Interactive big data analysis with R, SparkR and MongoDB: a friendly walkthrough

Milano, 27 October 2016
Thimoty Barbieri
ITECH Engineering
thimoty@thimoty.it

Marco Biglieri
Università di Pavia
marco.biglieri90@gmail.com
About us

- Thimoty Barbieri is contract professor of Software Engineering at the University of Pavia. He is a Certified Mongo DB Developer and Administrator. Among his research interests are big data applications in the financial domain. He is an enthousiast R user.

- Marco Biglieri is a Software Engineering undergraduate at University of Pavia. He is currently working at his Master thesis on Big Data processing architectures for interactive analysis of financial data based on the FIX protocol, using R, SparkR and Mongo DB
Introduction

Spark and MongoDB are a fantastic opportunity to enhance R with big-processing and big-data features – all in open source! We present a walkthrough to set up a working environment which will put Spark's parallel processing and Mongo's huge-data handling capabilities at your fingertips in R. We will discuss the main features of SparkR with MongoDB and provide a toy example.
Our goal is to provide R users with a system for interactive big data analytics, using an architecture composed of the following layers:

- **Application Layer**: uses R to provide an interactive analysis environment.
- **Processing Layer**: uses Spark for efficient parallel execution of R scripts.
- **Data Layer**: uses mongodb to efficiently handle huge amounts of data.
MongoDB is a NoSql document oriented database, designed to manage huge amounts of data providing high performance, high availability and consistency of data.

MongoDB uses a flexible JSON-based document data model, with a schema-less approach. MongoDB provides a lot of useful features like indexing, a rich query language, an aggregation framework, replicas and sharding.
Apache Spark is an open source engine for big data processing designed to be:

- **Fast**, 100x faster than Apache Hadoop by exploiting in-memory parallel computing
- **General purpose**, covers a wide range of workloads that previously required separate systems (ETL, queries, machine learning, streaming)

Spark allows to integrate many datasources (HDFS, MySQL, MongoDB, etc), cluster managers (Yarn, EC2, Mesos) and supports many languages including R, Java, Python and Scala.
What you can do with SparkR

- Connect R and Spark
- Use R’s strength for visualisation, statistics, analysis
- Use Spark’s RDD concept and functional lambdas
- Run local R functions in a distributed way with Spark.lapply
MongoDB and Spark are designed for horizontal scaling. This allows to improve performance by simply adding commodity servers to a cluster instead of using a more powerful system (vertical scaling).

In a cluster configuration:

- Spark transparently manages the execution of tasks generated from the R script on the nodes of the cluster (workers).
- MongoDB uses sharding techniques to manage huge amounts of data (data distributed in cluster nodes) and to maintain high throughput on operations.
A simple example

In this example, we show how to use the R-Spark-MongoDB architecture to fit linear models on big quantities of data stored in a mongodb database.

In this example we use a simplified, standalone Spark-MongoDB configuration.

This configuration has two main characteristics:

1. **MongoDb runs in single instance**, in this configuration there is a unique mongod instance for reading data, which is not sharded.
2. **Spark local mode**, in this configuration spark workers run on a single machine, and not in a cluster.
Execution Environment

The example is provided in virtual machine instance with the following hardware and software configuration.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Linux Mint Cinnamon 18 (Ubuntu 16.04 LTS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>1 (1.7Ghz) (2 threads)</td>
</tr>
<tr>
<td>RAM</td>
<td>4GB</td>
</tr>
</tbody>
</table>

In this environment we installed the following components:

<table>
<thead>
<tr>
<th>Component</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mongodb</td>
<td>3.2</td>
</tr>
<tr>
<td>Oracle Java</td>
<td>1.8.0</td>
</tr>
<tr>
<td>Spark</td>
<td>2.0.1 Prebuilt for Hadoop 2.7</td>
</tr>
<tr>
<td>R</td>
<td>3.3.1</td>
</tr>
<tr>
<td>R studio</td>
<td>0.99.903</td>
</tr>
</tbody>
</table>
Installation – MongoDB

To install MongoDB, we used the following commands in the ubuntu shell:

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv EA312927
echo "deb http://repo.mongodb.org/apt/ubuntu trusty/mongodb-org/3.2 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.2.list
sudo apt-get update
sudo apt-get install -y mongodb-org
```

We have correctly installed mongodb: a single instance of mongodb runs now in the system, and we can connect to it using the `mongo` command.
Spark requires a Java installation, to avoid compatibility problems we recommend using Oracle’s JDK.

To install Java, we used the following command in the ubuntu shell:

```
Sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
sudo echo oracle-java8-installer shared/accepted-oracle-license-v1-1 select true | sudo /usr/bin/debconf-set-selections
Sudo apt-get -y -q install oracle-java8-installer
Sudo update-java-alternatives -s java-8-oracle
```
To install Spark, download the package from the Spark site ([http://spark.apache.org/downloads.html](http://spark.apache.org/downloads.html)), choosing the appropriate spark version and operating system combination:

After extracting the package into a specific directory (e.g. /home/user/spark-2.0.1-bin-hadoop2.7), take note of it: the directory location is very important for the following configuration steps.
Installation – R & R studio

To install R, issue the following command in the ubuntu shell:

```
sudo add-apt-repository ppa:marutter/rrutter
sudo apt-get update
sudo apt-get install r-base r-base-dev
```

To install R studio, download the deb package from R studio site (https://www.rstudio.com/products/rstudio/download/)

After downloading, install the deb package using regular ubuntu package installation.
After installing all components necessary in our architecture, it is time to configure these components to make them communicate to each other.

To connect Spark with MongoDB, add the following three lines inside the configuration file `spark-defaults.conf`, contained inside the `conf` folder in the spark folder (spark-2.0.1-bin-hadoop2.7/conf/spark-defaults.conf):

```plaintext
spark.jars.packages=org.mongodb.spark:mongo-spark-connector_2.11:2.0.0-rc0
spark.mongodb.input.uri=mongodb://127.0.0.1/bank.FixInfo
spark.mongodb.output.uri=mongodb://127.0.0.1/bank.FixInfo
```

These three lines specify:

1. Spark connector’s jar location, the jar is downloaded from maven repository
2. MongoDB instance address location,
3. Database name and collection name to read and write data (in our case bank is the database and FixInfo the collection)
Configuration – Spark R

To connect R with Spark, we need to include the SparkR library in R, so that we can use all functions provided by the SparkR library. We can do this directly in R:

```r
if (nchar(Sys.getenv("SPARK_HOME"))<1){
    Sys.setenv(SPARK_HOME = "/home/user/spark-2.0.1-bin-hadoop2.7")
}
library(SparkR, lib.loc = c(file.path(Sys.getenv("SPARK_HOME"), "R", "lib")))
```

Otherwise we can configure R studio to load automatically the SparkR library: modify the file /etc/R/Renviron:

- Adding the location of sparkR library /home/user/spark-2.0.1-bin-hadoop2.7/lib/R in the line R_LIBS_SITE
- Adding the location of spark directory

The final result is:

```bash
R_LIBS_SITE=${R_LIBS_SITE-'(something pre-set library path:/home/user/spark-2.0.1-bin-hadoop2.7/R/lib')}
SPARK_HOME='"/home/user/spark-2.0.1-bin-hadoop2.7"'
```
Linear Modeling Example - Dataset

Now that all components are configured, let’s write some R to fit linear models on data stored in the mongodb database.

The dataset contains transactional data:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>N. of documents</td>
<td>137003</td>
</tr>
<tr>
<td>Size on disk</td>
<td>1,012 GB</td>
</tr>
<tr>
<td>Average document size</td>
<td>7.34MB</td>
</tr>
</tbody>
</table>

This is a huge dataset to fit a regular R dataframe: we have tried loading the dataset in an R dataframe, reaching a memory limit. R’s capabilities are limited by the local machine’s memory footprint.

```r
# previously installed rjson with install.package("rjson")
>Library("rjson")
>data<-fromJSON(file="/home/user/FixInfo.json")
Errore cannot allocate vector of size 934.9Mb
```
Linear Modeling Example - Dataset

The dataset contains documents with a complex data structure composed of approximately 192 fields including nested documents, (blue) arrays of documents (green) and simple datatype. This structure would be very difficult to handle in a linear R dataframe.
Linear Modeling Example - Loading Data

To compute the linear model, the first step is to load data from the mongodb collection (set in configuration – bank.FixInfo) using the commands:

```
SparkR.session(sparkConfig = list(spark.driver.memory = "4g"))
data <- read.df("", source ="com.mongodb.spark.sql.DefaultSource")
```

This two lines instantiate:

- the `sparkR.session`, connects our R programs with Spark cluster and allows to set Spark Properties. In local mode is necessary set the driver memory (not the executors memory because in local mode driver and executors share the same memory) to increase the memory available (default 512mb). It’s also possible to set it from the `spark-defaults.conf` file.

- the `SparkDataFrame` (data) that is a distributed collection of data distributed between workers, inside this dataframe is loaded the mongodb collection.
Linear Modeling Example - Data Schema

SparkR provide two main features for data stored in a SparkDataframe:

- Handle data with a complex datastructure, each column of the SparkDataframe can contain multiple nested structures, in this way is not necessary split the dataset in multiple dataframes.

- Inference data schema, Spark can automatically infers the schema as we can see using:

```
> PrintSchema(data)
root
|-- FixInfo: struct (nullable = true)
  |-- BeDate: string (nullable = true)
  |-- BeTime: string (nullable = true)
  |-- BestFormulaInfo: struct (nullable = true)
    |-- BestFormulaMrkInfo: struct (nullable = true)
      |-- MrkInstance: array (nullable = true)
        |-- element: struct (containsNull = true)
          |-- CliFee: string (nullable = true)
          |-- Depth: array (nullable = true)
            |-- element: struct (containsNull = true)
              |-- AskPrice: string (nullable = true)
              |-- AskQty: string (nullable = true)
```

If Spark can’t infer the right datatype, it is possible to add explicit casting in the following steps.
SparkR provides a set of functions to transform data on the whole Spark dataframe. We use the select function to identify the two columns of interest to fit the linear model:

```r
```

After we use the function na.omit to clear the dataframe from rows with empty values:

```r
df<-na.omit(df)
```

The dataframe transformation:

<table>
<thead>
<tr>
<th>Price</th>
<th>TradesAvgPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 17.5100000</td>
<td>NA</td>
</tr>
<tr>
<td>2 4.4000000</td>
<td>NA</td>
</tr>
<tr>
<td>3 7.1750000</td>
<td>NA</td>
</tr>
<tr>
<td>4 null</td>
<td>NA</td>
</tr>
<tr>
<td>5 null</td>
<td>NA</td>
</tr>
<tr>
<td>6 null</td>
<td>NA</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Price</th>
<th>TradesAvgPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 103.00000000</td>
<td>102.79700000, 102.8000000</td>
</tr>
<tr>
<td>2 93.60000000</td>
<td>93.60500000, 93.7050000</td>
</tr>
<tr>
<td>3 2.46000000</td>
<td>2.45900000, 2.4590000</td>
</tr>
<tr>
<td>4 null</td>
<td>122.87000000, 123.5000000, 122.4900000</td>
</tr>
<tr>
<td>5 null</td>
<td>99.23000000, 99.2300000</td>
</tr>
<tr>
<td>6 null</td>
<td>160.74000000, 160.5400000</td>
</tr>
</tbody>
</table>
Linear Modeling Example – Data Transformation

The MrkInstance in our mongo dataset is an array of documents containing some fields including the TradesAvgPrice that we want to extract, using the explode function we split the array in multiple lines:

```
df$TradesAvgPrice <- explode(df$TradesAvgPrice)
```

Notice that even if MrkInstance is an array in mongodb, Spark doesn’t respect this structure, it creates an array inside the field TradesAvgPrice that contains the values of TradesAvgPrice assumed by each instance of the array MrkInstance.

MongoDB MrkInstance is an array, TradesAvgPrice field of the documents inside array.

<table>
<thead>
<tr>
<th>Price</th>
<th>TradesAvgPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 103.00000000</td>
<td>102.7970000, 102.8000000</td>
</tr>
<tr>
<td>2 93.60000000</td>
<td>93.6050800, 93.7050000</td>
</tr>
<tr>
<td>3 2.46000000</td>
<td>2.4590000, 2.4590000</td>
</tr>
<tr>
<td>4 null</td>
<td>122.8700000, 123.5000000, 122.4900000</td>
</tr>
<tr>
<td>5 null</td>
<td>99.2300000, 99.2300000</td>
</tr>
<tr>
<td>6 null</td>
<td>160.7400000, 160.5400000</td>
</tr>
</tbody>
</table>

Spark Dataframe, TradesAvgPrice contains array

Spark Dataframe after explode

<table>
<thead>
<tr>
<th>Price</th>
<th>TradesAvgPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 103.00000000</td>
<td>102.7970000</td>
</tr>
<tr>
<td>2 103.00000000</td>
<td>102.8000000</td>
</tr>
<tr>
<td>3 93.60000000</td>
<td>93.6050800</td>
</tr>
<tr>
<td>4 93.60000000</td>
<td>93.7050000</td>
</tr>
<tr>
<td>5 2.46000000</td>
<td>2.4590000</td>
</tr>
<tr>
<td>6 2.46000000</td>
<td>2.4590000</td>
</tr>
</tbody>
</table>
We filter the dataframe from null field:

```r
df <- filter(df, isNotNull(df$Price) & isNotNull(df$TradesAvgPrice))
```

Spark inferred the wrong data type for columns Price and TradesAvgPrice. We therefore cast them from string to double:

```r
df$Price = cast(df$Price, "double")
df$TradesAvgPrice = cast(df$TradesAvgPrice, "double")
```

We filter rows to delete possible outliers:

```r
df = filter(df, df$Price < 500)
df = filter(df, df$TradesAvgPrice < 500)
```
Now we can fit a linear model of the data using the function glm from the library MLLib of Spark.

```r
model<-glm(Price~TradesAvgPrice, family='gaussian',data=df)
```

As result we obtain the linear model (summary of model):

<table>
<thead>
<tr>
<th>$\text{devianceResiduals}$</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Coefficients}$</td>
<td>Estimate</td>
<td>Std. Error</td>
</tr>
<tr>
<td>(Intercept)</td>
<td>-0.002402987</td>
<td>0.006375454</td>
</tr>
<tr>
<td>TradesAvgPrice</td>
<td>0.9992961</td>
<td>9.248635e-05</td>
</tr>
</tbody>
</table>
Linear Modeling Example – Compute R squared

glm is different R's lm as it does not compute directly the R squared value. For this reason we have created the function computeRsquared(), using SparkR's distributed features:

```r
computeRsquared <- function(model, df, nameColumn) {
  mean <- collect(agg(df, AVG_column = mean(df[[nameColumn]])))$AVG_column
  predictions <- predict(model, newData = df)
  predictions <- transform(predictions, S_res = (predictions[[nameColumn]] -
    predictions$prediction)^2, S_tot = (predictions[[nameColumn]] - mean)^2)
  n <- nrow(df)
  residual <- collect(agg(predictions, SS_res = sum(predictions$S_res), ss_tot = sum(predictions$S_tot)))
  r2 <- 1 - (residual$SS_res/residual$ss_tot)
  return(r2)
}
```

In our case we obtain as result:

```r
> computeRsquared(model, df, "Price")
[1] 0.99
```
Finally we can plot the regression line obtained from the model and the related points; we use ggplot2 (previously installed in R and included in script using `library(ggplot2)`).

ggplot requires a R dataframe - we can obtain it using the `collect(df)` function, that collects all the elements of a `sparkDataframe` and coerces them in a R dataframe.

```r
slope=coef(summary(model))[[2]]
intercept=coef(summary(model))[[1]]
ggplot(collect(df),aes(x=TradesAvgPrice,y=Price))
  +geom_point()
  +geom_abline(slope=slope,intercept=intercept)
```
Linear Modeling Example – Plotting

We obtain the graph:

\[ R^2 = 0.99 \]
Linear Modeling Example - Plotting

Another example of linear model using two different variables (Price, BidPrice):

\[ R^2 = 0.44 \]
Results

The previous script executed with the following timing:

- 5:95 minutes to compute the linear model
- 02:18 minutes to compute R squared

Good performance, compared with the result without using spark and mongo (memory limit of R dataframe), it can be improved using more powerful platforms (we used a small virtual VM). We can also scale up with the quantity of data, leveraging Mongodb’s sharding techniques and Spark’s cluster mode.
Conclusions

SparkR provides a set of functions to manipulate and generate models from data. All SparkR instructions (read.df, select, dapply, lapply, gapply, filter, explode, cast, glm) in our example were executed in a standalone environment, but can be transparently scaled to cluster mode to improve performance and handle even bigger volumes.
Want to try it? Get in touch!

Get in touch to get a shared link to download our R-SparkR-MongoDB setup on a Linux mint virtual machine instance, configured and ready for the use.

Contacts:

Thimoty Barbieri, Marco Biglieri
ITECH Engineering
thimoty@thimoty.it