Received: 17 January 2017 | Revised: 14 August 2017 | Accepted: 19 September 2017

DOI: 10.1002/cpe.4354

WILEY

SPECIAL ISSUE PAPER

Sensor data management in the cloud: Data storage, data ingestion, and data retrieval

Prajwol Sangat | **Maria Indrawan-Santiago** | **David Taniar**

Faculty of Information Technology, Monash University, Melbourne, Victoria, Australia

Correspondence
David Taniar, Faculty of Information Technology, Monash University, Melbourne, Victoria, Australia.
Email: David.Taniar@monash.edu

Summary

Sensors are widely used in the field of manufacturing, railways, aerospace, cars, medicines, robotics, and many other aspects of our everyday life. There is an increasing need to capture, store, and analyse the dynamic semi-structured data from those sensors. A similar growth of semi-structured data in the modern web has led to the creation of NoSQL data stores for scalability, availability, and performance, whereas large-scale data processing frameworks for parallel analysis. NoSQL data store such as MongoDB and data processing framework such as Apache Hadoop has been studied for scientific data analysis. However, there has been no study on MongoDB with Apache Spark, and there is a limited understanding of how sensor data management can benefit from these technologies, specifically for ingesting high-velocity sensor data and parallel retrieval of high volume data. In this paper, we evaluate the performance of MongoDB sharding and no-sharding databases with Apache Spark, to identify the right software environment for sensor data management.

KEYWORDS

Apache Spark, data ingestion, data retrieval, data storage, MongoDB, sensor data management

1 | INTRODUCTION

Sensors are becoming an integral part of many modern applications. The sensors ability to sense the environment such as temperature and humidity can provide valuable information to applications for real time or batch analytics. In some applications, a single sensor with multiple sensing capabilities is used to capture the environment data. For example, many fitness trackers employ accelerator, humidity sensor, and heart rate monitor in a single device. For some applications that require complex analytics, a single sensor with multiple sensing capabilities may not be sufficient. For example, in train safety monitoring systems, multiple groups of sensors with different capabilities may need to be deployed in different carriages to capture the movement of the train as a whole. While the data produced by the sensors are beneficial, managing collected sensor data when a large number of sensors are deployed is difficult.

The volume of data collected, the velocity of ingestion of data, and the variety of data format generated by different sensor types are the three main challenges in managing data captured by large sensors systems. As the price of small sensors becomes more affordable, some sensing applications replace many specialised built and high sampling sensors system with a network of off-the-shelf sensors with lower sampling capabilities.[1,2] Comparable accuracy can be achieved by the lower sampling rate by deploying more sensors. The volume of data captured by the specialised built sensor and the network of sensors may be similar. However, network of sensors usually does not have large temporary data storage unlike in a specialised built sensor where they usually include better processor and large storage capacity. In a network of sensors, data needs to be transferred to the main processing centre more often but in smaller size packets. This almost instantaneous transfer requirement will lead to massive increase in the velocity of data coming to the processing centre.

In many cases, in addition to real time processing, data captured by the sensors are used for different analytical processes. To support this analysis, sensors data needs to be stored in the permanent storage for later used in batch mode. Storing the data poses an additional challenge, managing the variety of data formats. With the sensors deployed capture only a single piece of information such as temperature or accelerator, the sensor data when arrive at the processing centre will have a combination of common data, such as time and location, and different sensing data such as acceleration or temperature. The data may be structured in different ways by the sensors.

Considering the challenges above, in this paper, we investigated current database technologies, in particular, the NoSQL database as a means to manage and process sensors data.[3] Two main data processing task for sensors data, data ingestion, and data retrieval were investigated. To handle

the large volume and high velocity, the parallel data processing capabilities in NoSQL database (e.g., MongoDB) in conjunction with parallel data processing platform (e.g., Spark) were tested. The investigation was based on a case study of heavy haul railway monitoring systems.[1]

Some NoSQL databases provide extensions that allow users to use the Map-Reduce paradigm to operate on the data stored in NoSQL databases.[4] Recently, MongoDB released *MongoDB Connector for Spark*[*] that can be used to connect Apache Spark to MongoDB. This integrated environment that allows the capture of semi-structured data using NoSQL database (MongoDB) and performs analytical queries on them (using Apache Spark), would address the problems discussed before. However, there is a limited understanding of the performance in this integrated environment because this connector is not yet mature.

In this paper, we present the efficiency and reliability of an integrated analysis environment that consists of MongoDB and Apache Spark interfaced using the official *MongoDB Connector for Spark*. Specifically, we evaluate the performance of MongoDB in sharding and no-sharding environment from the perspective of **data ingestion** and **data retrieval** in sensor data management. We make four significant contributions in this paper:

- We show that document-oriented databases such as MongoDB enable sensor data management by facilitating the storage of semi-structured data, and *sharding* helps in scaling the infrastructure along with the growth of data.
- We show that an integrated environment such as Apache Spark with MongoDB can improve the efficiency of data ingestion process. We have identified that using a data processing framework such as Apache Spark can achieve better performance than the native Mongo Import for both reads and write in the sharding environment. Therefore, sensor data management can benefit from an integrated environment because it supports high ingestion rate and in-line data processing.
- We show that the data retrieval is more efficient in an integrated environment than using native MongoDB query language and hence also addresses the problem of complex data analysis and use of machine learning algorithms.
- We find that *MongoDB Connector for Spark* provides an excellent interface for analyzing the data that lives in or needs to be stored in MongoDB even though it is in its embryonic stage.

The structure of the paper is as follows: we begin by describing the previous work in Section 2; then, we describe the technology used and the environment setup used for the experiment in Section 3. We describe the data storage used in the experiment in Section 4 and experiments for data ingestion and data retrieval in Sections 5 and 6. Finally, we conclude in Section 7.

2 | RELATED WORKS

There are several NoSQL databases available for use that provide similar functionality (e.g., fault tolerance, high availability, and scalability) but support different data models.[5] Not all of them have support for Apache Spark. Some of the NoSQL databases that support Apache Spark are Cassandra,[6] HBase,[7] MongoDB,[8] etc. However, our choice of MongoDB[8] is motivated by the need for a document-oriented store since we receive data in JSON format. There are other document-oriented databases such as CouchDB,[9] but they do not have official support for Apache Spark.

MongoDB provides native support for MapReduce algorithms. Bonnet et al[10] have described how MapReduce algorithms supported by MongoDB can be used to perform aggregates on a large volume of data. However, they do not provide quantitative results. Dede et al[11] have demonstrated that Hadoop MapReduce performs better than native MongoDB MapReduce for scientific data analysis. Verma et al[12] have evaluated MongoDB and Hadoop MapReduce performance for evolutionary genetic algorithms, but the configurations they used are different from Dede et al,[11] and they have not attempted to compare them. Dede et al[11] have characterized the performance of NoSQL when used with Hadoop/HDFS. However, to our knowledge, there is no prior work on the performance of NoSQL database (MongoDB) in shard and no-shard environment when used with Apache Spark.

Cooper et al[13] have compared different NoSQL and relational databases and have provided a Yahoo Cloud Serving Benchmark (YCSB) as an open source effort that can be extended to test different systems and workloads. Dory et al[14] studied the elastic scalability of MongoDB, HBase, and Cassandra on a cloud infrastructure. These studies have not considered the performance achievable using these NoSQL databases with big data processing frameworks.

Floratou et al[15] compared NoSQL databases (MongoDB and Hive) with a relational database (SQL Server Parallel Data Warehouse (PWD)) using YCSB[13] and TPC-H[†] benchmarks. They have compared these technologies for decision support analysis and interactive data serving. They have demonstrated that even though PWD performs better than NoSQL, the NoSQL databases provide functionality advantages of higher importance such as replication, sharding, load balancing, and flexible data models and all of these are included with MongoDB distribution.

3 | CASE STUDY ENVIRONMENT

This section provides brief information on the technologies used and environment setup for the experiment. We configured and deployed Hadoop, HDFS, Apache Spark, MongoDB, and the MongoDB Connector for Spark on the National eResearch Collaboration Tools and Resources (NeCTAR)[‡] cloud computing environment.

[*] https://docs.mongodb.com/spark-connector/
[†] http://www.tpc.org/tpch/
[‡] https://www.nectar.org.au/about-nectar

1. Hadoop/HDFS Setup: We deployed Hadoop/HDFS version 2.6 using standard configuration parameters. The application server is running the HDFS to store the incoming data and the Spark streaming program access the data from HDFS to process it before inserting into the MongoDB in the database server.
2. Spark Setup: Apache Spark[§] is an open source large-scale distributed data processing framework. Spark introduces an abstraction called resilient distributed datasets (RDDs), which represents a read-only multiset of data items partitioned across a set of machines that is maintained in a fault-tolerant way.[16] Spark is the first system to allow an efficient, general purpose programming language to be used interactively to process large data sets on a cluster. We deployed Apache Spark version 1.6.2 in a stand alone mode and controlled the configurations from the application code. For example, we set `spark.serializer` as `org.apache.spark.serializer.KryoSerializer`. The application was created using Scala 2.10.4.
3. MongoDB Setup: MongoDB is an open source NoSQL database built for scalability, performance, and high availability. The data is stored as *documents* instead of tables with columns and rows. Each of the document can be an associative array of scalar values, lists, or nested associative arrays. These documents are serialized as JSON objects and are stored internally using the binary encoding of JSON known as BSON.[17] In MongoDB, data is distributed and stored on more than one server (*known as shard server*) for concurrent access and efficient read/write operations. This technique is called *"Sharding."* It is implemented by dividing the MongoDB server into `mongos` - a set of routing servers, that route operations to `mongod` - a set of data servers. We ran MongoDB in two modes: as a single server and with sharding. For the single server test, each mapper connects to the single MongoDB server (which is same as a `router` in *shard server*). For sharded server, the mappers connected to `mongos` router that had three *config servers* and three *shard servers*.
4. MongoDB Connector for Spark: On 28 June 2016, the official version (V1.0) of MongoDB Connector for Spark was released. This connector provides a seamless integration between MongoDB and Apache Spark. It effectively uses MongoDB's aggregation pipelines and secondary indexes to extract, filter, and process only the subset of data required for the Spark process. In addition, to maximize performance across large distributed data sets, it links RDDs to the source MongoDB node and minimizes the data transfer across the cluster.[¶]
5. Machines: We conducted our experiments on the NeCTAR cloud computing environment. The *application server* running Spark had 16 dedicated cores, 64 GB of memory, 459 GB HDD and ran 64 bit Ubuntu GNU/Linux 14.04.2 LTS. The *database server* (server running `mongos`) and each of the *shard server* had 4 dedicated cores, 16GB of memory, 130 GB HDD whereas each of the *config servers* had 1 core, 4GB memory, and 30 GB HDD.

In the following sections, we describe and justify the tools used for data storage and discuss and analyse the results for the data ingestion and data retrieval experiments.

4 | DATA STORAGE

Over the past few years, there has been a tremendous difference in the way we store and access our data. We are no longer encumbered with hard disk drives and flash drives. With the emergence of cloud and distributed databases, data storage has transformed quite a bit. Therefore, when we say "Data Storage," we mean "Distributed Data Storage."[18]

Distributed data storage is a computer network where data or information is stored (or replicated) on more than one node or computer.[19] It commonly refers to a distributed database where information is stored on multiple nodes or a computer network in which information is stored on multiple peer network nodes.

Distributed databases are NoSQL databases that quickly retrieves data over a large number of nodes. In these databases, arbitrary or ad hoc querying is not considered as important as the availability of data. Therefore, distributed data stores have an increased availability of data at the expense of consistency. Also, the high-speed read/write access facility of distributed database consequences in reduced consistency because it is not possible to have consistency, availability, and partition tolerance all at the same time.[20]

There are several distributed databases available in the market. Google's Big Table,[21] Amazon Dynamo,[22] Apache Cassandra,[6] MongoDB[8] to name a few. Based on the classification by data model, Big Table, and Cassandra is column oriented, Dynamo is a key-value pair, and MongoDB is document based. The choice of the NoSQL distributed database typically depends on the data model as most of them provide similar functionality (such as scalability, fault tolerance, high availability).

As part of modernization drive that took place over the last decade, companies have been integrating sensors into their products and systems for different purposes. For instance, heavy haul railways are using sensors in the ore cars to monitor the track conditions and abnormalities. We have used the data available from heavy haul railways to perform the experiments.

Sensor data is semi-structured. In some of its forms, there is no separate schema, in others it exists but only places loose constraints on the data.[23] JavaScript Object Notation (JSON) is a semi-structured data format and is primarily used as a data exchange format on the modern web. JSON supports all the basic data types such as strings, boolean, numbers, as well as arrays and hashes. We have used JSON in our experiment as our data format, and the sample JSON record is given in Listing 1. Therefore, for the experiment, MongoDB is the good choice of the data storage because it uses JSON documents to store records, similar to tables and rows in a relational database.

[§] http://spark.apache.org/
[¶] https://docs.mongodb.com/spark-connector/

Listing 1 A sample JSON record, representing railway data

```
{
  ''acc'' : {
       "r3" : 0.000,
       "r4" : 0.000
  },
  ''somattime'' : 11534.000,
  ''kmh'' : 0.900,
  ''geocode'' : "qs76f1nb",
  ''carorient'' : 0.900,
  ''gps'' : {
       ''lon'' : 117.16774825,
       ''lat'' : -20.60475627
  },
  ''trackname'' : ''MLXX'',
  ''trackkm'' : 0.050,
  ''cfa'' : {
      ''min'' : 0.000,
      ''max'' : 0.000
  },
  ''direction'' : ''ToMine''
}
```

Figure 1 represents the overall architecture of the system used in the experiment. We performed experiments on single MongoDB database and MongoDB sharding database. *Sharding* (horizontal scaling) is a technique of distributing data across more than one computer or commodity servers for effective management of big data in the cloud.[24] This technique is used to support the server deployments that needs to process enormous data sets and requires high throughput. The database systems that store humongous data sets or run applications with high throughput often challenge the capability of a single server. For instance, if the querying rate is high, the CPU capacity of the single server can be easily exhausted, or if the data size is bigger than the available memory, it can exhaust the I/O capacity of the storage.[25] Therefore, not only sharding divides the same dataset and spreads over to different *shard servers*, but also the additional servers can be included in the cluster to increase the capacity if required. The *shard servers* may not be of high CPU clock speed or with large memory, but each machine manages the subset of the overall workload and, therefore, provides better efficiency than a single server with high CPU clock speed and humongous memory. The MongoDB sharded cluster used for the experiment consists of following components:

- *Config servers*: *Config servers* store meta information and configuration settings for the sharded cluster.
- *Shard servers*: Each shard server contains a subset of the dataset.
- *Database server* (`mongos`): The `mongos` acts as a router that interfaces between the application server and the sharded cluster.

MongoDB has enabled sensor data management by facilitating the storage of semi-structured data, and *sharding* has helped in scaling the infrastructure along with the growth of data. We performed tests on the single MongoDB database and MongoDB sharding database in order to compare the performance differences between them. The experiments are discussed in the following Sections 5 and 6.

5 | DATA INGESTION

Data Ingestion is the process of acquiring and importing data into a data store or a database. Sensors can transmit multi-paradigm data (i.e., data in real time or batches). If the data is ingested in real time, each record is pushed into the database as it is emitted by the sensors, else the records are collected in a buffer location (i.e., creating a batch of records) and pushed into the database in discrete chunks at periodic intervals. A constructive data ingestion process commences by organizing information sources, validating each document or information, and routing information to the correct destination.

One of the key consideration to storing a large amount of data is to ensure that the data is ingested efficiently and with confidence. Faulty or overloaded ingestion causes incomplete and inaccurate data to be pushed into the storage, therefore resulting in the incorrect analysis. Ingestion gets inadequate consideration for two reasons:

1. Firstly, it is not considered as crucial as data science or data analytics. Even though the internal architecture of the data ingestion system is critical to delivering novel insights, it does not stimulate data architects in quite the same way as machine learning.

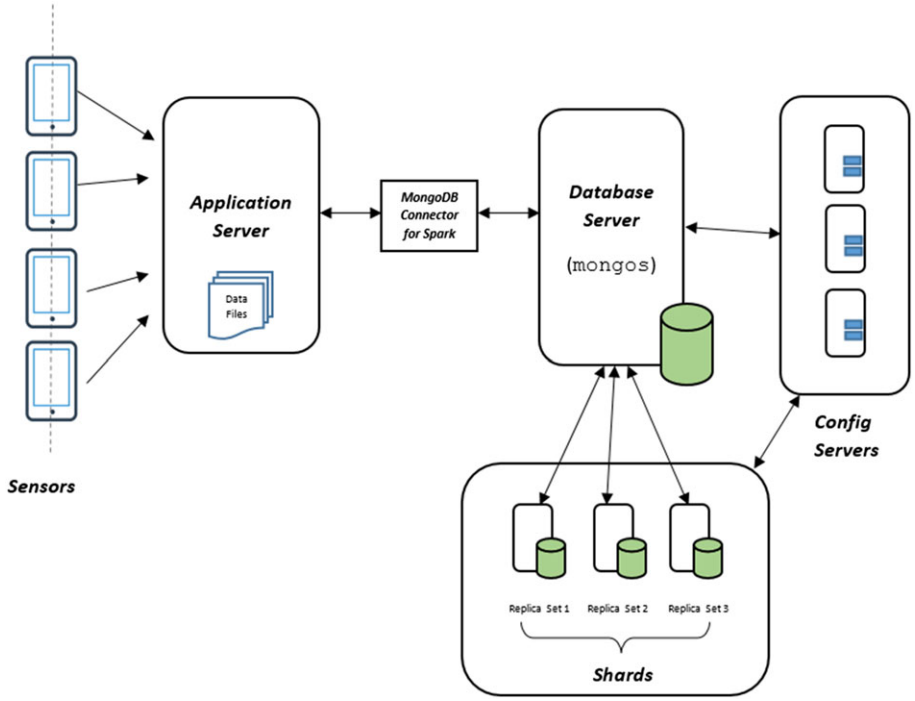

FIGURE 1 System architecture used in the experiment

2. Secondly, it is perceived to be easy. Even though we know the sources of our data (e.g., web logs and sensors), what we do not consider is that numerous sources of batch and streamed data can feed the data storage instead of single source, and therefore, the data ingestion can be a challenging operation considering physical resources.

When the data source is diverse (hundreds of sources emitting dozens of data formats), improving the rate of data ingestion and efficiency of data processing becomes a challenging task. The analytical value of the data in the storage entirely depends on its completeness, accuracy, and consistency. Therefore, the goal should not be to build the analytics, but rather a continually and accurately operating ingestion system, and this is a complex task requiring proper planning, specialized tools, and expertise. To that end, vendors nowadays offer different tools that are customized to particular computing environments or software applications (e.g., *MongoDB Connector for Spark*).

When the data ingestion process is automated, the application used to carry out this process may include data preparation features (e.g., formatting data) such that the data can be analyzed on the fly or at a later time using business intelligence or analytics software. In this experiment, we handle multi-paradigm data, i.e., the data can be in the form of real-time streams or micro batches of JSON documents. The application program running on the application server handles both real-time streams and micro batch inputs. The data ingestion process as mentioned earlier involves the data pre-processing (data preparation) and the data insertion into the MongoDB database. The design of the data ingestion system and the efficiency of the pre-processing algorithms has been discussed in Sangat et al.[1]

The main motivation of this experiment is to identify the upper limit of ingestion of the input records as well as find and compare the ingestion rate of the MongoDB vs. MongoDB shard database. MongoDB sharding technology is being tested because this technology distributes data across multiple servers and provides high throughput operations.[25] This experiment will help recommend the suitable database model for the high-velocity data and help us answer some of the questions such as

1. Does sharding improve the data ingestion?
2. Is there any defined limit of input records per second in the sharding database?

To answer these questions, we performed different tests which are discussed next.

5.1 | Ingestion time test

This test is performed to understand the time required for different setups to ingest the different amount of records. Figure 2 depicts the time taken to ingest the records over increasing number of records from 10 to 120 thousand with Mongo Import and Apache Spark (both using 4 Cores), MongoDB sharding database, and MongoDB no-sharding database. The time taken for ingestion is gradually increasing with the increasing number of records for each setup. However, it is captivating to account the Apache Spark showing a splendid ingestion time with MongoDB sharding database whereas the worst performance with MongoDB no-sharding database. The similar result has been obtained for Mongo Import. It performs well with MongoDB sharding database and relatively less with MongoDB no-sharding database. We believe that this was due to memory contention in MongoDB no-sharding database server. As we increased the input data size, we observed an increase in memory uses on the server for MongoDB

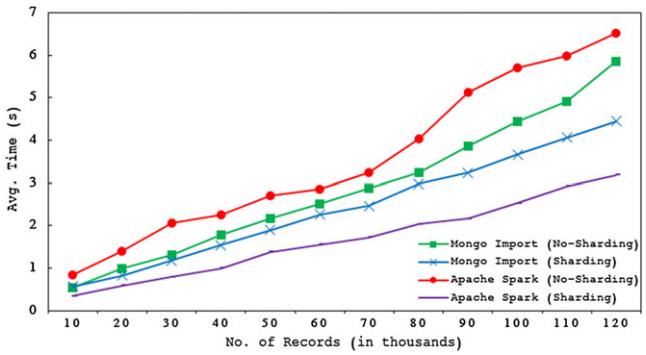

FIGURE 2 Average time taken for data ingestion using Mongo Import and Apache Spark in sharding and no-sharding database

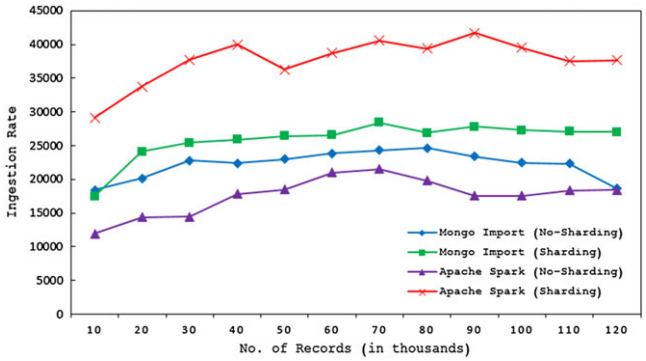

FIGURE 3 Comparison between Mongo Import and Apache Spark for data ingestion. *Ingestion Rate = No. of Records / Time (s)*. Apache Spark is the best for data ingestion when used with MongoDB Sharding

operations and this, in turn, lead to a slower time. The ingestion time improves in sharded setup because it can balance the increasing number of incoming concurrent connections from the application.[11] From this experiment, we can say that sharding improves the ingestion time and provides high throughput than no-sharding database.

5.2 | Ingestion rate test

Figure 3 shows the ingestion rate *(ingestion rate = number of records/time (s))* over increasing number of records from 10 to 120 thousand with Mongo Import and Apache Spark, MongoDB sharding database, and MongoDB no-sharding database. Apache Spark with MongoDB sharding database demonstrates exceptional ingestion rate (greater than 40,000 records per second) over other experiment setups, whereas, Apache Spark with MongoDB no-sharding database is the least performing setup. As mentioned above, it is because the no-sharding database cannot handle the increasing number of concurrent connections as efficiently as MongoDB sharding database.[11] That being said, it should be noted that the results may vary based on the hardware infrastructure *(e.g., Hard Disk Drive vs. Solid State Drive)* as ingestion is also dependent on the disk write speed.

The another question that arises is *"Why is there not much difference between the ingestion rate of Mongo Import in MongoDB sharding and MongoDB no-sharding database as for Apache Spark?"*. To answer the question, we need more information on MongoDB.

MongoDB is not CPU bound. It means that there is small work that CPU does. Most of the database installations including MongoDB are constrained by the amount of I/O and memory. In MongoDB, each incoming connection gets a thread. Typically, each parallel processing thread will be run on a separate core. Since, we have a 4-core machine, we had 4 concurrent executing threads, and all 4 cores were being used.

There is a huge dependency in the driver used. If the driver uses a connection pool and makes multiple connections to the database (*e.g., MongoDB connector for Spark*), then it can initiate multiple queries in parallel from the client, and it runs in parallel on the server side. However, if the driver does not provide a connection pool (*e.g., the 'Mongo' shell*), then the same connection will be used for each incoming query, and the queries would be run serially. Thus, there is not much difference in ingestion rate of Mongo Import in MongoDB sharding and no-sharding database.

The rule to understand is that each incoming connection to MongoDB gets its thread and each thread can only work on one operation at a time. Nevertheless, we can conclude that an integrated environment such as Apache Spark with MongoDB can improve the efficiency of data ingestion process. We have identified that using a data processing framework such as Apache Spark can achieve better performance than the native Mongo Import for both reads and write in the sharding environment. Therefore, sensor data management can benefit from an integrated environment because it supports high ingestion rate and in-line data processing. However, at some point, we expect that the write performance of MongoDB will remain to be a bottleneck along with the cost of routing data between the sharding servers.

6 | DATA RETRIEVAL

Data Retrieval is the process of searching, identifying, and extracting required data from a database.[26] It typically requires writing data extraction queries or command by the users or an application and executing them on a database. Based on the queries or commands provided, the database searches for and retrieves the data requested. Applications can produce data in different formats, and they can be stored in a file, printed, or viewed on the screen. In order to speed up data retrieval, parallel computers are often used.[27]

With the use of MongoDB, we can store the semi-structured data (JSON) used in the experiment as well as accommodate the continual changes in the data that can occur over time. MongoDB provides an appropriate data model and query language required for this experiment. However, we have some problems that MongoDB alone cannot address.

Firstly, we need to perform statistical data mining to discover patterns and trends in the data. These kinds of analytics are difficult with MongoDB but are naturally implemented as MapReduce programs. Secondly, all the queries in MongoDB have the scope of a single collection. It means that the data in the different collections cannot be joined to get meaningful insights from the data.

The alternative to MongoDB query language is to use big data processing frameworks such as Apache Spark for data retrieval. We could have used other big data processing frameworks such as Apache Hadoop, but Apache Spark has proved itself to be superior to Hadoop in performing analytical tasks[16,28] as well as it can be easily used in data retrieval using *MongoDB connector for Spark*.

Since the data resides in the MongoDB, we want to perform experiments to answer some of the questions such as

1. How fast can the data retrieval be done in standalone MongoDB?
2. Is the performance of data retrieval different using the shards?
3. Do queries in Mongo shell execute faster that application layer programs? (Answering this question is important because application layer programs can provide a lot of analytical functionalities that Mongo shell cannot.)
4. For the exact and range queries, Is Apache Spark better than the Mongo shell?

Answering these queries will help us recommend the suitable query tool for data retrieval regarding performance and our analytical requirement. To answer these questions, we performed different tests which are discussed next.

6.1 | Retrieval time test

This test is performed to understand the time required for different setups to retrieve the different amount of records. Figures 4 and 5 show the query execution time using three different approaches (Mongo Shell, Spark Core application, and Spark SQL application) for MongoDB sharding and no-sharding database. We used the same Exact Query and Range Query for each approach and recorded the execution time for those approaches.

From Figures 4 and 5, it is interesting to note that the time is gradually increasing for both MongoDB sharding and no-sharding database in Mongo Shell although the time is relatively consistent for the Spark Core and Spark SQL even with the increasing number of records.

In Mongo Shell, we do not have control over the execution. As mentioned before, the incoming connection to MongoDB gets its own thread. Each thread only works on one operation at a time, and only one thread will handle all the operations. Therefore, the increase in the number of records to search requires an almost linear increase in time to retrieve the results. However, with Apache Spark, we have control over the number of cores to use in the application that makes it possible for the multiple connections to MongoDB providing us faster results and quicker analysis of data.

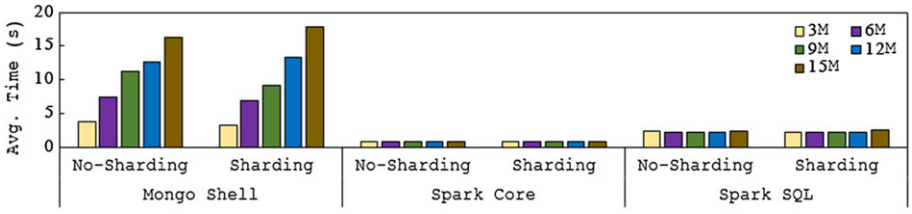

FIGURE 4 Mongo Shell vs. Spark Core vs. Shark SQL query execution time for Exact Query in MongoDB sharding and no-sharding database

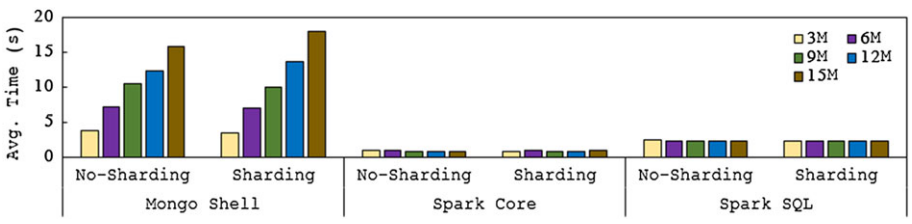

FIGURE 5 Mongo Shell vs. Spark Core vs. Shark SQL query execution time for Range Query in MongoDB sharding and no-sharding database

6.1.1 | Mongo shell query execution time test

From Figures 6 and 7, we found that with the increasing number of records, MongoDB sharding database performance degrades compared to the no-sharding database. The read time delay in the sharded setup is due to the bottleneck in the database server `(router)` that has the overhead of routing data required to be read from the sharding servers.

6.1.2 | Spark query time test

Next important observation we made was the difference in the query execution time of Spark Core and Spark SQL. Even though both of them are almost stable, we were curious to understand *"Why is Spark SQL taking more time compared to Spark Core?"*. The major difference was seen in *start up time* in Spark Core (i.e., starting the spark context) and Spark SQL (i.e., starting spark sql context) which was causing the overall delay in query execution time.

In Figure 8, there is a minor difference in the query time for Spark Core and Spark SQL. However, in Figure 9, we can see the drastic difference in the loading time. This can be attributed to Spark SQL requiring Spark Context to load before it can load itself, and therefore, the increase in time is observed. The search operation, when performed with Spark, is faster once the data and context are loaded since there are no more I/O operations, and only in-memory operations are performed.

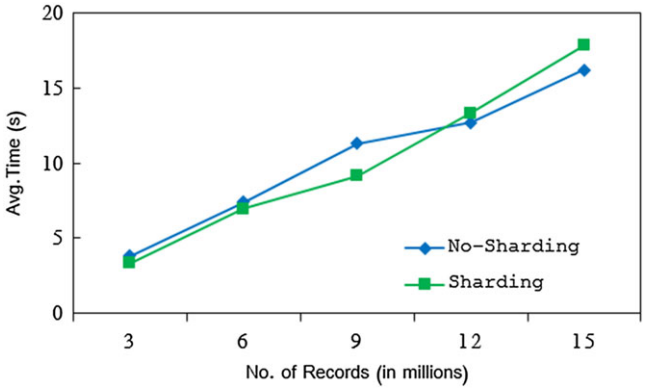

FIGURE 6 Mongo shell **exact query** execution results

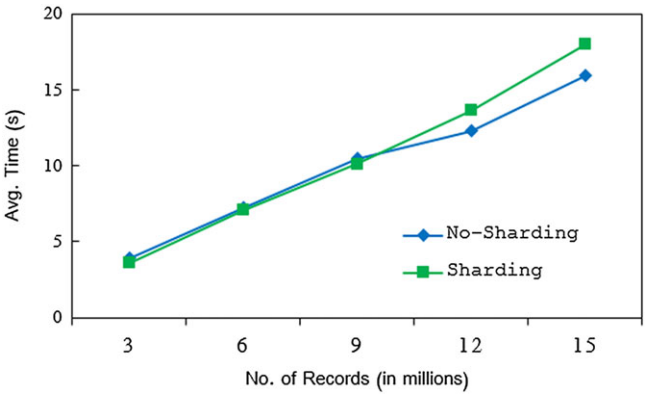

FIGURE 7 Mongo shell **range query** execution results

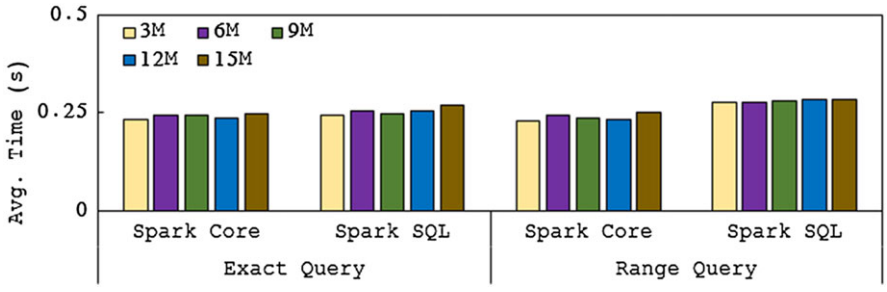

FIGURE 8 Query time comparison for Spark Core and Spark SQL in MongoDB sharding database

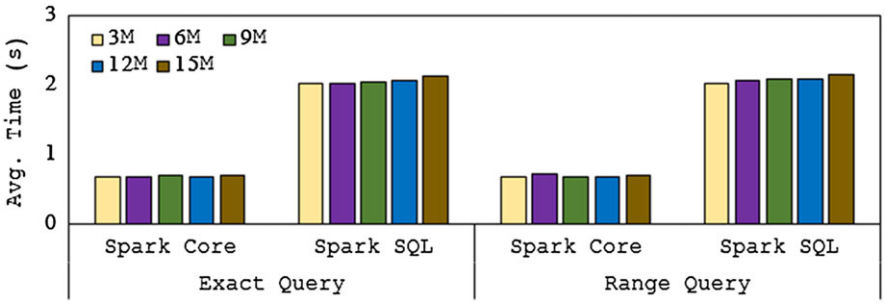

FIGURE 9 Load time comparison for Spark Core and Spark SQL in MongoDB sharding database

In summary, we can say that data retrieval using sharded database can be efficient for the small amount of data, but as the data grows, it can have the bottleneck in the database server `(router)` to route data required being read from the sharding servers. We can also conclude that the data retrieval is more efficient in an integrated environment than using native MongoDB query language and hence also addresses the problem of complex data analysis and use of machine learning algorithms. Also, we have found that *MongoDB Connector for Spark* provides an excellent interface for analyzing the data that resides or needs to be stored in MongoDB even though it is not yet mature.

7 | CONCLUSION

In this paper, we compared the performance of MongoDB in sharding and non-sharding environment. We have performed the data ingestion test using Mongo Import and Apache Spark. The data ingestion test was performed to find the difference in the performance between sharded setup and no-shard setup. It was also to understand if using data processing tools can have an advantage over native Mongo Import in ingestion of data. We also performed the data retrieval test using native Mongo Shell and Apache Spark. The retrieval test was performed to understand if the performance of data retrieval is different using the shards. It was also to investigate if queries in Mongo shell executed faster than the application layer programs. This investigation was important because application layer programs can provide a lot of analytical functionalities that Mongo shell cannot.

In summary, document-oriented databases such as MongoDB enables sensor data management by facilitating the storage of semi-structured data, and *sharding* helps in scaling the infrastructure along with the growth of data. In addition, an integrated environment such as Apache Spark combined with MongoDB improves the efficiency of data ingestion process. Using a data processing framework such as Apache Spark achieves a better performance than the native Mongo Import for both read and write operations in the sharding environment. Thus, sensor data management can benefit from an integrated environment because it supports high ingestion rate and in-line data processing. Furthermore, the data retrieval is more efficient in an integrated environment than using native MongoDB query language and hence also addresses the problem of complex data analysis and use of machine learning algorithms. Also, we have found that although the official *MongoDB Connector for Spark* is in its embryonic stage, it provides a good interface for analyzing the data that resides or needs to be stored in MongoDB without scarifying the performance.

In future, we are planning to test the architecture used in this study on real case studies to see how this integrated environment improves the data analysis in real world. Other operations and structures, such as update,[29] and parallel index,[27] will also be studied.

ORCID

David Taniar http://orcid.org/0000-0002-8862-3960

REFERENCES

1. Sangat P, Indrawan-Santiago M, Taniar S, Oh B, Reichl P. Processing high-volume geospatial data: a case of monitoring heavy haul railway operations. In: Connolly M, ed. International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA, Procedia Computer Science, Vol. 80: Elsevier; 2016:2221-2225.
2. Barolli A, Elmazi D, Obukata R, Oda T, Ikeda M, Barolli L. Experimental results of a Raspberry Pi and OLSR based wireless content centric network testbed: comparison of different platforms. *IJWGS*. 2017;13(1):131-141.
3. de Lima C, dos Santos Mello R. On proposing and evaluating a NoSQL document database logical approach. *IJWIS*. 2016;12(4):398-417.
4. Esposito C, Ficco M. Recent developments on security and reliability in large-scale data processing with MapReduce. *IJDWM*. 2016;12(1):49-68.
5. Han J, Haihong E, Le G, Du J. Survey on NoSQL database. In: Pervasive Computing and Applications (ICPCA), 2011 6th International Conference on IEEE. Port Elizabeth, South Africa; 2011:363-366.
6. Lakshman A, Malik P. Cassandra: a decentralized structured storage system. *Oper Syst Rev*. 2010;44(2):35-40.
7. George L. *Hbase: The Definitive Guide*. Sebastopol, CA, USA: "O'Reilly Media, Inc."; 2011.
8. Micbael Dirolf KC. *MongoDB: The Definitive Guide*; 2011.
9. Anderson JC, Lehnardt J, Slater N. *CouchDB: The Definitive Guide*. Sebastopol, CA, USA: "O'Reilly Media, Inc."; 2010.

10. Bonnet L, Laurent A, Sala M, Laurent B, Sicard N. Reduce, you say: what NoSQL can do for data aggregation and BI in large repositories. In: Morvan F, Tjoa AM, Wagner R, eds. *2011 Database and Expert Systems Applications, DEXA, International Workshops*, Toulouse, France, August 29 - Sept. 2, 2011: IEEE Computer Society; 2011:483-488.
11. Dede E, Govindaraju M, Gunter DK, Canon SR, Ramakrishnan L. Performance evaluation of a MongoDB and hadoop platform for scientific data analysis. In: Chard K, ed. ScienceCloud'13, Proceedings of the 4th ACM HPDC Workshop on Scientific Cloud Computing, New York, NY, USA, June 17, 2013: ACM; 2013:13-20.
12. Verma A, Llorà X, Venkataraman S, Goldberg DE, Campbell RH. Scaling eCGA model building via data-intensive computing. In: Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2010, Barcelona, Spain, 18-23 July 2010 IEEE; 2010:1-8.
13. Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R. Benchmarking cloud serving systems with YCSB. In: Hellerstein JM, Chaudhuri S, Rosenblum M, eds. Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010: ACM; 2010:143-154.
14. Dory T, Mejías B, Roy P, Tran NL. Measuring elasticity for cloud databases. In: Proceedings of the Second International Conference on Cloud Computing, GRIDS, and Virtualization Citeseer. Rome, Italy; 2011:1-7.
15. Floratou A, Teletia N, DeWitt DJ, Patel JM, Zhang D. Can the elephants handle the NoSQL onslaught? *PVLDB*. 2012;5(12):1712-1723.
16. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: Cluster computing with working sets. In: Nahum EM, Xu D, eds. *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10*, Boston, MA, USA, June 22, 2010. USENIX Association; 2010.
17. Chodorow K. *MongoDB: The Definitive Guide*. Sebastopol, CA, USA: "O'Reilly Media, Inc."; 2013.
18. Barbierato E, Gribaudo M, Iacono M. Modeling and evaluating the effects of big data storage resource allocation in global scale cloud architectures. *IJDWM*. 2016;12(2):1-20.
19. Reuter J, Jackson J, Voigt D, Veitch A. Distributed data-storage system Mar 6 2006. US Patent App. 11/369,240.
20. Brewer EA. Towards robust distributed systems (abstract). In: Neiger G, ed. Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA. ACM; 2000:7. https://doi.org/10.1145/343477.343502
21. Chang F, Dean J, Ghemawat S, et al. Bigtable: A distributed storage system for structured data. *ACM Trans Comput Syst*. 2008;26(2):4:1-4:26.
22. DeCandia G, Hastorun D, Jampani M. Dynamo: Amazon's highly available key-value store. In: Bressoud TC, Kaashoek MF, eds. Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007: ACM; 2007:205-220.
23. Buneman P. Semistructured data. In: Mendelzon AO, Özsoyoglu ZM, eds. Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12-14, 1997, Tucson, Arizona, USA. ACM Press; 1997:117-121.
24. Bagui S, Nguyen LT. Database sharding: to provide fault tolerance and scalability of big data on the cloud. *IJCAC*. 2015;5(2):36-52.
25. Liu Y, Wang Y, Jin Y. Research on the improvement of MongoDB auto-sharding in cloud environment. In: 2012 7th International Conference on Computer Science & Education (ICCSE). Melbourne, VIC, Australia: IEEE; 2012:851-854.
26. Taniar D, Leung CHC, Rahayu JW, Goel S. *High Performance Parallel Database Processing and Grid Databases*. John Wiley & Sons; 2008.
27. Taniar D, Rahayu JW. A taxonomy of indexing schemes for parallel database systems. *Distrib Parallel Databases*. 2002;12(1):73-106.
28. Shi J, Qiu Y, Minhas UF, et al. Clash of the titans: MapReduce vs. Spark for large scale data analytics. *PVLDB*. 2015;8(13):2110-2121.
29. Cui Z, Zhu H, Shi J, Chi L, Yan K. Efficient authorisation update on cloud data. *IJWGS*. 2016;12(2):109-141.

How to cite this article: Sangat P, Indrawan-Santiago M, Taniar D. Sensor data management in the cloud: Data storage, data ingestion, and data retrieval. *Concurrency Computat: Pract Exper*. 2018;30:e4354. https://doi.org/10.1002/cpe.4354