Execution of an Advanced Data Analytics by Integrating Spark with MongoDB

Ms. C. Vijayalakshmi, M.C.A., M.Phil.,
Department of Computer Science and Engineering, Jubail University College (Female Branch)
PO Box 10074, Jubail Industrial City - 31961, Kingdom of Saudi Arabia
vijimichael50@gmail.com

ABSTRACT

Spark has several advantages compared to other big data and MapReduce technologies like Hadoop and Storm. Spark provides a comprehensive, unified framework to manage big data processing requirements with a variety of data sets that are diverse in nature (text data, graph data etc.) as well as the source of data (batch vs. real-time streaming data). *Spark SQL* is an easy-to-use and power API provided by *Apache Spark*. Spark SQL makes it much easier reading and writing data to do analysis. *MongoDB Connector for Apache Spark* is a powerful integration that enables developers and data scientists to create new insights and drive real-time action on live, operational and streaming data. This paper demonstrates some experimentation on the MongoDB Connector for Apache Spark that how Spark SQL library can be used to store, retrieve and execute the structured/semi-structured datasets such as BSON against the Non-Relational database MongoDB, an open-source and leading NoSQL database.

KEYWORDS

Spark SQL, MongoDB, NoSQL databases, MongoDB Connector for Apache Spark, Data Analytics with Spark SQL and MongoDB, Data Mining on NoSQL data

1 INTRODUCTION

In the era of *Big data*, the Complex data is growing. Unstructured data will account for more than 80% of the data collected by organizations. Data increasingly stored in Non-Relational databases as shown in Table 1. The Non-Relational databases (such as MongoDB, HBASE etc.) are very efficient in the domains of *Big data*, *Content Management and Delivery*, *Mobile and Social Infrastructure*, *User Data Management* and *Data Hub* etc. as MongoDB provides document oriented storage, index on any attribute, replication, high availability, Auto-sharding, fast in-place updates and rich queries.

Table 1. Comparison of Data

Volume	GBs-TBs	TBs-PBs
Structure	Structured	Structured, Semi-structured and Unstructured
Database	Relational Datbases: (Oracle, SQL Server, MySQL etc.)	Non-Relational Datastores: (Hadoop, HBASE, MongoDB etc.)
Schema	Fixed	Dynamic/Flexible
Structure	DBA-controlled	Application-controlled

Apache Spark started as a research project at UC Berkeley in the AMPLab, which focuses on big data analytics. Their goal was to design a programming model that supports a much wider class of applications than MapReduce, while maintaining its automatic fault tolerance. In particular, MapReduce is inefficient for *multi-pass* applications that require low-latency data sharing across multiple parallel operations. These applications [1] are quite common in analytics, and include:

- *Iterative algorithms*, including many machine learning algorithms and graph algorithms like PageRank.
- *Interactive data mining*, where a user would like to load data into RAM across a cluster and query it repeatedly.
- *Streaming applications* that maintain aggregate state over time.

2 SPARK

Apache Spark is an open source big data processing framework built around speed, ease of use and sophisticated analytics. Spark is written in Scala Programming Language and runs on Java Virtual Machine (JVM) environment. Creating and running Spark programs are faster due to its less read and write of code. It supports other languages such as Java, Python and R etc. for developing applications. In addition to Map and Reduce operations, it supports SQL queries, streaming data, machine learning and graph data processing [2] as shown in Figure 1.

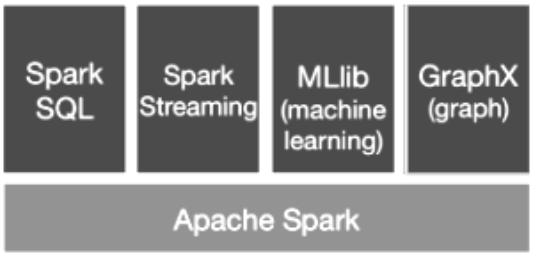


Figure 1. Spark Unified Stack

Spark operates on RDDs (Resilient Distributed Dataset) which is an in-memory data structure. Each RDD represents a chunk of the data which is partitioned across the data nodes in the cluster. RDDs are immutable and a new one is created when transformations are applied. RDDs are operated in parallel using transformations/actions like mapping, filtering etc. These operations are performed simultaneously on all the partitions in parallel. RDDs are resilient, if a partition is lost due to a node crash, it can be reconstructed from the original sources.

Spark Streaming can be used for processing the real-time streaming data. This is based on micro batch style of computing and processing. Spark Streaming provides as abstraction called DStream (Discrete Streams) which is a continuous stream of data. DStreams are created from input data stream or from sources such as Kafka, Flume or by applying operations on other DStreams. A DStream is essentially a sequence of RDDs.

Spark SQL provides the capability to expose the Spark datasets over JDBC API and allow running the SQL like queries on Spark data using traditional BI[1] and visualization tools. Spark SQL can read both SQL and NoSQL data sources. StreamSQL is a Spark component that combines Catalyst[2] and Spark Streaming to perform SQL queries on DStreams.

Spark MLlib [3] is Spark's scalable machine learning library consisting of common learning algorithms and utilities including classification, regression, clustering, collaborative filtering, dimensionality reduction as well as underlying optimization primitives.

Spark GraphX [3] is the new (alpha) Spark API for graphs and graph-parallel computation. At a high level, GraphX extends the Spark RDD by introducing the Resilient Distributed Property Graph: a directed multi-graph with properties attached to each vertex and edge. To support

[1] Business Intelligence (BI) is a broad category of computer software solutions that enables a company or organization to gain insight into its critical operations through reporting applications and analysis tools.
[2] Catalyst Optimizer optimizes the Relational algebra and expressions. It does Query optimization.

graph computation, GraphX exposes a set of fundamental operators (such as subgraph, joinVertices, and aggregateMessages etc.) as well as an optimized variant of the Pregel API. In addition, GraphX includes a growing collection of graph algorithms and builders to simplify graph analytics tasks.

3 SPARK SQL

SparkSQL is a distributed and fault tolerant query engine. It allows users to run interactive queries on structured and semi-structured data**.** Spark SQL can support Batch or Streaming SQL. It runs SQL/Hive queries, connects existing BI tools to Spark through JDBC. It binds Python, Scala, Java and R. Spark SQL's data source API can read and write DataFrames[3] using a variety of formats such as JSON, CSV, Parquet, MySQL, HDFS, HIVE, HBASE, Avro or JDBC etc.

3.1 Capabilities of SparkSQL [5]

- **Seamless Integration**
  Spark SQL allows us to write queries inside Spark programs, using either SQL or a DataFrame API. We can apply normal spark functions (map, filter, ReduceByKey etc) to SQL query results.
- **Supports variety of Data Formats and Sources**
  Data Frames and SQL provide connection to access a variety of data sources, including Hive, Avro, Parquet, Cassandra, CSV, ORC, JSON or JDBC. We can load, query and join data across these sources.
- **Hive Compatibility**
  We neither to make any changes to our data in existing hive metastore to make it work with Spark nor to change our hive queries. Spark SQL reuses the Hive frontend and metastore, giving full compatibility with existing Hive data, queries and UDF[4]s.
- **Standard Connectivity for JDBC or ODBC**
  A server mode provides industry standard JDBC and ODBC connectivity for Business Intelligence tools. We can use our existing BI tools like tableau.
- **Performance Scalability**
  At the core of Spark SQL is the Catalyst optimizer, which leverages advanced programming language features like columnar storage and code generation in a novel way to build an extensible query optimizer. It scales to thousands of nodes and multi hour queries using the Spark engine, which provides full mid-query fault tolerance.

4 MONGODB

MongoDB is an open-source *document database* that provides high performance, high availability and automatic scaling. MongoDB obviates the need for an Object Relational Mapping (ORM) to facilitate development. A record in MongoDB is a document, which is a data structure composed of field and value pairs. MongoDB documents are similar to JSON objects. The values of fields may include other documents, arrays and arrays of documents.

Table 2. Comparison of RDBMS terminology with MongoDB [9]

RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
Column	Field

[3] DataFrame is a distributed collection of rows organized into named columns. It is an abstraction for selecting, filtering, aggregating and plotting structured data.

[4] User-Defined Functions (UDF) is a feature of Spark SQL to define new Column-based functions that extend the vocabulary of Spark SQL's DSL to transform Datasets.

Table Join	Embedded Documents
Primary Key	Primary Key (Default key *_id* is provided by MongoDB itself)

MongoDB stores documents in collections. Collections are analogous to tables in relational databases. Unlike a table, however, a collection does not require its documents to have the same schema. In MongoDB, documents stored in a collection must have a unique *_id* field that acts as a primary key. MongoDB stores data records as BSON[5] documents.

4.1 Advantages of MongoDB over RDBMS

- Schema less (Number of fields, content and size of the document can be differing from one document to another.)
- Structure of a single object is clear
- No complex joins
- Deep query-ability (MongoDB supports dynamic queries on documents using a document-based query language.)
- Tuning
- Ease to scale
- Conversion / mapping of application objects to database objects is not needed
- Uses internal memory for storing the (windowed) working set, enabling faster access of data

5 MONGODB SPARK CONNECTOR

The MongoDB Spark Connector provides integration between MongoDB and Apache Spark. With the connector, we have access to all Spark libraries for use with MongoDB datasets: Datasets for analysis with SQL (benefiting from automatic schema inference), streaming, machine learning, and graph APIs. We can also use the connector with the Spark Shell.

The connector enables developers to build more functional applications faster and with less complexity, using a single integrated analytics and database technology stack. With industry estimates assessing that data integration consumes 80% of analytics development, the connector enables data engineers to eliminate the requirement for shuttling data between separate operational and analytics infrastructure. Each of these systems demands their unique configuration, maintenance and management requirements.

Written in Scala, Apache Spark's native language, the connector provides a more natural development experience for Spark users. The connector exposes all of Spark's libraries, enabling MongoDB data to be materialized as DataFrames and Datasets for analysis with machine learning, graph, streaming and SQL APIs, further benefiting from automatic schema inference. The connector also takes advantage of MongoDB's aggregation pipeline and rich secondary indexes to extract, filter and process only the range of data it needs.

To maximize performance across large, distributed data sets, the MongoDB Connector for Apache Spark can co-locate Resilient Distributed Datasets (RDDs) with the source MongoDB node, thereby minimizing data movement across the cluster and reducing latency as shown in Figure 2.

[5] BSON is a binary representation of JSON documents, though it contains more data types than JSON.

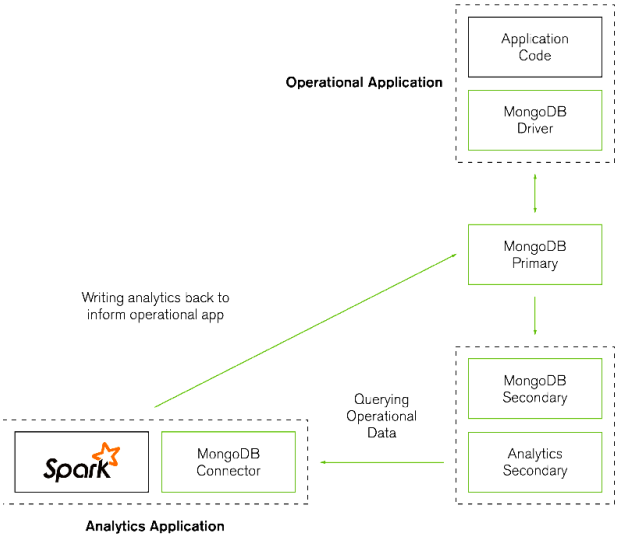


Figure 2. Using MongoDB Replica Sets to Isolate Analytics from Operational Workloads

6 EXPERIMENTATION

This experimentation demonstrates on how the MongoDB Connector for Apache Spark can access to all Spark libraries to analyze MongoDB dataset [10][11][12].

MongoDB Spark Connector is compatible with MongoDB 2.6 or later, Apache Spark 1.6.x and Scala 2.10.x (if using the mongo-spark-connector_2.10 package) or Scala 2.11.x (*if* using the mongo-spark connector_2.11 package). This experimentation uses the Spark shell with Mongo shell[6].

6.1 Proposed Model:

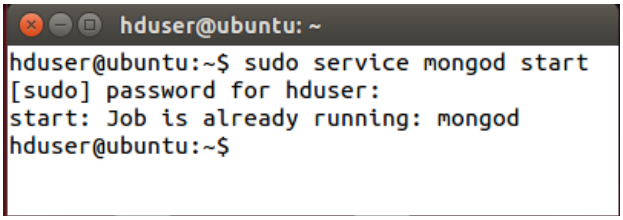
The MongoDB Connector is a plugin for both Hadoop and Spark that provides the ability to use MongoDB as an input source and/or an output destination for jobs running in both environments.

- Start the Mongo shell
- Start the MongoDB Spark Connector, while starting the Spark shell (with Scala)
- Create a BSON file in Spark
- Save the BSON file (from Spark) into MongoDB (as a Collection)
- When needed, Load the MongoDB collection into Spark (as a DataFrame)
- Perform required SQL queries on the DataFrame of Spark
- Store the output of the SQL queries (from the DataFrame of Spark) into MongoDB (as a Collection)

6.2 Model Implementation:

Step (1): To Start Mongo Shell

```
hduser@ubuntu:~$ sudo service mongod start
```

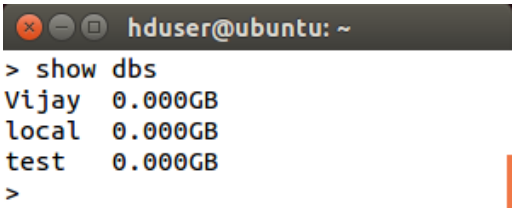



```
hduser@ubuntu:~$ mongo
MongoDB shell version: 3.2.8
connecting to: test
>
```

Note: When we run mongo without any arguments, the mongo shell will attempt to connect to the MongoDB instance running on the localhost interface on port 27017.

Step (2): To show Current databases of MongoDB

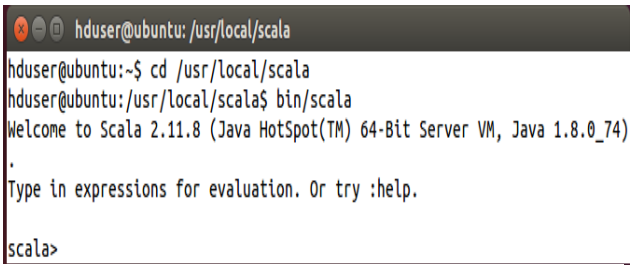
```
> show dbs
```



Step (3): To Start Scala

```
hduser@ubuntu:~$ cd /usr/local/scala
hduser@ubuntu:/usr/local/scala$
bin/scala
```

[6] The mongo shell is an interactive JavaScript interface to MongoDB and is a component of the MongoDB package. We can use the mongo shell to query and update data as well as perform administrative operations.



Step (4): To Start MongoDB Spark Connector

```
hduser@ubuntu:/usr/local/scala$ cd ../spark
hduser@ubuntu:/usr/local/spark$ bin/spark-shell --conf "spark.mongodb.input.uri=mongodb://127.0.0.1/Sivijaya.primer?readPreference=primaryPreferred" \--conf "spark.mongodb.output.uri=mongodb://127.0.0.1/Sivijaya.primer" \--packages org.mongodb.spark:mongo-spark-connector_2.10:1.0.0
```

Note:

- The *bin/spark-shell* to start the Spark shell.
- The *--conf* option to configure the MongoDB Spark Connector. These settings configure the SparkConf object.
- The *spark.mongodb.input.uri* specifies the MongoDB server address (127.0.0.1), to connect the database (Sivijaya) and the collection (primer) from which to read data with the read preference.
- The *spark.mongodb.output.uri* specifies the MongoDB server address (127.0.0.1), to connect the database (Sivijaya) and the collection (primer) to which to write data.
- The *--packages* option to download the MongoDB Spark Connector package mongo-spark-connector_2.10 to use with Scala 2.10.x

```
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.6.2
      /_/

Using Scala version 2.10.5 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_74)
Type in expressions to have them evaluated.
Type :help for more information.
16/08/23 08:54:25 WARN Utils: Your hostname, ubuntu resolves to a loopback address: 127.0.1.1; using 192.168.127.130 instead (on interface eth0)
16/08/23 08:54:25 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
Spark context available as sc.
16/08/23 08:54:42 WARN Connection: BoneCP specified but not present in CLASSPATH (or one of dependencies)
16/08/23 08:54:44 WARN Connection: BoneCP specified but not present in CLASSPATH (or one of dependencies)
16/08/23 08:54:53 WARN ObjectStore: Version information not found in metastore. hive.metastore.schema.verification is not enabled so recording the schema version 1.2.0
16/08/23 08:54:53 WARN ObjectStore: Failed to get database default, returning NoSuchObjectException
16/08/23 08:55:03 WARN Connection: BoneCP specified but not present in CLASSPATH (or one of dependencies)
16/08/23 08:55:05 WARN Connection: BoneCP specified but not present in CLASSPATH (or one of dependencies)
SQL context available as sqlContext.

scala>
```

Step (5): To import necessary packages

```
scala> import com.mongodb.spark._
scala> import org.bson.Document
scala> import com.mongodb.spark.sql._
scala> import org.apache.spark.sql.SQLContext
scala> val sqlContext = SQLContext.getOrCreate(sc)
```

Step (6): To create a BSON file in Spark, We took a sample JSON file named **primer-dataset.json**[7]

```
scala> val mydocs = """
     | {"address": {"building": "1007", "coord": [-73.856077, 40.848447], "street": "Morris Park Ave", "zipcode": "10462"}, "borough": "Bronx", "cuisine": "Bakery", "grades": [{"date": {"$date": 1393804800000}, "grade": "A", "score": 2}, {"date": {"$date": 1378857600000}, "grade": "A", "score": 6}, {"date": {"$date": 1358985600000}, "grade": "A", "score": 10}, {"date": {"$date": 1322006400000}, "grade": "A", "score": 9}, {"date": {"$date": 1299715200000}, "grade": "B", "score": 14}], "name": "Morris Park Bake Shop", "restaurant_id": "30075445"}
     | {"address": {"building": "469", "coord": [-73.961704, 40.662942], "street": "Flatbush Avenue", "zipcode": "11225"}, "borough":
```

[7] https://raw.githubusercontent.com/mongodb/docs-assets/primer-dataset/primer-dataset.json

```
"Brooklyn", "cuisine": "Hamburgers",
"grades": [{"date": {"$date":
1419897600000}, "grade": "A", "score":
8}, {"date": {"$date": 1404172800000},
"grade": "B", "score": 23}, {"date":
{"$date": 1367280000000}, "grade":
"A", "score": 12}, {"date": {"$date":
1336435200000}, "grade": "A", "score":
12}], "name": "Wendy'S",
"restaurant_id": "30112340"}
     | {"address": {"building": "351",
"coord": [-73.98513559999999,
40.7676919], "street": "West   57
Street", "zipcode": "10019"},
"borough": "Manhattan", "cuisine":
"Irish", "grades": [{"date": {"$date":
1409961600000}, "grade": "A", "score":
2}, {"date": {"$date": 1374451200000},
"grade": "A", "score": 11}, {"date":
{"$date": 1343692800000}, "grade":
"A", "score": 12}, {"date": {"$date":
1325116800000}, "grade": "A", "score":
12}], "name": "Dj Reynolds Pub And
Restaurant", "restaurant_id":
"30191841"}
     | {"address": {"building":
"2780", "coord": [-73.98241999999999,
40.579505], "street": "Stillwell
Avenue", "zipcode": "11224"},
"borough": "Brooklyn", "cuisine":
"American ", "grades": [{"date":
{"$date": 1402358400000}, "grade":
"A", "score": 5}, {"date": {"$date":
1370390400000}, "grade": "A", "score":
7}, {"date": {"$date": 1334275200000},
"grade": "A", "score": 12}, {"date":
{"$date": 1318377600000}, "grade":
"A", "score": 12}], "name": "Riviera
Caterer", "restaurant_id": "40356018"}
          | {"address": {"building":
"8825", "coord": [-73.8803827,
40.7643124], "street": "Astoria
Boulevard", "zipcode": "11369"},
"borough": "Queens", "cuisine":
"American ", "grades": [{"date":
{"$date": 1416009600000}, "grade":
"Z", "score": 38}, {"date": {"$date":
1398988800000}, "grade": "A", "score":
10}, {"date": {"$date":
1362182400000}, "grade": "A", "score":
7}, {"date": {"$date": 1328832000000},
"grade": "A", "score": 13}], "name":
"Brunos On The Boulevard",
"restaurant_id":
"40356151"}""".trim.stripMargin.split(
"[\\r\\n]+").toSeq
```

```
hduser@ubuntu: /usr/local/spark

scala> val mydocs="""
     | {"address": {"building": "1007", "coord": [-73.856077, 40.848447],
"street": "Morris Park Ave", "zipcode": "10462"}, "borough": "Bronx", "cui
sine": "Bakery", "grades": [{"date": {"$date": 1393804800000}, "grade": "A
", "score": 2}, {"date": {"$date": 1378857600000}, "grade": "A", "score":
6}, {"date": {"$date": 1358985600000}, "grade": "A", "score": 10}, {"date"
: {"$date": 1322006400000}, "grade": "A", "score": 9}, {"date": {"$date":
1299715200000}, "grade": "B", "score": 14}], "name": "Morris Park Bake Sho
p", "restaurant_id": "30075445"}
     |
     | {"address": {"building": "469", "coord": [-73.961704, 40.662942], "
street": "Flatbush Avenue", "zipcode": "11225"}, "borough": "Brooklyn", "c
uisine": "Hamburgers", "grades": [{"date": {"$date": 1419897600000}, "grad
e": "A", "score": 8}, {"date": {"$date": 1404172800000}, "grade": "B", "sc
ore": 23}, {"date": {"$date": 1367280000000}, "grade": "A", "score": 12},
{"date": {"$date": 1336435200000}, "grade": "A", "score": 12}], "name": "W
endy'S", "restaurant_id": "30112340"}
     |
     | {"address": {"building": "6409", "coord": [-74.00528899999999, 40.6
28886], "street": "11 Avenue", "zipcode": "11219"}, "borough": "Brooklyn",
 "cuisine": "American ", "grades": [{"date": {"$date": 1405641600000}, "gr
ade": "A", "score": 12}, {"date": {"$date": 1375142400000}, "grade": "A",
"score": 12}, {"date": {"$date": 1360713600000}, "grade": "A", "score": 11
}, {"date": {"$date": 1345075200000}, "grade": "A", "score": 2}, {"date":
{"$date": 1313539200000}, "grade": "A", "score": 11}], "name": "Regina Cat
erers", "restaurant_id": "40356649"}
     |
     | {"address": {"building": "1839", "coord": [-73.9482609, 40.6408271]
, "street": "Nostrand Avenue", "zipcode": "11226"}, "borough": "Brooklyn",
 "cuisine": "Ice Cream, Gelato, Yogurt, Ices", "grades": [{"date": {"$date
": 1405296000000}, "grade": "A", "score": 12}, {"date": {"$date": 13734144
00000}, "grade": "A", "score": 8}, {"date": {"$date": 1341964800000}, "gra
de": "A", "score": 5}, {"date": {"$date": 1329955200000}, "grade": "A", "s
core": 8}], "name": "Taste The Tropics Ice Cream", "restaurant_id": "40356
731"}""".trim.stripMargin.split("[\\r\\n]+").toSeq
```

Step (7): To save the BSON file into MongoDB collection

```
scala>
sc.parallelize(mydocs.map(Document.par
se)).saveToMongoDB()
```

Step (8): To view the stored collection (primer) of the database (Sivijaya) from MongoDB

```
> show dbs
> use Sivijaya
> show collections
> db.primer.find()
```

```
hduser@ubuntu: ~
> show dbs
Sivijaya  0.000GB
Vijay     0.000GB
local     0.000GB
test      0.000GB
> use Sivijaya
switched to db Sivijaya
> show collections
primer
> db.primer.find()
{ "_id" : ObjectId("57bc785fedaab00ce7139da9"), "address" : { "building" : "1007", "coord" : [ -73.856077, 40
 Ave", "zipcode" : "10462" }, "borough" : "Bronx", "cuisine" : "Bakery", "grades" : [ { "date" : ISODate("201
, "score" : 2 }, { "date" : ISODate("2013-09-11T00:00:00Z"), "grade" : "A", "score" : 6 }, { "date" : ISODate
: "A", "score" : 10 }, { "date" : ISODate("2011-11-23T00:00:00Z"), "grade" : "A", "score" : 9 }, { "date" : I
rade" : "B", "score" : 14 } ], "name" : "Morris Park Bake Shop", "restaurant_id" : "30075445" }
{ "_id" : ObjectId("57bc785fedaab00ce7139daa"), "address" : { "building" : "469", "coord" : [ -73.961704, 40.
nue", "zipcode" : "11225" }, "borough" : "Brooklyn", "cuisine" : "Hamburgers", "grades" : [ { "date" : ISODat
 : "A", "score" : 8 }, { "date" : ISODate("2014-07-01T00:00:00Z"), "grade" : "B", "score" : 23 }, { "date" :
grade" : "A", "score" : 12 }, { "date" : ISODate("2012-05-08T00:00:00Z"), "grade" : "A", "score" : 12 } ], "n
: "30112340" }
{ "_id" : ObjectId("57bc785fedaab00ce7139dab"), "address" : { "building" : "351", "coord" : [ -73.98513559999
t   57 Street", "zipcode" : "10019" }, "borough" : "Manhattan", "cuisine" : "Irish", "grades" : [ { "date" :
grade" : "A", "score" : 2 }, { "date" : ISODate("2013-07-22T00:00:00Z"), "grade" : "A", "score" : 11 }, { "da
Z"), "grade" : "A", "score" : 12 }, { "date" : ISODate("2011-12-29T00:00:00Z"), "grade" : "A", "score" : 12 }
Restaurant", "restaurant_id" : "30191841" }
```

```
> db.primer.find().pretty()
```

```
hduser@ubuntu: ~
> db.primer.find().pretty()
{
        "_id" : ObjectId("57bc785fedaab00ce7139da9"),
        "address" : {
                "building" : "1007",
                "coord" : [
                        -73.856077,
                        40.848447
                ],
                "street" : "Morris Park Ave",
                "zipcode" : "10462"
        },
        "borough" : "Bronx",
        "cuisine" : "Bakery",
        "grades" : [
                {
                        "date" : ISODate("2014-03-03T00:00:00Z"),
                        "grade" : "A",
                        "score" : 2
                },
                {
                        "date" : ISODate("2013-09-11T00:00:00Z"),
                        "grade" : "A",
                        "score" : 6
                },
```

Step (9): To load the stored collection (primer) from MongoDB into Spark (as DataFrame)

```
scala> val
sample=MongoSpark.load(sqlContext)

scala> val sample=MongoSpark.load(sqlContext)
sample: org.apache.spark.sql.DataFrame = [_id: struct<oid:string>, address
: struct<building:string,coord:array<double>,street:string,zipcode:string>
, borough: string, cuisine: string, grades: array<struct<date:timestamp,gr
ade:string,score:int>>, name: string, restaurant_id: string]

scala>
```

Step (10): To Store (or To Register) the DataFrame into Table

```
scala>
sample.registerTempTable("primer")
```

Step (11): To perform SQL queries on the loaded data (registered as Table) in Spark

```
scala> val
mydf1=sqlContext.sql("select
restaurant_id,name,borough,cuisine
from primer where cuisine == 'American
'")
scala> mydf1.show()
```

```
hduser@ubuntu: /usr/local/spark

scala> sample.registerTempTable("primer")

scala> val mydf1=sqlContext.sql("select restaurant_id,name,borough,cuisine
 from primer where cuisine == 'American '")
mydf1: org.apache.spark.sql.DataFrame = [restaurant_id: string, name: stri
ng, borough: string, cuisine: string]

scala> mydf1.show()
+------------+------------------+--------+---------+
|restaurant_id|              name| borough|  cuisine|
+------------+------------------+--------+---------+
|    40356018|    Riviera Caterer|Brooklyn|American |
|    40356151|Brunos On The Bou...|  Queens|American |
|    40356649|    Regina Caterers|Brooklyn|American |
+------------+------------------+--------+---------+
```

```
scala> val
mydf2=sqlContext.sql("select
restaurant_id,address,grades,borough,c
uisine from primer where borough ==
'Queens' or cuisine == 'American '")
scala> mydf2.show()
```

```
hduser@ubuntu: /usr/local/spark
scala> val mydf2=sqlContext.sql("select restaurant_id,address,grades,borou
gh,cuisine from primer where borough == 'Queens' or cuisine == 'American '
")
mydf2: org.apache.spark.sql.DataFrame = [restaurant_id: string, address: s
truct<building:string,coord:array<double>,street:string,zipcode:string>, g
rades: array<struct<date:timestamp,grade:string,score:int>>, borough: stri
ng, cuisine: string]

scala> mydf2.show()
+------------+--------------------+--------------------+--------+--------
-----+
|restaurant_id|             address|              grades| borough|      cu
isine|
+------------+--------------------+--------------------+--------+--------
-----+
|    40356018|[2780,WrappedArra...|[[2014-06-09 17:0...|Brooklyn|    Amer
ican |
|    40356068|[97-22,WrappedArr...|[[2014-11-23 16:0...|  Queens|Jewish/K
osher|
|    40356151|[8825,WrappedArra...|[[2014-11-14 16:0...|  Queens|    Amer
ican |
|    40356649|[6409,WrappedArra...|[[2014-07-17 17:0...|Brooklyn|    Amer
ican |
+------------+--------------------+--------------------+--------+--------
-----+
```

Step (12): To save the outputs of the (Step 11) SQL queries from Spark into MongoDB

```
scala>
mydf1.write.option("collection",
"Query1_output").mode("overwrite").for
mat("com.mongodb.spark.sql").save()

scala>
mydf2.write.option("collection",
"Query2_output").mode("overwrite").for
mat("com.mongodb.spark.sql").save()
```

Step (13): To view the stored outputs of the Step 12) SQL queries (as collections) from the database (Sivijaya) of MongoDB

```
> use Sivijaya
> show collections
```


```
> use Sivijaya
switched to db Sivijaya
> show collections
Query1_output
Query2_output
primer
>
```

```
> db.Query1_output.find()
> db.Query2_output.find()
```

Step (14): To Stop Mongo Shell

```
hduser@ubuntu:~$ sudo service mongod stop
mongod stop/waiting
```

7 INTEGRATING MONGODB & SPARK ANALYTICS WITH BI TOOLS & HADOOP

Real time analytics generated by MongoDB and Spark can serve both online operational applications and offline reporting systems, where it can be blended with historical data and analytics from other data sources. To power dashboards, reports and visualizations, MongoDB offers integration with more of the leading BI and Analytics platforms than any other non-relational database, including tools from Actuate, Alteryx, Informatica, Qliktech, and Talend etc.

A range of ODBC/JDBC connectors[8] for MongoDB provide integration with additional analytics and visualization platforms including Tableau [13] and others. At the most fundamental level, each connector provides read and write access to MongoDB. The connector enables the BI platform to access MongoDB documents, parse them before blend them with other data. Result sets can also be written back to MongoDB if required. More advanced functionality available in some BI connectors [14] includes integration with the MongoDB aggregation framework for in-database analytics and summarization, schema discovery and intelligent query routing within a replica set.

8 CONCLUSION

The MongoDB Connector is a plugin for both Hadoop and Spark that provides the ability to use MongoDB as an input source and/or an output destination for jobs running in both environments. The connector directly integrates Spark with MongoDB and has no dependency on also having a Hadoop cluster running.

Input and output classes are provided allowing users to read and write against both live MongoDB collections and against BSON (Binary JSON) files that are used to store MongoDB snapshots. Also, JSON formatted queries and projections can be used to filter the input collection, which uses a method in the connector to create a Spark RDD from the MongoDB collection.

[8] http://www.simba.com/
https://www.progress.com/datadirect-connectors

MongoDB natively provides a rich analytics framework within the database. Multiple connectors are also available to integrate Spark with MongoDB to enrich analytics capabilities by enabling analysts to apply libraries for machine learning, streaming and SQL to MongoDB data.

REFERENCES

[1] http://spark.apache.org/research.html

[2] http://spark.apache.org/

[3] https://www.infoq.com/articles/apache-spark-introduction

[4] http://www.slideshare.net/SparkSummit/adding-complex-data-to-spark-stackneeraja-rentachintala

[5] Poonam Ligade, "Processing JSON data using Spark SQL Engine: DataFrame API", October 2015, Article in edupristine

[6] http://www.kdnuggets.com/2015/09/spark-sql-real-time-analytics.html

[7] http://www.slideshare.net/databricks/spark-dataframes-simple-and-fast-analytics-on-structured-data-at-spark-summit-2015

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

[9] http://www.tutorialspoint.com/mongodb

[10] https://docs.mongodb.com/

[11] https://www.mongodb.com/press/mongodb-enables-advanced-real-time-analytics-on-fast-moving-data-with-new-connector-for-apache-spark

[12] A MongoDB white paper, "Apache Spark and MongoDB – Turning Analytics into Real-Time Action"

[13] http://www.tableau.com/

[14] http://www.simba.com/webinar/connect-tableau-big-data-source/