the corresponding graffiti.) The runtime includes the time to delete the found graffiti, excluding the time to find it. For the given number of nodes, after each run of Use Case 4, the database is reloaded with the removed data so the next run can delete the same documents.

As can be seen in Fig. 10, when the write concern is set to w:all, the runtime is increased by roughly 20% as compared to the case of write concern w:1. This is in line with expectations, as the write operation must be replicated to the number of nodes specified by the write concern for the operation to be acknowledged.

Another factor we are interested to examine is the impact of sharding overhead on the performance of write queries. As the number of shards increases, the overhead involved in maintaining connections among the shards also increases. We fix the arrival rate of write operations and only increase the number of shards in each run. The result shows that runtimes do not change significantly as the number of shards increase, indicating the impact of sharding overhead on query performance is not substantial.

6. The extensibility of GeoYCSB

A benchmark framework should be extensible to create new workloads and add support for new systems. In this section, we demonstrate the extensibility of GeoYCSB.

We first present MongoDB benchmarking experiments using a combination of two datasets with a wider spatial extent and many complex geometries such as polygons and linestrings. The original datasets cover Japan's entirety in JSON documents with a total size of about 1GB. We synthesize the combined datasets to make

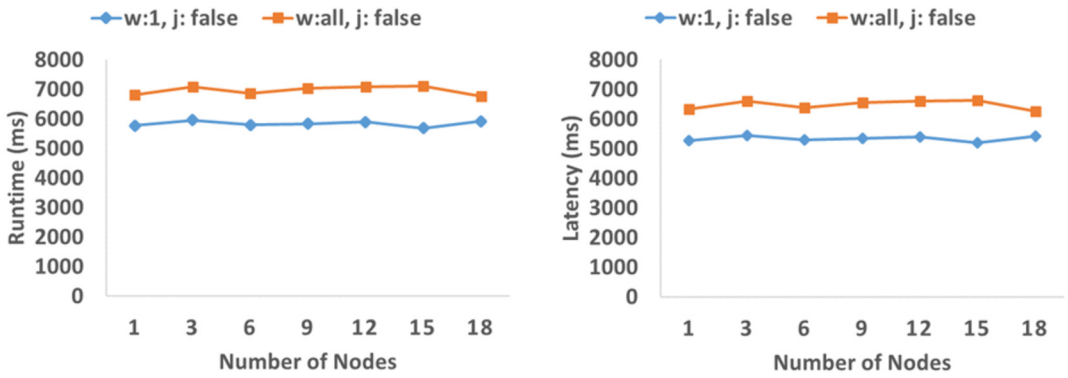

Fig. 10. Impact of Write Consistency Level on the Performance of Write-Intensive Use Cases.

it 40GB and conduct a horizontal scalability test by increasing the number of nodes in the cluster up to 18. We detail the component of GeoYCSB, ParameterGenerator, responsible for accommodating the various geospatial geometries of a new dataset. One of the future works of the GeoYCSB project is to automate the end-to-end benchmarking process for reliable and reproducible benchmarking. We present the cornerstone of this plan in the horizontal scalability test by automating cloud resource allocation and DBMS deployment.

Next, we present another set of experiments to demonstrate the extensibility of GeoYCSB for benchmarking NoSQL systems that use different data models. We chose Accumulo, a wide-column store, with GeoMesa, an open-source framework, applied on top. GeoMesa enables a wide range of geospatial queries on distributed computing systems as well as geospatial indexing. Therefore, we can benchmark Accumulo under workloads consisting of a broader range of geospatial operations than what is supported by MongoDB. Specifically, the workloads consist of nine spatial predicate functions derived from the DE-9IM (Dimensionally Extended Nine-Intersection Model) [23].

6.1. Dataset

We use a combination of two open-source datasets in GeoJSON format provided by the Japanese Ministry of Land, Infrastructure, Transport, and Tourism (MLIT) [24,25]: one containing all counties and another containing all bus routes in Japan. The counties dataset comes native in JSON format, but the bus routes dataset is converted from shapefile to JSON using QGIS, an open-source software that provides visualization of spatial data and data editing [26]. The counties dataset consists of around 100,000 polygons and the bus routes dataset consists of around 26,000 linestrings, totaling roughly over 1 GB. Using the same synthesis method covered in the macrobenchmark section, we synthesize this dataset twice: once to form a 2x2 grid, approximately 4GB in size with roughly 475,000 polygons and 100,000 linestrings, and again to form a 7x7 grid approximately 40GB in size with roughly 5,800,000 polygons and 1,300,000 linestrings.

6.2. GeoYCSB components responsible for supporting a new dataset and a new database system

Several GeoYCSB components are involved in supporting a new dataset and a new database. The ParameterGenerator class holds the most responsibility when adding a new dataset to GeoYCSB. It possesses the following important roles: to synthesize new documents as needed based on the workload's parameters, to populate the in-memory data store (e.g., Memcached), and to fetch parameters from the in-memory data store for the queries that constitute the workload. Note that for large datasets, such as in our following experiments, it may be unrealistic to store the entirety of a table in the in-memory data store, as the data store occupies a portion of RAM on the same machine as GeoYCSB. In this case, the ParameterGenerator class populates the in-memory data store with document geometries at a random 5% rate to ensure the even distribution of potential parameters while still maintaining a large pool to pull from, without reserving too much memory from GeoYCSB. Some modifications are necessary to add support for a new dataset. The ParameterGenerator class must be modified to ensure its synthesis logic properly accounts for the new dataset's geometry types. In addition, the GeoWorkload class' metadata for the given dataset must be updated, such as the new tables, their document counts, and the new longitude and latitude values of the dataset's bounding box.

To add support for a new database system, the ParameterGenerator and GeoWorkload classes need to be modified. Also, a SpatialDBClient that implements the binding of the new database needs to be added. The ParameterGenerator is updated to fetch all required geometries according to the needs of the database's queries. Additionally, if the new database system expects data in a non-JSON format, the ParameterGenerator needs to convert JSON to the expected data type. For example, the ParameterGenerator class converts JSON to WKT in the case of GeoMesa so that the parameters are prepared in WKT before they are fetched for parameterization. The GeoWorkload class receives minor additions to add support for any new operation required by the workload.

6.3. Scalability benchmarking experiment using a large dataset with complex geometries

6.3.1. Workloads

In order to observe scaling with a large dataset consisting of complex geometries, we perform 100% intersection operations between polygons from the counties dataset and linestrings from the bus routes dataset. The intersection operation is chosen as the most expensive geospatial query of MongoDB to form the workload. The type and density of queries can be tuned as shown in the microbenchmarking experiments. This workload is run on both the 4GB dataset and the 40GB dataset for single node, 6-node, 12-node, and 18-node sharded clusters where the throughput is captured at the point the system saturates. We make a cold-run of each experiment three times, and the throughputs are averaged for the final result.

6.3.2. Experimental setup

We deploy GeoYCSB on an AWS EC2 m5a.4xlarge instance with 16 vCPUs, 64GB RAM, and 32GB storage. Mongos, config servers, and shards are each deployed on a m5a.xlarge instance with 4 vCPUs, 16GB RAM, and 64GB storage. All machines are running

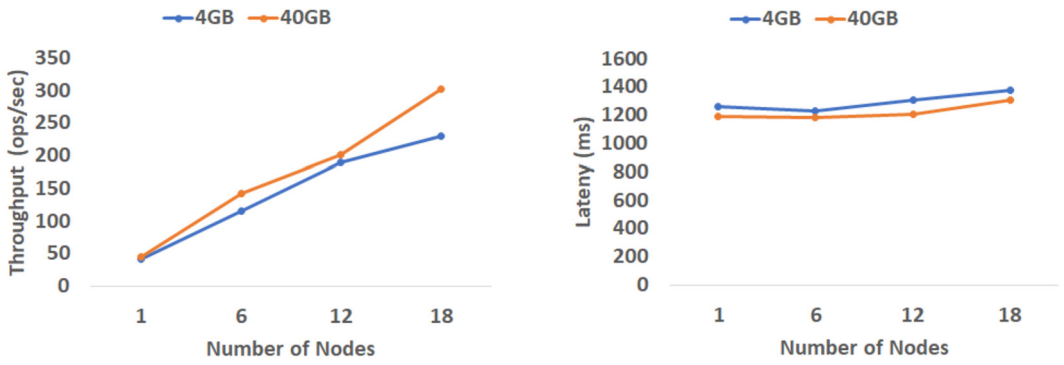

Fig. 11. Impact of Data Size on Horizontal Scalability.

Ubuntu 18.04 and MongoDB Community Edition version 4.4. We deploy the mongos process on its own instance to ensure throughput is not throttled by sharing memory with other mongod processes, and the config server is also isolated on its own instance for the same reason. As sharding is the key contributor to the horizontal scalability of distributed NoSQL systems, we deploy sharding without replicating data, and thus the shards as well as the config server are implemented in a replica set of size 1. We use Ansible [27] to assist in automating the configuration of AWS EC2 instances as well as quickly restarting our instances for cold-run benchmarking. The entire MongoDB configuration in the scalability test is set up, started up, and shut down through Ansible, allowing the main focus to be running GeoYCSB.

6.3.3. Results and analysis

The experimental results are presented in Fig. 11. As we increase the number of nodes, both throughputs from the 4GB dataset and the 40GB dataset experiments increase. However, the impact of sharding on the horizontal scalability is stronger with the 40GB dataset than the 4GB dataset, as 4GB is not big enough to take proper advantage of sharding at larger cluster sizes. This is most evident in the 18-node experiment, where the throughput increment from a 12-node to an 18-node cluster with the 4GB dataset is not as steep as that with the 40GB dataset. The difference in their performance gains is as high as 30%. When the bottleneck for performance is the amount of available system resources, the system effectively scales as we add more nodes to the cluster. However, depending on the size of the dataset, there is a point at which adding nodes to the sharded cluster will result in diminishing returns, as the amount of resources becomes sufficient to serve the dataset. Eventually, continuing to add shards to this cluster will not result in any significant performance gains. These points are reached faster by smaller datasets than larger datasets, due to larger datasets naturally requiring more resources, such as memory and storage, to index and query efficiently. Larger datasets therefore have a higher potential for performance growth using horizontal scaling.

6.4. Benchmarking GeoMesa on top of Accumulo, a wide-column database, with a wide range of geospatial queries

The purpose of these experiments is to demonstrate the extensibility of GeoYCSB for new systems using different NoSQL data models with a wide range of geospatial queries. We chose GeoMesa running on top of Apache Accumulo, a wide-column database, for this purpose.

6.4.1. GeoMesa and Accumulo

GeoMesa is an open-source framework that makes geospatial indexing and querying possible on wide-column databases, such as Google Big Table, Apache Cassandra, and Apache Accumulo [28]. Fig. 12 presents the main components of the GeoMesa and Accumulo architecture under test.

We use GeoMesa with Apache Spark (a.k.a. the GeoMesa-Spark module) to test a wider range of geospatial queries, specifically nine geospatial operations from the DE-9IM, than GeoMesa alone can support. Spark's parallel computation enables GeoMesa to support more complex geospatial operations. Zookeeper is responsible for coordinating Accumulo processes and maintains the metadata to route client requests to the Accumulo tablet servers. An effective geospatial index is crucial to improve the performance of geospatial queries. By default, GeoMesa creates a Z2 index and a record table for points as well as a XZ2 index and a record table for spatially extended objects, defined as non-point objects, such as polygons and linestrings [29]. The datasets we use in this experiment contain extended geometries only. For each dataset, GeoMesa creates the XZ2 index using the XZ-ordering space-filling curve along with an Accumulo table of key-value pairs where the row key consists of the XZ2 value followed by the UUID of the record. The XZ-ordering is a technique to approximate geometries with spatial extensions into single integer representations while preserving the locality between the geometries. We disabled the creation of an ID index because it will not be used under our workloads.

The GeoMesa-Spark module processes a query through two steps. First, GeoMesa-Spark will connect to Zookeeper to find out the tablet server addresses of the table it is looking for. After obtaining the addresses, GeoMesa-Spark will access the tablet servers and submit read requests to obtain a superset of data. When data is returned to GeoMesa-Spark, GeoMesa-Spark will perform a second round of filtering to find out actual matching records and report back to the client, GeoYCSB.

6.4.2. Workloads

We create nine workloads, each consisting of 100% of a geospatial predicate function from the DE-9IM: equals, disjoint, intersects, touches, within, contains, crosses, covers, and overlaps. We fetch a parameter from the routes dataset for the first four operations and a parameter from the counties dataset for the following four operations, all of which are applied to the routes dataset. The overlaps operations are applied to the counties dataset, but we parameterize them with synthesized polygons. According to the DE-9IM, the overlaps operation takes two geometries with the same dimension, and their interiors must intersect to be considered overlapping. Since none of the counties' interiors overlap with one another, we use a synthesized polygon for the overlaps operation's parameter. The parameters fetched from the counties dataset and the routes dataset are polygons and linestrings, respectively.

6.4.3. Experimental setup

We deploy our client machine and Spark 2.4.7 on an AWS EC2 m5a.4xlarge instance with 16 vCPUs, 64GB RAM, and 32GB storage. GeoMesa 3.0.0 is applied on top of Accumulo 2.0.1. For a

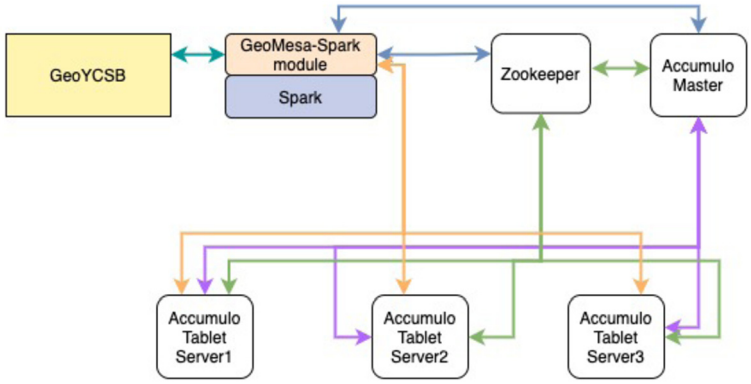

Fig. 12. GeoMesa Accumulo Architecture.

single node experiment, an Accumulo master, tablet server, and Zookeeper are deployed across two m5a.xlarge instances with 4 vCPUs, 16GB RAM, and 64GB storage. We isolate the tablet server from the Accumulo master and Zookeeper. For a multi-node experiment, we add two additional tablet servers on two m5a.xlarge instances with 4 vCPUs, 16GB RAM, and 64GB storage. Each tablet server is installed with Hadoop 3.2.2 as the internal file storage system. All machines are running Ubuntu 18.04.

6.4.4. Results and analysis

We measure the throughput of the system with 4GB and 40GB of data under each workload. We also scale the system from a single node to a three-node cluster to see the impact of sharding on the query performance. The experimental results are presented in Figs. 13 and 14, respectively.

The within and contains operations are logically equivalent according to the DE-9IM standard, with inverted parameters. The results consistently show that the within and contains operations perform similarly. For the overlaps operation, the use of a synthesized simple polygon as the parameter decreases the complexity of the operation's computations, explaining its comparatively high throughput.

The intersects operation is the inverse of disjoint. We confirm this in a separate experiment by finding that the query results returned by the disjoint operation and the negated intersects operation are the same. The reason for disjoint performing better than intersects is likely implementation-specific, depending on how GeoMesa optimizes the computation of these operations. To our best knowledge, there is no public documentation available to investigate this internal matter of GeoMesa.

The throughput of equals, touches, and covers is measured very low across all experiments. According to the DE-9IM, all three of these operations have stricter definitions than the rest of the operations, possibly indicating a heavier computational requirement. The results consistently show the throughput of these operations is significantly lower than that of the rest. We found that these operations' query performance is correlated to the low seek values monitored by the Accumulo Monitoring System. However, no documentation is available that defines the meaning of the seek value in the monitoring system's context. Thus, we could not derive any conclusion from the correlation. In separate experiments, we increase the number of spark servers from 1 (by default) to 2 to test the impact of parallel processing done by multiple spark servers on the query performance, but their query performance is not improved. However, when we double the number of shards from 3 to 6, keeping one spark server per node to decouple the gains of the spark servers' parallel processing from the query performance, the query performance is improved by about 30-40%. We presume that the cause of low query performance is the combination of the complexity of these operations and the lack of index efficiency.

The results show that increasing nodes improves the system performance for both the 4GB dataset and the 40GB dataset, as expected. Something to note is for the three-node 4GB experiment, the intersects operation, which performed better than within and contains for a single node, is worse than its three-node counterparts. This result is not reflected in the 40GB dataset experiment.

7. Related work

YCSB (Yahoo Cloud Service Benchmark) is an extensible open-source benchmarking framework aimed at cloud systems, mainly including NoSQL databases. Cooper et al. proposed YCSB in [8] and presented the performance comparison of Cassandra, HBase, PNUTS, and MySQL using YCSB. YCSB promoted active research for benchmarking distributed databases. Using YCSB, Rabl et al. evaluated representative key-value stores, wide-column stores, and distributed MySQL for the use cases of application management monitoring tools [9]. Khuhlenkamp et al. evaluated the scalability and elasticity of Cassandra and HBase [10]. The authors first reproduce the benchmarking results from [9] and then extend the experiment scope by including vertical scaling experiments and elasticity benchmarking. Neither of these two studies considers the impact of sharding and replication together on system performance. Using YCSB, Haughian et al. studied the impact of replication, consistency levels, and data access distributions on the performance of sharded Cassandra and MongoDB [11]. While YCSB is a prominent benchmarking tool for NoSQL systems, it currently does not support geospatial workloads. GeoYCSB is developed based on YCSB, inheriting its extensibility while enhancing it with new components that allow geospatial workloads to be supported. To our knowledge, there is no prior work on benchmarking replicated and sharded NoSQL databases using YCSB for geospatial workloads.

As spatial data and the popularity of location-based services surge, geospatial workloads become increasingly important. The characteristics of geospatial workloads differ from traditional database workloads, as represented by TPC benchmarks. Simion et al. stated the key differences are the computational complexity involved in spatial queries' evaluation of relationships between spatial objects and the unstructured nature of spatial data, which hinders storage optimization [30]. They developed a framework that can categorize spatial queries in terms of resource consumption, such as CPU intensity and I/O intensity, and can also characterize spatial workloads by varying the buffer size and query density in the workload mix.

Since the spatial queries that constitute geospatial workloads query the spatial relations of spatial entities [31], characterizing them naturally starts with the characterization of spatial relations. DE-9IM (Dimensionally Extended 9-Intersection Model) has been used by the Open Geospatial Consortium (OGC) for the definition of topological relationships in spatial databases. DE-9IM has

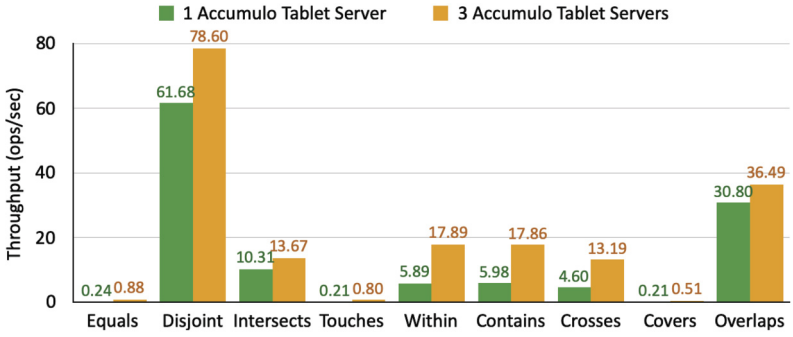

Fig. 13. GeoMesa Accumulo Throughput with 4GB data.

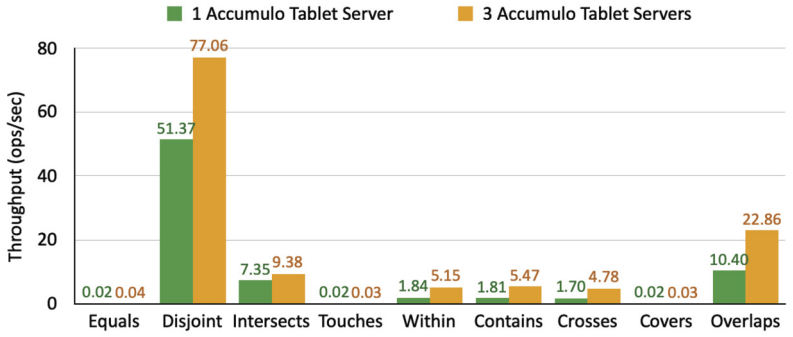

Fig. 14. GeoMesa Accumulo Throughput with 40GB data.

been widely adopted by various database systems (e.g., Oracle, IBM DB2, PostgreSQL) to define the topological operators of their spatial query language [32]. With GeoYCSB, one can evaluate a wide range of such spatial operations, as demonstrated by the use of workloads consisting of nine operations from DE-9IM in our extensibility test. Spatial joins are one of the most resource-demanding spatial queries, and thus a plethora of research has been done in the area of optimizing spatial joins [30]. GeoYCSB's macrobenchmarks involve intensive use of spatial joins and can be used to effectively evaluate spatial databases' performance for spatial joins.

Spatial NoSQL databases emerged with the essential features needed in geospatial scenarios, namely, flexible schema, cloud scalability, and geospatial data storage, management, and queries. Guo et al. surveyed the top 10 popular NoSQL systems' geospatial features in terms of supported geometry objects, geometry functions, spatial indexes, and data format [33].

A considerable amount of research has also been conducted in evaluating spatial data stores and big data processing infrastructures for efficient geospatial data management and processing. Ray et al. developed Jackpine [34], a spatial database benchmark that can support any database with a JDBC driver implementation. Jackpine comes with a microbenchmark to evaluate basic topological relationships and spatial analysis functions individually as well as a macrobenchmark that describes common use cases of geospatial applications. These benchmarks are used to evaluate PostgreSQL, MySQL, and Informix. Geographica [35] is a representative benchmark developed for geospatial RDF stores supporting the OGC GeoSPARL, an extension to the SPARQL query language for processing geospatial RDF data. It is developed following the approach of Jackpine benchmark [34]. Geographica comes with micro and macrobenchmarks composed of both synthetic and real-world workloads. Baralis et al. presented a performance evaluation of Azure SQL database and Azure Document DB for two common use cases of geospatial applications, retrieving location within a bounding box and writing geolocalized reports [2]. They studied the impact of the number of concurrent users, the number of records, and the performance level (configuration cost) on the average response time of queries. Their experiments were performed using a synthetically generated dataset. Duan et al. compared the efficiency of retrievals of a specified number of location points within a fixed bounding box for ArsGIS and MongoDB [3]. Alam et al. introduced SpatialIgnite, an extended Apache Ignite data system with geospatial operations, and conducted benchmarking experiments to compare the performance of SpatialHadoop (a Hadoop-based spatial data system), GeoSpark (a Spark-based spatial data system), and SpatialIgnite [36] using microbenchmarks similar to that of [34], with some extension.

8. Conclusions and future work

In this paper, GeoYCSB, a benchmark framework for geospatial NoSQL databases, was presented. GeoYCSB supports both microbenchmarks and macrobenchmarks.

Microbenchmarking experiments were conducted to evaluate the performance and scalability of MongoDB and Couchbase and the experimental results were analyzed. We found that the efficiency of the search algorithm of the spatial index structure is one important factor that affects query performance, especially for complex spatial queries such as intersection queries. We demonstrated that GeoYCSB allows for the description of various geospatial operations defined by OGC. Under workloads involving writes, the write overhead that the spatial index structure carries should be carefully studied to avoid performance degradation. Our experimental results show that both MongoDB and Couchbase scale well under various workload mixes, including a workload with a complex ratio for query density. Both systems exploit sharding, which contributes to their horizontal scalability by parallelizing user requests over multiple servers. With replication, the consistency level is a very important factor that trades off latency, and therefore subsequently affects the throughput of the system.

By extending GeoYCSB to include macrobenchmarks, a database can be tested under more realistic use cases relevant to the application. Macrobenchmarking experiments were conducted on MongoDB. We were able to test geospatial use cases such as point-

of-interest searches, spatial joins, and basic data analysis. We observed that with the use of sharding, MongoDB scaled well under all use cases, but observed use cases involving points scaled better than use cases involving polygons due to computational complexity. Careful selection of the geospatial operations involved and their parameters can decrease computational complexity and improve performance. We also demonstrated the exchange of performance for consistency by increasing the write consistency level in write-intensive use cases. In addition, we evidenced the extensibility of GeoYCSB by utilizing a large dataset consisting of complex geometries, a new database system, and workloads of a vast range of geospatial operations. These results emphasize the importance of an extensible benchmark framework, such as GeoYCSB, in evaluating geospatial NoSQL systems so that the most suitable data store can be chosen for the given application use cases.

Future work includes benchmarking various spatial databases beyond NoSQL, including relational systems, using GeoYCSB. Automating the GeoYCSB benchmarking process is another future work that will help boost the framework's usability and extensibility.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

[1] J.-G. Lee, M. Kang, Geospatial big data: challenges and opportunities, Big Data Res. 2 (2015) 74–81.
[2] E. Baralis, A.D. Valle, P. Garza, F. Scullino, SQL versus NoSQL databases for geospatial applications, in: Proceedings of 2017 IEEE International Conference on Big Data (Big Data), Boston, MA, USA, 2017, pp. 3388–3397.
[3] M. Duan, G. Chen, Assessment of MongoDB's spatial retrieval performance, in: Proceedings of International Conference on Geoinformaticss, Wuhan, China, 2015.
[4] M.B. Brahim, W. Drira, F. Filali, N. Hamdi, Spatial data extension for Cassandra NoSQL database, J. Big Data 3 (2016) 118–173.
[5] S. Agarwal, K. Rajan, Analyzing the performance of NoSQL vs. SQL databases for spatial and aggregate queries, in: Proceedings of Free and Open Source Software for Geospatial (FOSS4G), vol. 17, 2017, pp. 6–13.
[6] A. Makris, K. Tserpes, G. Spiliopoulos, D. Anagnostopoulos, Performance evaluation of MongoDB and PostgreSQL for spatio-temporal data, in: The Workshop Proceedings the EDBT/ICDT 2019 Joint Conference, Lisbon, Portugal, 2019.
[7] M. López, S. Couturier, J. López, Integration of NoSQL databases for analyzing spatial information in geographic information system, in: Proceedings of the 8th International Conference on Computational Intelligence and Communication Networks, CICN, 2016.
[8] B.F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears, Benchmarking cloud serving systems with YCSB, in: Proceedings of the 1st ACM Symposium on Cloud Computing, Indianapolis, Indiana, USA, 2010, pp. 143–154.
[9] T. Rabl, S. Gómez-Villamor, M. Sadoghi, V. Muntés-Mulero, H.-A. Jacobsen, S. Mankovskii, Solving big data challenges for enterprise application performance management, in: Proceedings of the VLDB Endowment, Istanbul, Turkey, 2012, pp. 1724–1735.
[10] J. Kuhlenkamp, M. Klems, O. Ross, Benchmarking scalability and elasticity of distributed database systems, in: Proceedings of the VLDB Endowment, Hangzhou, China, 2014, pp. 1219–1230.
[11] G. Haughian, R. Osman, W.J. Knottenbelt, Benchmarking replication in Cassandra and MongoDB NoSQL datastores, in: DEXA 2016. Lecture Notes in Computer Science, 2016, p. 9828.
[12] S. Kim, Y.S. Kanwar, GeoYCSB: a benchmark framework for the performance and scalability evaluation of NoSQL databases for geospatial workloads, in: 2019 IEEE International Conference on Big Data, Los Angeles, CA, USA, 2019, pp. 3666–3675.
[13] MongoDB geospatial queries, https://docs.mongodb.com/manual/geospatial-queries/.
[14] New Geo Features in MongoDB 2.4, https://www.mongodb.com/blog/post/new-geo-features-in-mongodb-24.
[15] GeospatialQueries, https://docs.couchbase.com/server/6.5/fts/fts-geospatial-queries.html.
[16] Couchbase views, https://docs.couchbase.com/server/6.0/learn/views/views-intro.html.
[17] A. Gyryk, YCSB-JSON: implementation for couchbase and MongoDB, https://blog.couchbase.com/ycsb-json-implementation-for-couchbase-and-mongodb/.
[18] Graffiti abatement incidents data set, https://catalog.data.gov/dataset/graffiti-abatement-incidents-bbbaf.
[19] Open Geospatial Consortium, http://www.opengeospatial.org/ogc.
[20] DB-engines ranking, https://db-engines.com/en/ranking.
[21] Building footprints dataset, https://catalog.data.gov/dataset/building-footprints-usgs-91e75.
[22] Tempe schools, https://www.tempe.gov/government/community-development/neighborhood-services/new-resident-directory/tempe-schools.
[23] C. Strobl, Dimensionally extended nine-intersection model (DE-9IM), in: Encyclopedia of GIS, Springer, Boston, MA, 2008, pp. 240–245.
[24] Japan's counties dataset, https://nlftp.mlit.go.jp/ksj/gml/datalist/ksjtmplt-n03-v2_4.html.
[25] Japan's routes dataset, https://nlftp.mlit.go.jp/ksj/gml/datalist/ksjtmplt-n07.html.
[26] QGIS, https://www.qgis.org/en/site/.
[27] Red Hat Ansible, https://www.ansible.com.
[28] Apache Accumulo, https://accumulo.apache.org/.
[29] GeoMesa index, https://www.geomesa.org/documentation/stable/user/datastores/index_overview.html.
[30] B. Simion, S. Ray, A.D. Brown, Surveying the landscape: an in-depth analysis of spatial database workloads, in: Proceedings of 2012 ACM SIGSPATIAL GIS, Redondo Beach, CA, USA, 2012, pp. 376–385.
[31] L. De Floriani, P. Marzano, E. Puppo, Spatial queries and data models, spatial information theory a theoretical basis for GIS, in: COSIT 1993, in: Lecture Notes in Computer Science, vol. 716, Springer, 1993, pp. 113–138.
[32] E. Clementini, Dimension-extended topological relationships, in: Encyclopedia of Database Systems, Springer, New York, NY, 2018, pp. 1115–1119.
[33] D. Guo, E. Onstein, State-of-the-art geospatial information processing in NoSQL databases, Int. J. Geo-Inf. 9 (5) (2020) 331, 1–20.
[34] S. Ray, B. Simion, A.D. Brown, Jackpine: a benchmark to evaluate spatial database performance, in: Proceedings of IEEE 27th International Conference on Data Engineering, Hannover, Germany, 2011, pp. 1139–1150.
[35] G. Garbis, K. Kyzirakos, M. Koubarakis, Geographica: a benchmark for geospatial RDF stores, in: Proceedings of International Semantic Web Conference, ISWC 2013, 2013, pp. 343–359.
[36] M.M. Alam, S. Ray, V.C. Bhavsar, A performance study of big spatial data systems, in: Proceedings of ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data, BigSpatial 2018, Seattle, WA, USA, 2018.