The Journal of Supercomputing (2022) 78:2556–2579
https://doi.org/10.1007/s11227-021-03946-7

A comparative experimental study of distributed storage engines for big spatial data processing using GeoSpark

Hansub Shin[1] · **Kisung Lee**[2] · **Hyuk-Yoon Kwon**[1]

Accepted: 29 May 2021 / Published online: 1 July 2021

Abstract

With increasing numbers of GPS-equipped mobile devices, we are witnessing a deluge of spatial information that needs to be effectively and efficiently managed. Even though there are several distributed spatial data processing systems such as GeoSpark (Apache Sedona), the effects of underlying storage engines have not been well studied for spatial data processing. In this paper, we evaluate the performance of various distributed storage engines for processing large-scale spatial data using GeoSpark, a state-of-the-art distributed spatial data processing system running on top of Apache Spark. For our performance evaluation, we choose three distributed storage engines having different characteristics: (1) HDFS, (2) MongoDB, and (3) Amazon S3. To conduct our experimental study on a real cloud computing environment, we utilize Amazon EMR instances (up to 6 instances) for distributed spatial data processing. For the evaluation of big spatial data processing, we generate data sets considering four kinds of various data distributions and various data sizes up to one billion point records (38.5GB raw size). Through the extensive experiments, we measure the processing time of storage engines with the following variations: (1) sharding strategies in MongoDB, (2) caching effects, (3) data distributions, (4) data set sizes, (5) the number of running executors and storage nodes, and (6) the selectivity of queries. The major points observed from the experiments are summarized as follows. (1) The overall performance of MongoDB-based GeoSpark is degraded compared to HDFS- and S3-based GeoSpark in our experimental settings. (2) The performance of MongoDB-based GeoSpark is relatively improved in large-scale data sets compared to the others. (3) HDFS- and S3-based GeoSpark are more scalable to running executors and storage nodes compared to MongoDB-based GeoSpark. (4) The sharding strategy based on the spatial proximity significantly improves the performance of MongoDB-based GeoSpark. (5) S3- and HDFS-based GeoSpark show similar performances in all the environmental settings. (6) Caching

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2021R1F1A1064050). This research was also supported by National Science Foundation (Grant No. 2032745).

Extended author information available on the last page of the article

in distributed environments improves the overall performance of spatial data processing. These results can be usefully utilized in decision-making of choosing the most adequate storage engine for big spatial data processing in a target distributed environment.

Keywords Spatial data · GeoSpark · Comparative study · Distributed storage engines · Query performance

1 Introduction

With the widespread use of GPS-equipped mobile devices such as smartphones, smartwatches, and connected vehicles, we are witnessing an explosion of spatial data in our everyday lives [1–3]. For example, online social network users can tag their fine-grained location to each post (e.g., tweets on Twitter), and trajectory information of an individual based on the locations of cell towers and credit card (and/or transportation card) transactions is being used for contact tracing during the pandemic such as COVID-19. With a large number of location-based applications and services, the scale of real-world spatial data is growing at an unprecedented rate. For example, Uber reported that there were 15 million trips per day on its ride-sharing platform in 2017 [4]. In addition, airborne and satellite remote sensing platforms are collecting a huge amount of location-aware data every day. For example, on the NASA's Earth Observing System Data and Information System (EOSDIS), its total data size in 2017 was 22 petabytes (PB) and expected to increase to about 250 PB by 2025.

It is challenging to efficiently and effectively store and process such massive amounts of spatial data because of the inherent complexity of spatial data processing. For example, most spatial data processing is based on spatial proximity. Without location-aware partitioning, a huge amount of irrelevant data can be evaluated, decreasing the overall query processing performance and wasting our computing resources. Another challenge for spatial data processing is heterogeneous spatial data from various spatial data formats. To address the big spatial data challenges, several distributed systems have been developed on top of a general-purpose big data framework providing MapReduce-like computation models, such as Apache Hadoop and Spark. Hadoop-based systems include Hadoop GIS [5] and SpatialHadoop [6], and there are several Spark-based systems such as GeoSpark (Apache Sedona) [7], LocationSpark [8], SparkGIS [9], Simba [10], and Magellan [11]. Such systems generally extend Hadoop or Spark to represent spatial objects and support spatial query processing. They often build a spatial index on each data partition to expedite spatial data processing. Even though such systems have been evaluated for large-scale spatial data processing in recent studies [12], the effects of different storage engines in distributed environments have not been well studied.

In this paper, we evaluate the performance of various distributed storage engines for large-scale spatial data processing using a distributed spatial data processing system. For our evaluation, we choose GeoSpark as the spatial data processing system

because it demonstrates superior performance for large-scale spatial data processing [12]. There are several options for storing large-scale spatial data, but their performance effects in spatial data processing (e.g., processing using GeoSpark) have not been well studied in the literature. In cloud computing environments (e.g., AWS), we need to choose the best storage option to optimize our spatial data processing, but it would be hard to select one without detailed performance metrics on a real cluster. To address the current limitations, we evaluate three representative storage systems (a distributed file system, a NoSQL store, and an object store) and settings on AWS in this paper.

We first select HDFS as a distributed storage engine because this would be a logical solution to get benefits from data locality by colocating the Spark cluster with the distributed file system. Compared to remote storage systems, we intuitively expect some performance improvement because we can minimize the data movement over the network, but we need to experimentally measure the benefits on a real cluster to validate (or invalidate) the claim. In this paper, we report some interesting results that are not intuitive based on data locality.

Our second choice is MongoDB, a representative NoSQL store. MongoDB is a reasonable solution to store spatial data because it provides built-in spatial indices and query processing. Even though we can run basic spatial queries directly on MongoDB, we need to utilize a specialized distributed system for large-scale spatial data processing (e.g., GeoSpark). Even though GeoSpark on top of MongoDB can be a logical solution for both small- and large-scale spatial processing, we need to experimentally measure the effects of using the disk-based NoSQL store on a real cluster. We report the performance issues of MongoDB when we use it as a storage engine for GeoSpark in this paper.

Our last choice is Amazon S3, a representative cloud-based object store. Most cloud providers offer S3-like persistent data storage as the most cost-efficient way of storing large-scale data. Despite the benefits of cloud-based object stores such as fault tolerance and scalability, there are conflicting reports about their performance for large-scale data processing. For example, the official Spark website[1] states that "Reading and writing data can be significantly slower than working with a normal filesystem." while a performance study insists that S3 is better than HDFS on performance per dollar.[2] In this paper, we report that the performance (not performance per dollar) of S3 is comparable to that of HDFS for certain workloads, which was previously not reported for Spark-based processing to the best of our knowledge.

To evaluate the different storage engines in real cloud computing settings, we design and configure our evaluation environments using Amazon EMR clusters. For the evaluation of big spatial data processing, we generate data sets considering four kinds of data distributions, which include a distribution in the real-world data set, and various data sizes up to one billion point records (38.5GB raw size). Through the extensive experiments, we measure the processing time on the storage engines with the following variations: (1) sharding strategies in MongoDB, (2) caching

[1] https://spark.apache.org/docs/2.3.0/cloud-integration.html.
[2] https://databricks.com/blog/2017/05/31/top-5-reasons-for-choosing-s3-over-hdfs.html.

effects, (3) data distributions, (4) data set sizes, (5) the number of running executors and storage nodes, and (6) the selectivity of queries.

We make the following observations from the experiments.

1. The overall performance of MongoDB-based GeoSpark is degraded compared to HDFS- and S3-based GeoSpark in our environmental settings. For example, in the experiment where the data distribution follows the real-world data set[3] and the data set has 10 million records, the performance of MongoDB-based GeoSpark is degraded by 2.08 ~ 7.84 times compared to HDFS-based GeoSpark and by 2.44 ~ 8.91 times compared to S3-based GeoSpark with various query types.
2. The performance of MongoDB-based GeoSpark is relatively improved in large-scale data sets compared to the others. The performance degradation of MongoDB-based GeoSpark becomes reduced from by 3.51 ~ 9.40 times to by 1.23 ~ 1.96 times compared to HDFS-based GeoSpark when the data set size is varied from 1 million to 1 billion records; from by 3.04 ~ 10.33 times to 1.27 ~ 1.92 times compared to S3-based GeoSpark.
3. HDFS- and S3-based GeoSpark are more scalable to running executors and storage nodes compared to MongoDB-based GeoSpark. The performance of HDFS-based GeoSpark increases by 1.55 ~ 2.65 times as running executors and storage nodes increase; S3-based GeoSpark by 1.84 ~ 2.65 times; however, MongoDB-based GeoSpark only by 1.09 ~ 1.48 times.
4. The sharding strategy based on the spatial proximity significantly improves the performance of MongoDB-based GeoSpark. Data sharding according to the z order, which is one of space-filling curves, improves the performance of MongoDB by 1.84 ~ 3.97 times compared to random data sharding with various query types.
5. S3- and HDFS-based GeoSpark show similar performances in all the environmental settings.
6. Caching in distributed environments improves the overall performance of spatial data processing. Caching in MongoDB improves the performance by 2.10 ~ 15.93 % compared to the case where the data and indexes do not reside in the memory; HDFS-based GeoSpark by 31.70 ~ 64.43 %; S3-based GeoSpark by 16.72 ~ 67.71 %.

In Sect. 2, we describe large-scale spatial data management and storage systems as the background of the paper. Specifically, we explain GeoSpark and three underlying storage engines: HDFS, MongoDB, and Amazon S3. In Sect. 3, we present a design of distributed experimental environments for big spatial data management systems using GeoSpark. In Sect. 4, we define our data and query sets for large-scale spatial data processing and report our evaluation results. We summarize the related work in Sect. 5 and conclude this paper in Sect. 6.

[3] http://www.naturalearthdata.com/downloads/10m-cultural-vectors/10m-populated-places/

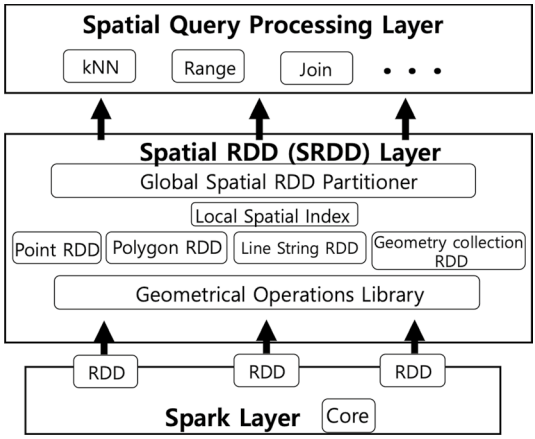


Fig. 1 Architecture of GeoSpark [7]

2 Background

In this section, we describe the design concepts and features of GeoSpark and three storage engines we evaluate in this study.

2.1 GeoSpark

GeoSpark (Apache Sedona) is a distributed spatial data processing engine running on top of Apache Spark. As input formats for spatial data, GeoSpark supports various file formats including ESRI Shapefile, GeoJSON, well-known text (WKT), well-known binary (WKB), and NetCDF. GeoSpark can process several geometry types such as points, polygons, and rectangles. To process spatial data, GeoSpark extends Spark resilient distributed data sets (RDDs), the fundamental unit of data in Spark, for spatial data, called Spatial RDDs. GeoSpark supports various Spatial RDD types, such as Point RDDs, Polygon RDDs, and Rectangle RDDs, with compact in-memory representations. We can perform spatial data processing (e.g., spatial range, *k*-NN, and join queries) using the Spatial RDD API or a structured API called GeoSpark SQL [13]. Like other Spark RDDs, data in Spatial RDDs can be cached in the main memory of the cluster for fast accesses. Figure 1 shows the overall architecture of GeoSpark. In a recent benchmark study, GeoSpark exhibits the best performance in most cases [12].

To partition spatial objects by their spatial proximity, GeoSpark supports global spatial data partitioning. Its basic idea is that GeoSpark randomly samples the RDD and builds an index (e.g., KD-Tree, R-Tree, Quad-Tree) on the sample. Next, it uses the leaf nodes of the index for repartitioning the RDD by spatial proximity. The global spatial data partitioning incurs a data shuffle (i.e., wide dependency in the RDD transformation). GeoSpark also supports local spatial indexing on each Spatial RDD partition using traditional spatial indexing techniques such as R-Tree and Quad-Tree. Unlike the global index structure built using sampled data, the local index structure is built on all spatial objects in the partition.

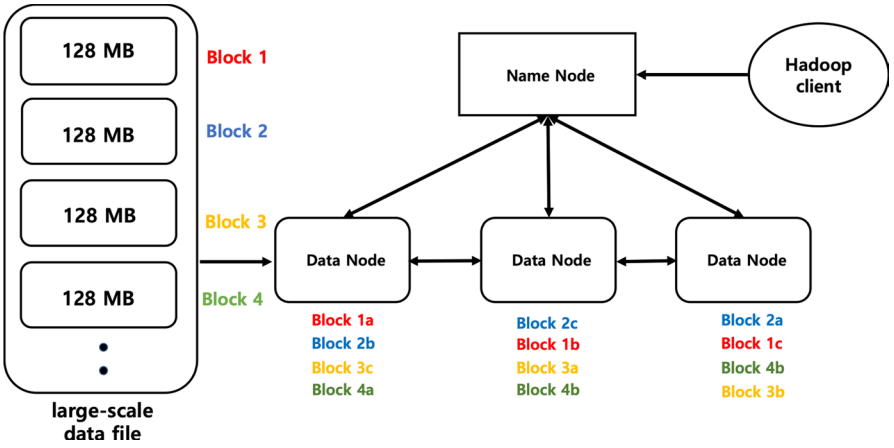


Fig. 2 Architecture of HDFS

Like Spark, GeoSpark can be used with various storage engines that store raw spatial data. For example, we can read spatial data from distributed file systems (e.g., HDFS), NoSQL stores (e.g., MongoDB, HBase), and cloud-based object storage services (e.g., Amazon S3). Even though GeoSpark can access spatial data stored in various types of storage engines having different characteristics, the effects of underlying storage engines in terms of spatial data processing performance were not well studied. In this study, we evaluate three systems having different characteristics as underlying storage engines for GeoSpark: (1) HDFS, (2) MongoDB, and (3) Amazon S3.

2.2 HDFS

Hadoop Distributed File System (HDFS) is the representative distributed file system designed to run on multiple commodity computers on a cluster [14]. Because it is designed as an underlying file system for Hadoop ecosystems, it has been also widely used for Apache Spark [15]. Figure 2 shows the overall architecture of HDFS. A large-scale file is partitioned into multiple data blocks, and each data block is replicated on multiple nodes. By default, HDFS splits the entire data file in the unit of 128 MB and replicates it with a replication factor of 3. Based on the master–slave architecture, a typical HDFS cluster configuration consists of one active name node and multiple data nodes. The name node manages the metadata that describes which data node stores which data blocks and sends a list of data nodes storing the requested data blocks to the client. The data node stores actual data blocks that are partitioned from the entire data file and provides the requested data blocks to the client. To efficiently process the data blocks stored in HDFS, we need to consider data distribution stored over the cluster when assigning a given request to the nodes. Apache Spark on HDFS tries to utilize the data locality of HDFS, i.e., assigning each task into the data node storing the data block required for the task, for minimizing the network communication between nodes [16, 17].

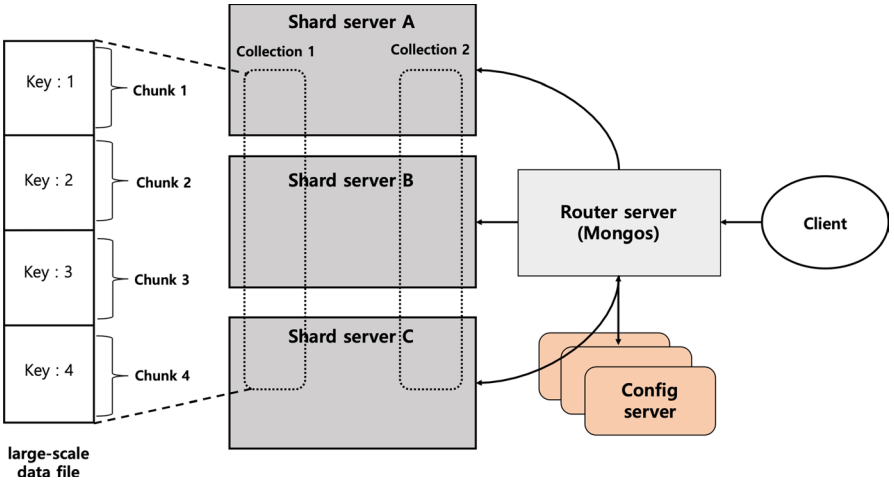


Fig. 3 Architecture of MongoDB [18]

HDFS has the following strengths as a storage engine to manage large-scale data files in distributed environments. First, it provides the fault tolerance for each data block by replicating the same data block in multiple nodes. Second, it is scalable to newly added data nodes because it automatically redistributes each data block over the cluster considering the added data nodes.

2.3 MongoDB

MongoDB is one of the most widely used NoSQL stores and is categorized as a document store. It can effectively store hierarchical documents such as JSON and XML formats. Unlike relational database management systems that require strict schemas for storing data, we can import document data into MongoDB without defining any schema. To improve the efficiency of search operations, MongoDB supports various index types based on a B-tree index structure.

When we store the entire data files in MongoDB on distributed environments across multiple machines, we need to consider how to partition the data files into multiple machines (i.e., *data sharding*). Figure 3 shows the architecture of MongoDB. MongoDB consists of three components: (1) Config server, (2) Router server (i.e., Mongos), and (3) Shard server. The Config server manages the metadata storing the information about which data shards are stored in which shard servers. The Router server routes the client requests to the shard server using the metadata stored in the Config server. Each shard server stores a data shard, which is partitioned from the entire data sets and is replicated over the cluster where the default replication factor is 3. The most important factor to the performance in data sharding of MongoDB is sharding strategies. There are two sharding strategies:[4] (1) hash-based sharding (i.e., partitioning the data

[4] https://docs.mongodb.com/manual/sharding/

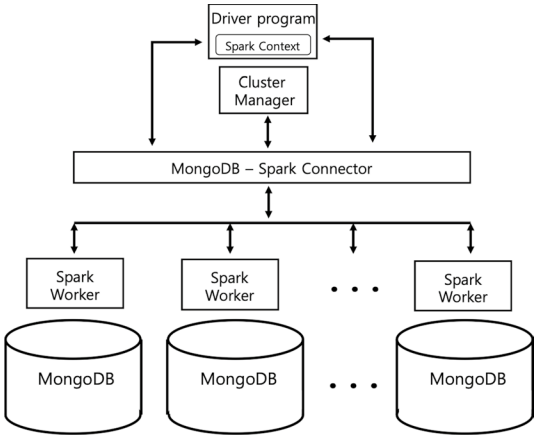


Fig. 4 Architecture of MongoDB-based GeoSpark [20]

evenly over the cluster, but losing data similarity in the same shard server) and (2) range-based sharding (i.e., keeping data similarity in the same shard server, but resulting in a biased data distribution between the machines in the cluster). In this paper, since we deal with spatial data, we need to consider a sharding strategy based on spatial proximity of data. As the preliminary evaluation in Sect. 4.3.1, we will discuss the performance variation according to two different sharding strategies: (1) random sharding and (2) sharding by the spatial proximity of data.

MongoDB also provides spatial query processing based on built-in spatial indexing structures and processing algorithms. MongoDB provides two spatial index types (*2d* and *2dsphere*) for efficiently processing spatial queries such as *within*, *intersects*, and *near* operators. For spatial data formats, it supports not only lat/long coordinate pairs but also various GeoJSON objects such as Point, LineString, and Polygon. An evaluation study reports that MongoDB is more efficient than PostGIS, an extension of PostgreSQL for spatial data management, for certain spatial queries [19].

To support MongoDB on Spark, MongoDB provides a MongoDB connector for Spark. Using this connector, we can read data from MongoDB, as the data storage engine, to Spark, as the data processing engine, and write data from Spark to MongoDB. The connector also supports aggregation pipelines that can filter data and perform aggregations in MongoDB, instead of loading the entire data in Spark, reducing the amount of data transferred from MongoDB to Spark. As the representative NoSQL store for an underlying storage engine of GeoSpark, we evaluate the performance of MongoDB. Figure 4 shows our evaluation setting to use distributed MongoDB as a storage engine for GeoSpark. It is worth noting that, even though MongoDB directly provides its MapReduce computation operations, we do not evaluate them in this paper because our focus is the evaluation of underlying storage engines using GeoSpark.

2.4 Amazon S3

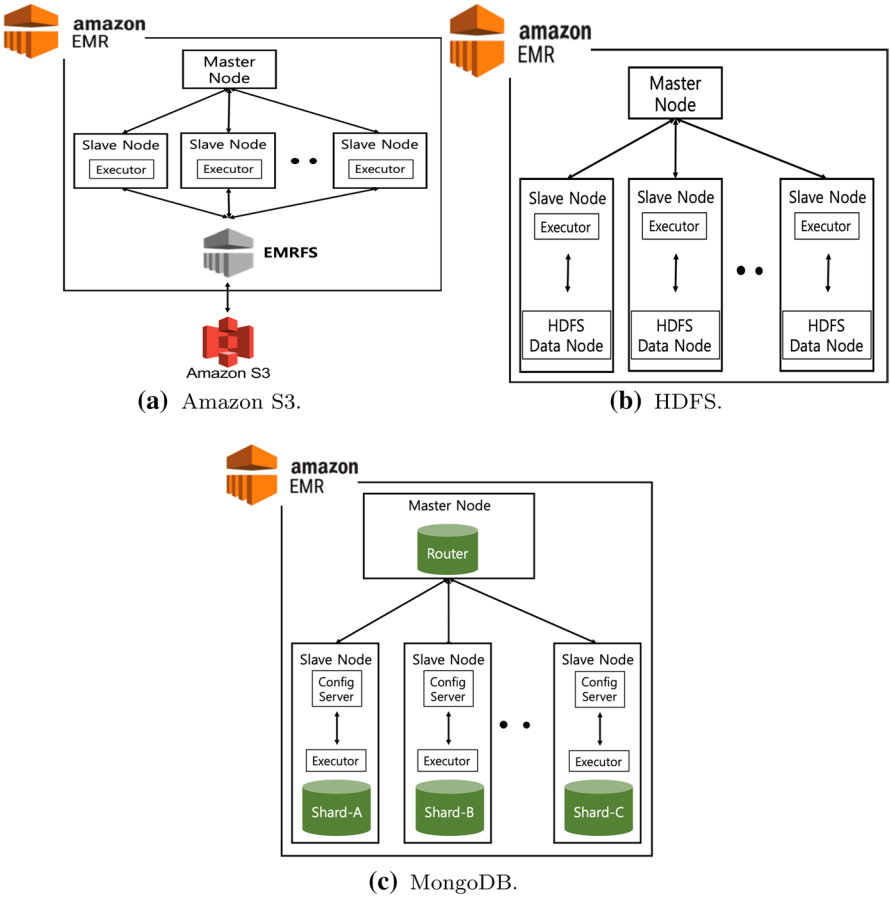
With the prevalence of cloud services such as Amazon Web Services (AWS), Google Cloud Platforms (GCP), and Microsoft Azure, they also provide the scalable storage based on cloud environments. The representative cloud-based storage engines are Amazon S3,[5] Google Cloud Storage,[6] and Azure Storage.[7] In this paper, we consider Amazon S3 as the cloud-based storage engine for spatial data processing. Amazon S3 provides a scalable distributed storage and manages the data based on the unit of the object and the bucket. The object contains a data file and its associated metadata. The object cannot be relocated in a distributed environment, and each object has the corresponding URI so that we can access it through the SOAP or REST-based interface. The bucket is a container to store multiple objects. It has been known that Amazon S3 follows a design principle of Amazon Dynamo [21] even if the detailed implementations are not disclosed. As a result, it provides scalability, fault tolerance, and availability for storing large-scale data.

3 Design of cloud-based evaluation environments for distributed storage engines using GeoSpark

In this section, we describe how we design our distributed computing environments to evaluate the performance of different storage engines for large-scale spatial data processing. Figure 5 shows the overall framework for distributed environments using three different storage engines: HDFS, MongoDB, and Amazon S3. Here, we fix the processing engine as GeoSpark and use Amazon Elastic MapReduce (Amazon EMR) as a cloud-based platform that supports various open-source big data tools including Apache Spark, for all the designs in common. In this architecture, the master node submits a job to the slave nodes; each slave node accesses to three types of storage engines and sends the results of the job into the master node.

We configure multiple EMR clusters to run GeoSpark for distributed processing of spatial data by using different storage engines. Figure 5a shows a design for using Amazon S3 as the storage engine. Because we cannot configure the number of nodes used for Amazon S3, we directly connect from GeoSpark to Amazon S3 in EMR. To configure Amazon S3 as a storage engine for spatial data, we utilize EMR File System (EMRFS) that is an implementation of HDFS extending the ability to directly access data from Amazon EMR to Amazon S3. Figure 5b shows a design for using HDFS. We run HDFS on EMR and vary the number of data nodes from 2 to 6 to check the performance variation in terms of

[5] https://aws.amazon.com/s3
[6] https://cloud.google.com/storage/
[7] https://docs.microsoft.com/azure/storage/



(a) Amazon S3. **(b)** HDFS. **(c)** MongoDB.

Fig. 5 Overall framework for distributed environments

the scalable storage. Figure 5c shows a design for using MongoDB. We deploy sharded MongoDB clusters by increasing the number of Shard servers from 2 to 6. We run one Config server in each slave and replicate the metadata in Config server in all the slaves.

We use EMR version 5.12.1 that is configured with Apache Spark 2.2.1, GeoSpark 1.2.0, MongoDB 4.4, YARN-based Ganglia 3.7.2, Zeppelin 0.7.3., and Hadoop 2.8.3. We use a master node in EMR equipped with 16 vCPUs and 30 GB of the memory for all the distributed storage engines. It is communicated with slave nodes in EMR. Each slave node is equipped with 8 vCPUs and 30 GB of the memory.

Table 1 Characteristic of the spatial data sets used in the experiments

Data sets	Description	Number of points	Size of data
Natural Earth	Points of populated places	7343	5MB
$RealDist_{1M}$	Real-world data distribution	1,000,000	38.5MB
$RealDist_{10M}$	Real-world data distribution	10,000,000	385.6MB
$RealDist_{100M}$	Real-world data distribution	100,000,000	3.85GB
$RealDist_{1B}$	Real-world data distribution	1,000,000,000	38.5GB
$ExponentialDist_{10M}$	Exponential distribution	10,000,000	385.6MB
$UniformDist_{10M}$	Uniform distribution	10,000,000	385.6MB
$NormalDist_{10M}$	Normal distribution	10,000,000	385.6MB

4 Comparative performance evaluation

4.1 Experimental data and query sets

4.1.1 Data sets

Table 1 shows the characteristics of the spatial data sets used in the experiments. We generate multiple data sets for various distributions of the data set, in particular, including the distribution of the real-world data set, and for various sizes of the data set.

As the real-world data set, we use Natural Earth data.[8] The used data set consists of points of interest representing populated places in the map such as capitals, major cities, and towns, plus a sampling of smaller towns in sparsely inhabited regions. It consists of 7343 points, and the total size of data is 5MB. This data set is used to generate four synthetic data sets in Table 1 based on the real-world data distribution.

For generating data sets, we consider four types of distributions: (1) exponential, (2) uniform, (3) normal, and (4) real-world data distributions. Gunther et al. have proposed to generate a data set based on uniform, normal, and exponential distributions [22]. Based on their framework, we add a real-world data distribution. The specific method of the data generation for real-world data distribution is as follows. First, we divide the entire data range of the *Natural Earth* data set into 30x30 grids. Then, we calculate the proportion of the points belonging to each grid and generate points in each grid according to the calculated proportion considering a target distribution. Specifically, by considering the covariance between (x, y) coordinates of the real-world data, we generate multivariate normal and exponential data distribution using the Python numpy library.[9] In addition, we generate uniform data distribution by randomly generating spatial points in each grid. As a result, we generate four types of data sets with various data distributions as the number of points is fixed as

[8] http://www.naturalearthdata.com/downloads/10m-cultural-vectors/10m-populated-places/.
[9] https://numpy.org/

Table 2 Characteristic of the queries used in the experiments (default values are represented in bold)

Queries	Description
k-NN queries	Find the nearest k points to a query point (k = 10, **100**, 1000, 100,000)
Circle queries	Find all the points within a circle having a radius r around a query point (r = 10KM, **100KM**, 1000KM)
Bounding box queries	Find all the points within a bounding box (10KM × 10KM, **100KM** × **100KM**, 1000KM × 1000KM)
Distance join queries	Find all the pairs from two spatial data sets A and B where the distance of each pair is less or equal than a threshold (0.1KM, **1KM**, 10KM). Here, A is a variable data set and B is a fixed data set consisting of 100 points

10 million. We also generate four types of data sets with various data set sizes, i.e., 1 million, 10 million, 100 million, and 1 billion, as the data distribution is fixed as the real-world data distribution. Existing studies for evaluating the performance of geographic information systems have usually used between 5M and 200M numbers of POI (Point of Interest) data sets [12, 19, 23, 24] and the total size of data is ranged from 1.1GB to 11GB data when dealing with other types of spatial data such as polygon and line [19, 23–25]. Therefore, 1 billion of spatial points with 38.5GB is considered a sufficiently large-scale spatial data set.

4.1.2 Query sets

Table 2 shows four types of queries used in the experiments. The circle and bounding box queries are included in basic relations (i.e., Within and Contains) defined by dimensionally extended nine-intersection model (DE-9IM) [26]. We additionally define the k-NN query [27], which is useful to find the closest places to a given query point in Geographic Information Systems (GIS), and the distance join query [28], which leverages the effect of the map–reduce framework [29]. Here, we use arbitrary query points and regions for each query.

4.2 Experimental methods

In the experiments, we compare the performance of distributed storage engines using GeoSpark, i.e., HDFS-, MongoDB-, and S3-based GeoSpark, from multiple perspectives. As the preliminary evaluation, we conduct the following two evaluations: (1) the performance variation of MongoDB according to different sharding strategies and (2) the performance evaluation to check the caching effects of all the storage engines in distributed environments. Then, we measure the time of the spatial data processing based on distributed storage engines according to the following variations: (1) data distribution, (2) the size of data set, (3) the number of storage nodes and running executors in GeoSpark, and (4) the selectivity of the query.

The default settings for the experiments are as follows. We fix the number of data nodes in HDFS (or Shard servers in MongoDB) as 6 where one executor is running for each data node (or Shard server). We run a total of 6 executors in S3-based

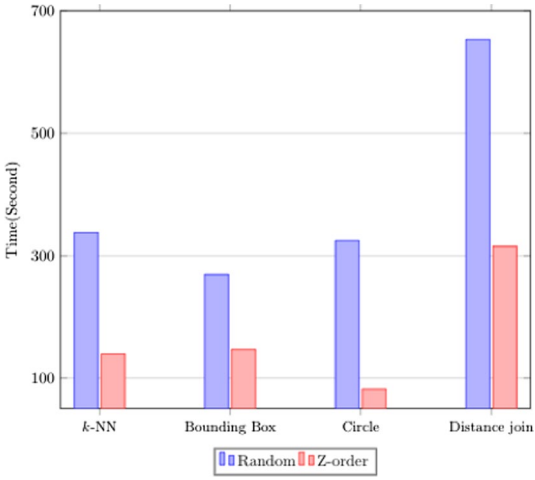


Fig. 6 Performance of MongoDB according to different sharding strategies

GeoSpark for the sake of fairness even though we cannot control the number of data nodes in S3. We use $RealDist_{10M}$ as the default data set and use four kinds of query sets defined in Table 2 for each experiment. For all the experiments, we conduct each query 100 times and obtain its average processing time.

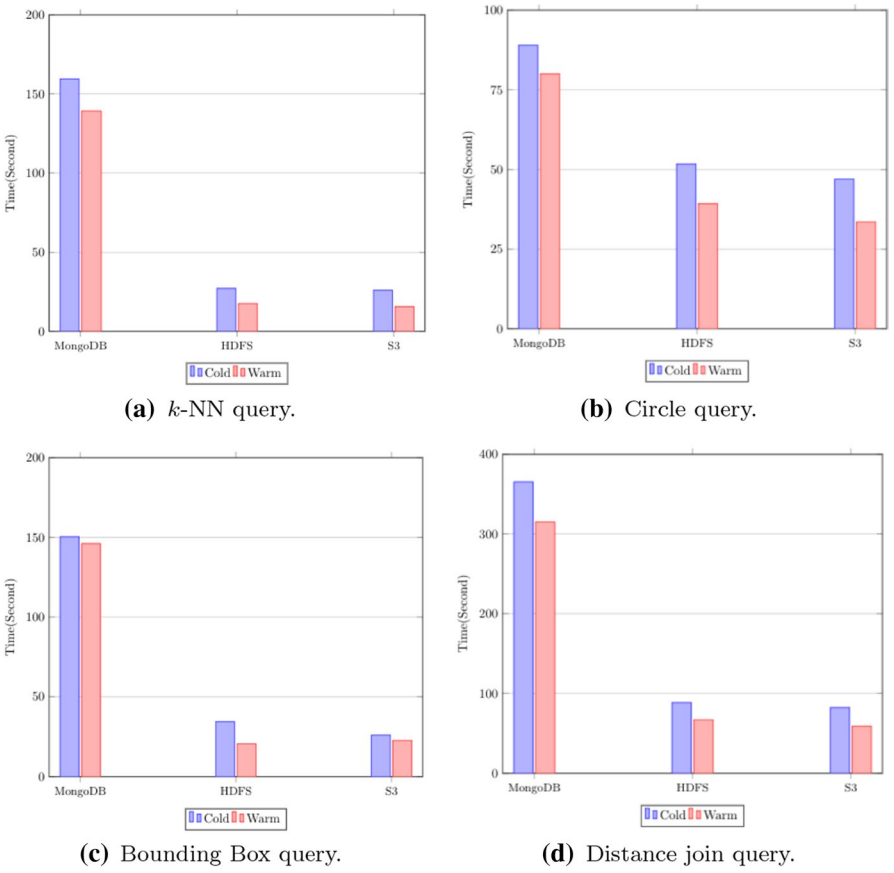
We use GeoSpark SQL [13] to run each kind of queries. That is, we represent the query in the SQL form for GeoSpark SQL. Then, GeoSpark SQL transforms the query as a set of operations for the map–reduce framework. For all the distributed storage engines, we commonly use the same global and local indexes: R-tree for the global index and Quad-Tree for the local index.

To check the caching effects of the systems, we construct environments according to the maintenance of caching. The reason for this experiment is to observe the performance of the storage systems for the environments with an initial state or limited memory that do not utilize the caching effect fully. First, *cold cache* is an environment where data and indexes do not reside in the memory, i.e., queries have never been executed. Second, *warm cache* is an environment where data and indexes reside in the memory by executing the same query prior to conducting the experiment. To make the cold cache environment, we execute a Linux command "drop caches" before each query is performed on both master and slave nodes. This command frees page caches in the Linux system [30]. We observe the caching effect of the storage system in Sect. 4.3.2.

4.3 Experimental results

4.3.1 Data sharding in MongoDB

The sharding strategy of MongoDB significantly affects the performance because it partitions the entire data sets into multiple Shard servers according to the shard key. However, since built-in spatial indexes supported in MongoDB, i.e., 2d and



(a) k-NN query. **(b)** Circle query.

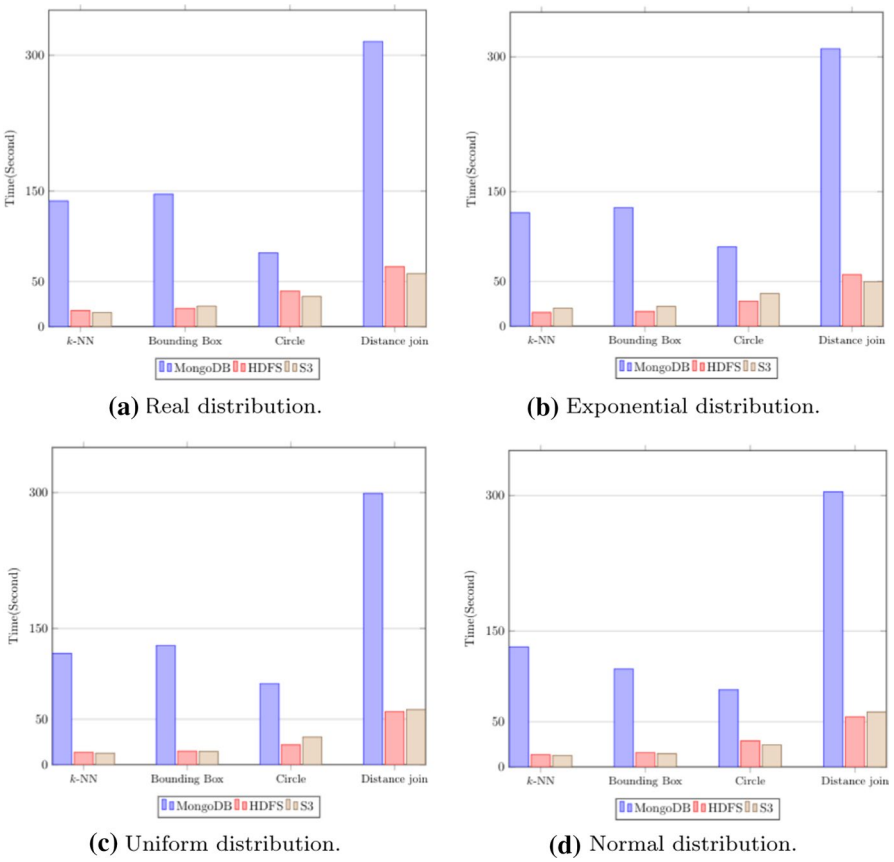
(c) Bounding Box query. **(d)** Distance join query.

Fig. 7 Performance of HDFS-, MongoDB-, and S3-based GeoSpark according to the caching effects

2dsphere, cannot be used for a shard key,[10] we need to investigate an effective sharding strategy for MongoDB. Zhang et al. have used the Hilbert curve, one of the space-filling curves, and k-means clustering to leverage the spatial proximity of data [31]. In this paper, we consider two kinds of sharding strategies: (1) random sharing and (2) sharding according to z order [32], one of the space-filling curves. The calculation of z-values and obtaining z order according to the z-values are performed once before data sharding. Thus, it does not affect the query performance.

Figure 6 shows an experimental result of MongoDB according to different sharding strategies. The result shows that sharding according to z order significantly improves random sharding in all the kinds of queries. Specifically, the processing time of sharding according to z order is reduced compared to random sharding by 2.42 times in the k-NN query, by 1.84 times in the bounding box query, by 3.97

[10] https://docs.mongodb.com/manual/core/2dsphere.



(a) Real distribution. **(b)** Exponential distribution.

(c) Uniform distribution. **(d)** Normal distribution.

Fig. 8 Performance of HDFS-, MongoDB-, and S3-based GeoSpark as the distribution of data set is varied

times in the circle query, and by 2.07 times in the distance join query, respectively. This result indicates that data sharding according to z order clusters the entire data into multiple Shard servers effectively by maintaining the spatial proximity of data. As a result, we use the z order as the shard key for MongoDB in the remaining experiments.

4.3.2 Caching effects of distributed storage engines

Figure 7 shows the performance of MongoDB-, HDFS-, and S3-based GeoSpark in the cold cache and warm cache environments. We indicate that the performance in the warm cache environment is improved compared to the cold cache environment for all the distributed storage engines. Specifically, MongoDB-based GeoSpark in warm cache improves the performance in cold cache by 2.10 ~ 15.93 %; HDFS-based GeoSpark by 31.70 ~ 64.43 %; and S3-based GeoSpark by 16.72 ~ 67.71

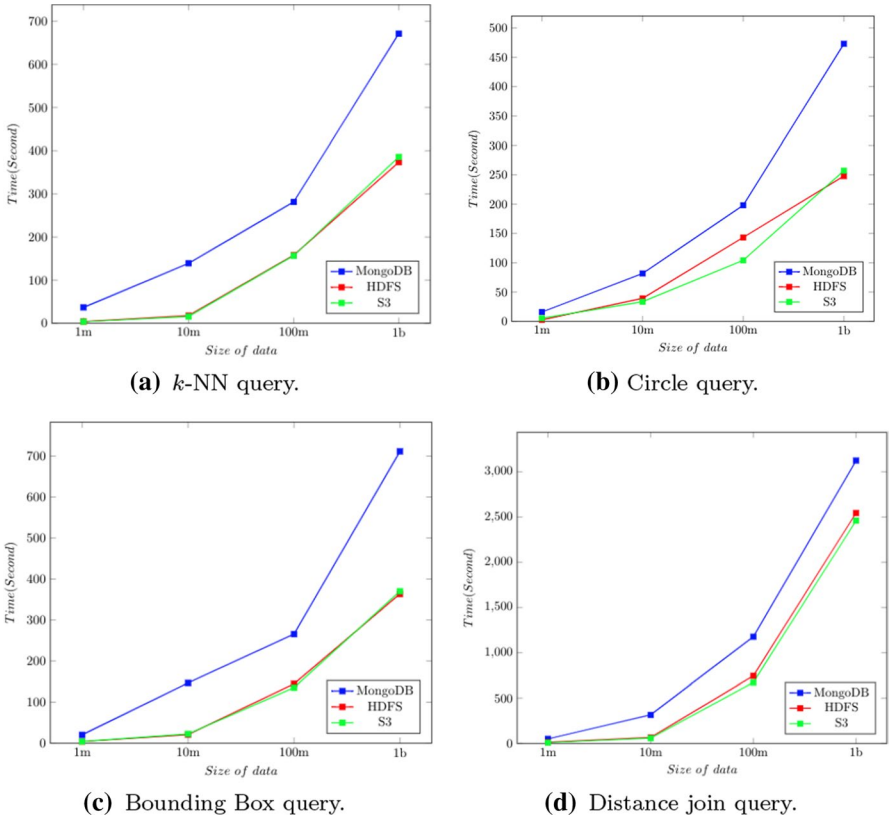
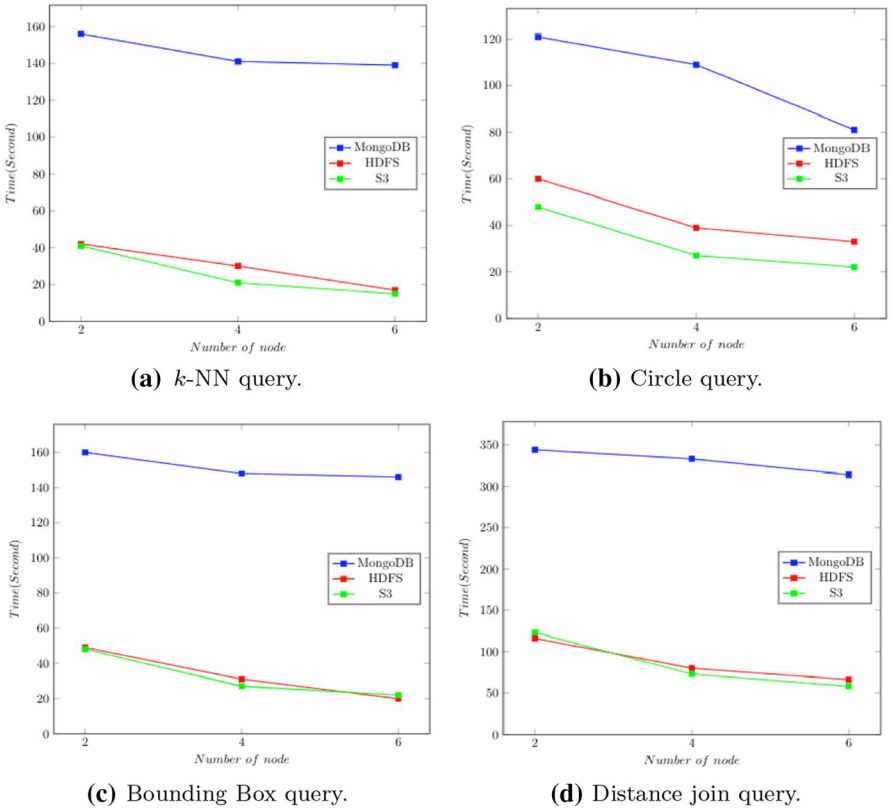


Fig. 9 Performance of HDFS- and MongoDB-, and S3-based GeoSpark as the size of data set is varied

%. This result shows the caching effects of the used storage engines in distributed environments. As a result, we use the warm cache environment in the remaining experiments.

4.3.3 The performance as the data distribution is varied

Figure 8 shows the performance of HDFS-, MongoDB-, and S3-based GeoSpark as the data distribution is varied. We observe that the performances of both HDFS- and S3-based GeoSpark are better than MongoDB-based GeoSpark for all the distributions. Specifically, for k-NN queries, HDFS-based GeoSpark shows a better performance than MongoDB-based GeoSpark by 7.84 ~ 9.89 times with a variety of data distribution, and S3-based GeoSpark shows a better performance by 6.30 ~ 10.39 times. For bounding box queries, HDFS-based GeoSpark shows a better performance by 6.84 ~ 8.86 times, and S3-based GeoSpark shows a better performance by 5.98 ~ 8.99 times. For circle queries, HDFS-based GeoSpark shows a better performance by 2.08 ~ 3.04 times, and S3-based GeoSpark shows a better performance by



(a) k-NN query. **(b)** Circle query.

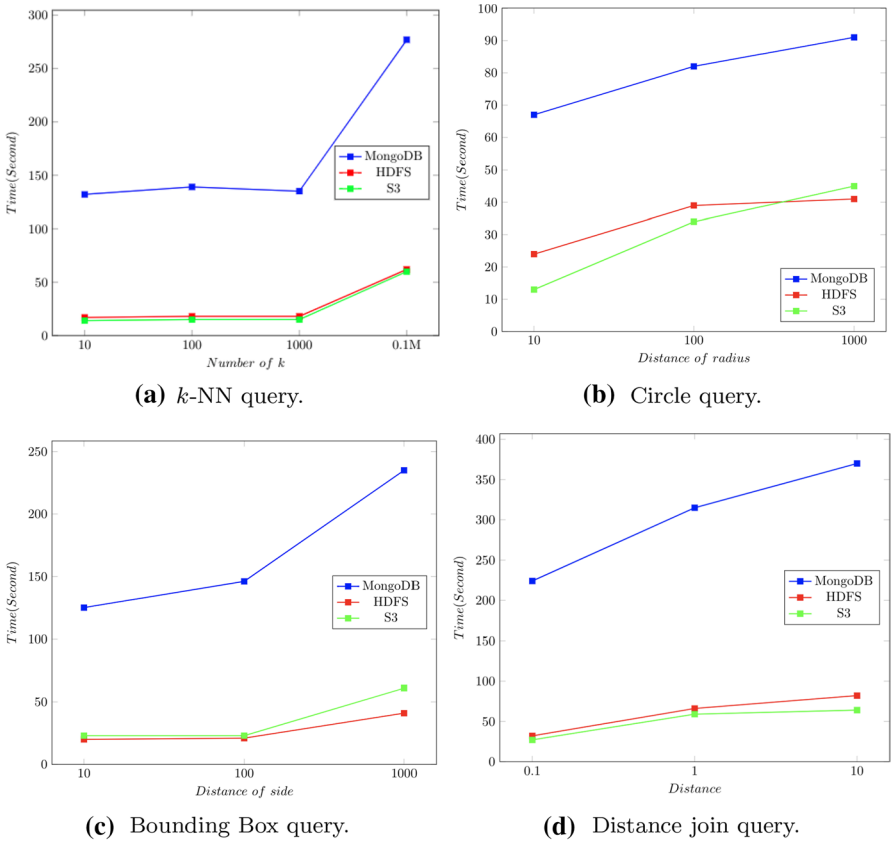
(c) Bounding Box query. **(d)** Distance join query.

Fig. 10 Performance of HDFS- and MongoDB-, and S3-based GeoSpark as the total number of executors in GeoSpark is varied

2.42 ~ 3.47 times. For distance join queries, HDFS-based GeoSpark shows a better performance by 4.75 ~ 5.44 times; S3-based GeoSpark shows a better performance by 4.95 ~ 6.22 times. We also note that the overall query performance of S3-based GeoSpark is comparable to that of HDFS-based GeoSpark because it extends the HDFS implementation tailored to Amazon EMR.

4.3.4 The performance as the size of data set is varied

Figure 9 shows the performance of HDFS-, MongoDB-, and S3-based GeoSpark as the size of the data set is varied from 1 million to 1 billion objects. Overall, we observe that the performances of both HDFS- and S3-based GeoSpark are better than MongoDB-based GeoSpark. However, the performance gap between them is reduced as the size of the data set increases. Specifically, the performance of MongoDB-based GeoSpark is degraded by 3.51 ~ 9.40 times compared to HDFS-based GeoSpark and by 3.04 ~ 10.33 times compared to S3-based GeoSpark, respectively, in the case of $RealDist_{1M}$. However, it is degraded only by 1.23 ~ 1.96 times



(a) k-NN query. **(b)** Circle query.

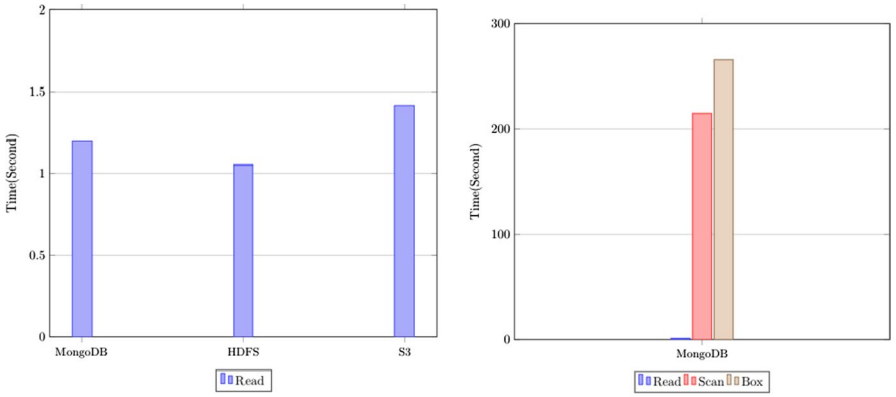
(c) Bounding Box query. **(d)** Distance join query.

Fig. 11 Performance of HDFS-, MongoDB-, and S3-based GeoSpark as the selectivity of queries in GeoSpark is varied

compared to HDFS-based GeoSpark and only by 1.27 ∼ 1.92 times compared to S3-based GeoSpark, respectively, in the case of $RealDist_{1B}$.

4.3.5 The performance as the number of running executors and storage nodes in GeoSpark is varied

Figure 10 shows the performance of HDFS-, MongoDB-, and S3-based GeoSpark as the number of running executors and storage nodes (i.e., Data nodes in HDFS and Shard servers in MongoDB) in GeoSpark is varied. For this, we vary the number of storage nodes from 2 to 6 and we fix one executor for each storage node. We formulate this that the total number of executors in GeoSpark is varied from 2 to 6 when the number of storage nodes is varied from 2 to 6. For Amazon S3, because we cannot control the number of storage nodes, we vary the total executors from 2 to 6 for the same configuration of S3. Then, the used data nodes for S3 will be internally reconfigured.



(a) Read operation of storage systems. **(b)** Scan operation in MongoDB.

Fig. 12 Performance of HDFS and MongoDB, S3 read and scan operations

The result indicates that the performance improvement of HDFS- and S3-based GeoSpark as the total number of running executors becomes large compared to MongoDB-based GeoSpark. Specifically, the performance improvement ratio of HDFS-based GeoSpark is 1.55 ~ 2.65 times when the total number of running executors is varied from 2 to 6; that of MongoDB-based GeoSpark is 1.09 ~ 1.48 times; and that of S3-based GeoSpark is 1.84 ~ 2.65 times. This result implies that the architectures of HDFS- and S3-based GeoSpark are more scalable than that of MongoDB-based GeoSpark to the running executors and data nodes.

4.3.6 The performance as the selectivity of queries is varied

In this section, we measure the performance of MongoDB-, HDFS-, and S3-based GeoSpark varying the selectivity of each query. Figure 11 shows the experimental results. In *k*-NN queries, we vary the number of *k* from 10 to 100,000. In circle queries, we vary the radius from 10KM to 1000KM. In bounding box queries, we vary the bounding box from 10KM x 10KM to 1000KM x 1000KM, respectively. In distance join queries, we vary the threshold distance from 0.1KM to 10KM.

Overall, the performance of queries gradually degrades as the selectivity of queries increases. Specifically, the performance of MongoDB is degraded by 1.35 ~ 2.1 times when we compare the lowest selectivity with the highest selectivity; that of HDFS is degraded by 1.69 ~ 3.65 times; and that of S3 is degraded by 2.38 ~ 4.29 times, which is relatively much more affected by the selectivity.

The interesting result is that, unlike other types of queries, *k*-NN queries show consistent performance for a small number of *k* in all the storage systems. GeoSpark partitions the entire data to multiple partitions and builds the spatial index on the partitions. Then, it uses branch-and-bound tree traversal algorithm so as to find *k* nearest neighbor from the spatial index [12]. The algorithm finds at least one partition from the spatial index where multiple spatial RDDs are stored. As a result, all the spatial RDDs in the partition are loaded in the memory, resulting in the

constant performance for a small of k in the case of k-NN queries while the other queries incrementally require more partitions as the selectivity increases. For actually checking this mechanism, we dramatically increase k up to 100,000, observing the significant performance degradation for all the systems as presented in Fig. 11a.

4.3.7 The performance analysis of the distributed storage systems

In this section, to analyze the performance difference among distributed storage systems, we have conducted two experiments with primitive operations using $RealDist_{100M}$ data set. First, we measure the elapsed time of the read operation from the storage in slave nodes to the memory in the master node to check the pure storage overhead in a cloud configuration for all the systems. To implement the read operation, we used getObject() supported by AWS SDK for Amazon S3, get() supported by Java FileSystem API for HDFS, and MongoCollection.find() supported by MongoClient API for MongoDB. Figure 12a shows the experimental result. The result shows that, even if HDFS shows the best performance, the performance of Amazon S3 is comparable to HDFS, indicating that S3 is well clustered and efficiently connected with Amazon EMR considering it is located on remote servers. We also note that the storage performance of MongoDB is also comparable to the other systems, indicating that its query processing for GeoSpark requires much overhead.

Second, to figure out the performance degradation of MongoDB, we compare the elapsed time of MongoDB for three kinds of operations: (1) the previous read operation, (2) the scan operation for the entire data set, which is the simplest query type that retrieves all the entries in a data set, removing the additional complexity occurred in GeoSpark, and (3) a type of geospatial queries, i.e., bounding box query. Here, for implementing the scan operation, we used MongoCursor() in MongoClient API, retrieving each record (i.e., each SRDD) from a collection data type. Figure 12b shows the experimental result. As a result, we observe that the scan operation takes much more time than the read operation and it occupies approximately 80% of the elapsed time for the bounding box query. This indicates that interpreting of SRDDs for GeoSpark from the collection data type in MongoDB requires much time.

5 Related work

There have been several research efforts to define the data and query sets for the benchmark of spatial data processing and to perform the evaluation. Stonebraker et al. have proposed SEQUOIA, which is the first research effort to define the spatial data sets for benchmarking [25]. They have also defined the query sets, which are real-world queries on the earth science problem: data loading, raster data management, selections, spatial joins, and recursive spatial queries [25]. Using the data and query sets, they have conducted the performance evaluation of three systems, i.e., GRASS, IPW, and Postgres [25]. Gunther et al. have proposed a spatial benchmark [22]. Here, they have considered normal, exponential, and uniform

distributions of data sets and have focused on the spatial join queries for the rectangle type and have created a rather small-scale data set comprised of 1 million objects. Paton et al. have proposed a benchmark framework to compare the PostgresSQL with the Rock & Roll in functionality and performance perspectives. They have also shown that the proposed framework can be applied to other spatial data processing systems [23]. Ray et al. have proposed a benchmark for evaluating all the databases supporting JDBC [24]. They have defined two kinds of scenarios: microscenarios such as topological relations and macroscenarios such as flood risk.

The previous benchmarks for GeoSpark have focused on the comparison between Hadoop- and Spark-based processing systems. Lenka et al. have compared the architectures between SpatialHadoop and GeoSpark and have shown the superiority of GeoSpark by measuring the performance as the number of nodes is varied [33]. Pandey et al. have evaluated Spark-based spatial data processing systems, i.e., GeoSpark, Simba, LocationSpark, Magellan, and SpatialSpark [12]. They have used 200 million point data sets and four types of queries, i.e., *k*-NN, range queries, distance join, and spatial join. They have shown that GeoSpark has the best performance in most cases.

There have been several research efforts to evaluate the performance of the storage engines used in this paper in terms of spatial data processing. Agarwal et al. have compared the performance of MongoDB with the traditional relational databases, i.e., PostGIS and PostgreSQL, on two types of queries: line intersection and point containment [19]. As a result, MongoDB outperforms PostGIS and PostgreSQL in the experiments according to various data set sizes. Makris et al. have evaluated the performance of MongoDB and PostgreSQL to find a better storage system for industrial spatial applications [34]. As a result, they have shown that PostgreSQL outperforms MongoDB in every business scenario they defined. Dede et al. have compared the performance, scalability, and fault tolerance between HDFS and MongoDB and have shown that HDFS outperforms MongoDB [35].

A preliminary version of this work appeared in *Proc. 2020 IEEE International Conference on Big Data and Smart Computing*, pp. 197-200, Feb. 2020 [20]. This is a fully rewritten and extended version of the preliminary version. The major extensions include (1) the evaluation in cloud-based distributed environments, (2) the extended data sets for big spatial data processing, (3) the extended comparative experiments including a cloud-based storage engine, Amazon S3, and (4) the detailed and extended study on related work.

6 Conclusions

In this paper, we have evaluated the performance of big spatial data management systems using GeoSpark. We have evaluated its performance using three underlying distributed storage engines: (1) HDFS, (2) MongoDB, and (3) Amazon S3. To conduct our experimental study on a real cloud computing environment, we have designed and built a distributed experimental environment based on Amazon EMR using up to 6 instances. For the evaluation of big spatial data processing, we have generated data sets considering four kinds of various data distributions and various

data sizes up to one billion point records (i.e., 38.5GB). Through the extensive experiments, we have measured the processing time of storage engines as the following variations: (1) sharding strategies in MongoDB, (2) caching effects, (3) data distributions, (4) data set sizes, (5) the number of running executors and storage nodes, and (6) the selectivity of queries.

The major points observed from the experiments are summarized as follows. (1) The overall performance of MongoDB-based GeoSpark is degraded compared to HDFS- and S3-based GeoSpark in our environmental settings. (2) The performance of MongoDB-based GeoSpark is relatively improved in large-scale data sets compared to the others. (3) HDFS- and S3-based GeoSpark are more scalable to running executors and storage nodes compared to MongoDB-based GeoSpark. (4) The sharding strategy based on the spatial proximity significantly improves the performance of MongoDB-based GeoSpark. (5) S3- and HDFS-based GeoSpark show similar performances in all the environmental settings. (6) Caching in distributed environments improves the overall performance of spatial data processing. These results can be usefully utilized in decision-making of choosing the most adequate storage engine for big spatial data processing in a target distributed environment.

In this paper, we conducted the performance evaluation of spatial data management systems using GeoSpark for focusing on the map–reduce operation, which has the strength in processing large amounts of data. As further work, we plan to extend the current evaluation to encompass more general types of spatial queries, not only depending on the map–reduce framework. In particular, geospatial queries have been integrated with other data types, e.g., graph data on social networks and textual data on various map-based services. By extending the query types, we can naturally include wider types of systems in the evaluation framework, i.e., ranging from not only other types of NoSQL stores such as column stores and graph databases, but also RDBMS such as PostgresSQL.

References

1. Frias-Martinez V, Virseda J, Rubio A, Frias-Martinez E (2010) Towards large scale technology impact analyses: Automatic residential localization from mobile phone-call data. In: Proceedings of the 4th ACM/IEEE international conference on information and communication technologies and development, p. 11. ACM
2. Guille A, Hacid H, Favre C, Zighed DA (2013) Information diffusion in online social networks: A survey. ACM Sigmod Record 42(2):17–28
3. Yang C, Huang Q, Li Z, Liu K, Hu F (2017) Big data and cloud computing: innovation opportunities and challenges. Int J Digital Earth 10(1):13–53
4. Bhuiyan J Uber powered four billion rides in 2017. It wants to do more – and cheaper – in 2018. https://www.vox.com/2018/1/5/16854714/uber-four-billion-rides-coo-barney-harford-2018-cut-costs-customer-service
5. Aji A, Wang F, Vo H, Lee R, Liu Q, Zhang X, Saltz J (2013) Hadoop gis: a high performance spatial data warehousing system over mapreduce. Proceed VLDB Endowment 6(11):1009–1020
6. Eldawy A, Mokbel MF (2015) Spatialhadoop: A mapreduce framework for spatial data. In: 2015 IEEE 31st international conference on Data Engineering, pp. 1352–1363. IEEE
7. Yu J, Zhang Z, Sarwat M (2018) Spatial data management in apache spark: the geospark perspective and beyond. GeoInformatica pp. 1–42

8. Tang M, Yu Y, Malluhi QM, Ouzzani M, Aref WG (2016) Locationspark: A distributed in-memory data management system for big spatial data. Proceed VLDB Endowment 9(13):1565–1568
9. Baig F, Vo H, Kurc T, Saltz J, Wang F (2017) Sparkgis: Resource aware efficient in-memory spatial query processing. In: Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, pp. 1–10
10. Xie D, Li F, Yao B, Li G, Zhou L, Guo M (2016) Simba: Efficient in-memory spatial analytics. In: Proceedings of the 2016 International Conference on Management of Data, pp. 1071–1085
11. Magellan: Geospatial analytics using spark. https://github.com/harsha2010/magellan
12. Pandey V, Kipf A, Neumann T, Kemper A (2018) How good are modern spatial analytics systems? Proceed VLDB Endowment 11(11):1661–1673
13. Huang Z, Chen Y, Wan L, Peng X (2017) Geospark sql: An effective framework enabling spatial queries on spark. ISPRS Int J Geo-Inf 6(9):285
14. Shvachko K, Kuang H, Radia S, Chansler R (2010) The hadoop distributed file system. In: 2010 IEEE 26th symposium on mass storage systems and technologies (MSST), pp. 1–10. Ieee
15. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I et al (2010) Spark: Cluster computing with working sets. HotCloud 10(10–10):95
16. Dean J, Ghemawat S (2008) Mapreduce: simplified data processing on large clusters. Commun ACM 51(1):107–113
17. Zaharia M, Borthakur D, Sen Sarma J, Elmeleegy K, Shenker S, Stoica I (2010) Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In: Proceedings of the 5th European conference on Computer systems, pp. 265–278
18. Banker K (2011) MongoDB in action. Manning Publications Co
19. Agarwal S, Rajan K (2016) Performance analysis of mongodb versus postgis/postgresql databases for line intersection and point containment spatial queries. Spatial Inf Res 24(6):671–677
20. Shin H, Lee K, Kwon HY (2020) Performance evaluation of spatial data management systems using geospark. In: 2020 IEEE International Conference on Big Data and Smart Computing (BigComp), pp. 197–200. IEEE
21. DeCandia G, Hastorun D, Jampani M, Kakulapati G, Lakshman A, Pilchin A, Sivasubramanian S, Vosshall P, Vogels W (2007) Dynamo: amazon's highly available key-value store. ACM SIGOPS Op Syst Rev 41(6):205–220
22. Gunther O, Oria V, Picouet P, Saglio JM, Scholl M (1998) Benchmarking spatial joins a la carte. In: Proceedings. Tenth International Conference on Scientific and Statistical Database Management (Cat. No. 98TB100243), pp. 32–41. IEEE
23. Paton NW, Williams MH, Dietrich K, Liew O, Dinn A, Patrick A (2000) Vespa: A benchmark for vector spatial databases. In: British National Conference on Databases, pp. 81–101. Springer
24. Ray S, Simion B, Brown AD (2011) Jackpine: A benchmark to evaluate spatial database performance. In: 2011 IEEE 27th International Conference on Data Engineering, pp. 1139–1150. IEEE
25. Stonebraker M, Frew J, Gardels K, Meredith J (1993) The sequoia 2000 storage benchmark. ACM SIGMOD Record 22(2):2–11
26. Strobl C (2008) Dimensionally extended nine-intersection model (de-9im)
27. Peterson LE (2009) K-nearest neighbor. Scholarpedia 4(2):1883
28. García-García F, Corral A, Iribarne L, Mavrommatis G, Vassilakopoulos M (2017) A comparison of distributed spatial data management systems for processing distance join queries. In: European Conference on Advances in Databases and Information Systems, pp. 214–228. Springer
29. Jacox EH, Samet H (2007) Spatial join techniques. ACM Trans Database Syst (TODS) 32(1):7
30. Love R (2005) Linux-Kernel-Handbuch: Leitfaden zu Design und Implementierung von Kernel 2.6, vol. 2204. Pearson Deutschland GmbH
31. Zhang S, Zhang B, Chen Z, Lu S (2013) Point collection partitioning in mongodb cluster. In: Proceedings of the 12th International Conference on GeoComputation, LIESMARS, Wuhan University, Wuhan, China
32. Ramsak F, Markl V, Fenk R, Zirkel M, Elhardt K, Bayer R (2000) Integrating the ub-tree into a database system kernel. VLDB 2000:263–272
33. Lenka RK, Barik RK, Gupta N, Ali SM, Rath A, Dubey H (2016) Comparative analysis of spatialhadoop and geospark for geospatial big data analytics. In: 2016 2nd International Conference on Contemporary Computing and Informatics (IC3I), pp. 484–488. IEEE
34. Makris A, Tserpes K, Spiliopoulos G, Anagnostopoulos D (2019) Performance evaluation of mongodb and postgresql for spatio-temporal data. In: EDBT/ICDT Workshops

35. Dede E, Govindaraju M, Gunter D, Canon RS, Ramakrishnan L (2013) Performance evaluation of a mongodb and hadoop platform for scientific data analysis. In: Proceedings of the 4th ACM workshop on Scientific cloud computing, pp. 13–20. ACM

Authors and Affiliations

Hansub Shin[1] · Kisung Lee[2] · Hyuk-Yoon Kwon[1]

✉ Hyuk-Yoon Kwon
hyukyoon.kwon@seoultech.ac.kr

Hansub Shin
sostkr@seoultech.ac.kr

Kisung Lee
lee@csc.lsu.edu

[1] Department of Industrial Engineering, Seoul National University of Science and Technology, Seoul, Republic of Korea

[2] Division of Computer Science and Engineering, Louisiana State University, Baton Rouge, LA, USA