LAB 2 – SPARK / D-STREAM PROGRAMMING
SCIENTIFIC APPLICATIONS FOR IOT WORKSHOP

ICTP, Trieste, March 24th 2015

The objectives of this session are:

- Understand the Spark RDD programming model
- Familiarize with the Spark Shell interactive API
- Setup a scala Spark project
- Run Spark and D-Stream jobs

INTRODUCTION

The instructions contained in this sheet assume you are using a Hadoop environment preconfigured as in the Cloudera QuickStart VM version. You can complete this lab from your laptop (needs to have a reasonable amount of memory and 64bit OS), by downloading the VM from this link.

We will be using Scala as the language of choice for Spark, as it is the main supported language by the platform, and makes development self-contained.

Compilation will be done using Maven. We provide a fully functional skeleton for each part of the lab.

Over this lab sheet it will be convenient to have quick access to the Spark Programming guide, as well as the slides from this morning. In particular, quick reference to quick transformations.

There are multiple ways of writing anonymous functions (closures) in Scala. For a balance of expressivity and conciseness, we will follow in the examples the notation **x => f(x)** in all the code snippets.

http://spark.apache.org/docs/1.2.1/programming-guide.html

Prior to running this lab exercises, it is highly recommended to change Spark's default logging level from DEBUG to WARN. This will make the log output substantially less verbose, allowing us to see our trace lines as we execute Spark commands. In the Cloudera VM, we need to edit the log4j.properties file of Spark, which we can easily do with the following commands taking the provided template

```
sudo mv /etc/spark/conf/log4j.properties.template /etc/spark/conf/log4j.properties
sudo vi /etc/spark/conf/log4j.properties
```

When opening the file, modify the first line (root category), replacing DEBUG with WARN. Save and exit vi with the sequence **`Esc … :wq`**

Part 1: Spark programming using the Spark Shell

For simplicity reasons we will reuse the same dataset of the previous lab during the main Spark part of this coursework (The NASDAQ stock files).

In this first part of the lab we will use Spark's interactive shell to write line by line Spark commands, and retrieve back their input. We can start the spark-shell from anywhere in the command line by typing **`spark-shell`** in the command line.

After several debug lines detailing initialization, we will see a command prompt such as **`scala>`**. We are now going to start writing Spark commands one by one, and observe the results directly from the shell.

It is important to remember that Spark jobs are lazily evaluated. Transformations will only be applied the moment we request to retrieve a result from them by invoking an action on an RDD.

The Spark Shell already initialises a SparkContext object that we can use to create RDDs, under the handle **`sc`**.

We start our session by creating an RDD from the stocks dataset we used in the previous lab.

```
val stocks = sc.textFile("hdfs://quickstart.cloudera:8020/user/cloudera/stock")
```

Note that Scala does not require us to explicitly specify the types for the variables we will be using. As we are loading the file as a **`textFile`**, the RDD contains a List of Strings. Each one of the Strings represents one entry from the dataset.

When executing the command, the interpreter will notify us a reference has been created with the object. The feedback should be similar to:

```
stocks: org.apache.spark.rdd.RDD[String] = hdfs://quickstart.cloudera:8020/user/cloudera/stock
MappedRDD[1] at textFile at <console>:12
```

However, the file will not be actually loaded until we invoke an action on it. Let's do it by invoking **`count`**, which should return to our driver the number of lines in our base dataset. You will notice this operation takes longer, as Spark needs now to load the data, and perform the count over the parts of the RDD.

```
scala> stocks.count
res0: Long = ??????
```

Q: According to the Long value, how many records are contained in the dataset? Does this result match what you found yesterday in the Hadoop lab?

Once we have loaded an RDD, we can process it by invoking several transformations. In this case, we want to count the number of appearances for each unique key. Contrary to Map/Reduce, we will generally use more steps, as we decompose the operations into multiple elements. The flow will require the following three transformations:

`stocks.map -> symbols.map -> pairs.reduceByKey`

We can define the first map transformation as follows (selecting only the stock symbol from each line):

```
val symbols = stocks.map(x => x.split(",")(1))
```

We could check our program is working OK up to this point by evaluating already this new RDD. For example, we could use **`symbols.distinct.count`**, to transform the RDD into one with only the unique elements, and then return the number of unique symbols (hence the number of unique companies in the dataset).

Now we can convert each stock symbol to a pair, of value 1 (mirroring how the Map/Reduce equivalent works).

```
val pairs = symbols.map(x => ( x, 1) )
```

Once we have pairs, we can use the **`ReduceByKey`** transformation to define the final transform. Reduce is different to Map/Reduce, and more similar to its functional programming equivalent. We define how each pair of elements can be reduced to one, obtaining a single value for the list of all values that have been grouped by the same key:

```
val mins = pairs.reduceByKey( (a,b) => math.min(a,b) )
```

`mins` will have the desired result for our program. We could save it to HDFS with the action **`saveAsTextFile`** (for the path see how we created the lines RDD). As we are simply learning Spark commands, we will select three random sample results, by invoking the **`takeSample`** action

```
mins.takeSample(false, 3)
```

Part 2: Creating Spark projects

In this second part we will reimplement the second part of yesterday's lab in Spark. This way, we'll learn how to properly compile and package Spark programs. We will use Apache Maven for this purpose. Maven projects are governed by the pom.xml file in the root folder of the project. The provided file lab2.zip contains already the defined project structure inside the folder **`sparktests`**. pom.xml contains the project name, and most importantly the dependencies required to compile your program. Maven also imposes a rigid project structure. All the code you implement will be saved in the /src/main/scala folder. You should see a skeleton of a Spark program there, reproduced here for your convenience:

```
package bigdata.stock


import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf


object SparkStockCount {
  def main(args: Array[String]) {
    val sc = new SparkContext(new SparkConf().setAppName("Spark Stock Count"))
   //dataset format: exchange, stock_symbol, date, stock_price_open, stock_price_high,
stock_price_low, stock_price_close, stock_volume,stock_price_adj_close.
```

```
    val stocks = sc.textFile("hdfs://quickstart.cloudera:8020/user/cloudera/stock")
.. ..//write here your Spark program
  }
}
```

Taking the skeleton as the basis, you will have to add lines to your Spark program to compute. You can start by repeating the commands from the first part of the lab, and testing that compiled version, before writing the new program.

In the new program for this part, you need to compute, for each Stock symbol, the minimum close value among the dataset (same as yesterday). It might be easier to test your code first in the interactive shell, and edit the main class as you progress with the implementation. We provide the following hint: the following function can be mapped to the initial dataset, already creating the required pairs of values. You should be able to complete the rest of the implementation.

```
x => ( x.split(",")(1), x.split(",")(6).toDouble )
```

Once you have obtained the requested values you can either save the results to an HDFS folder (action **`saveAsTextFile`**, or use **`System.out.println`** to output the result of your actions).

In order to test your program, you can submit it to the framework after you have created the jar file (similarly to Hadoop). In order to compile the package, go to the root folder of the project (where the pom.xml file is located), and run the following command:

```
mvn clean package
```

If you get a success result, you will see a newly created target folder in your project, with a jar file containing the job. You can now run your program by invoking spark-submit:

```
spark-submit --class bigdata.stock.SparkStockCount --master local
target/sparkstocktests-0.0.1-SNAPSHOT.jar
```

If you get the correct results, you can further practice Spark by implementing the third job suggested in the previous lab session.

Part 3: Creating Spark D-Stream projects

The final part of the lab will show you how to run a working D-Stream setup that processes message from an incoming data stream. We will connect to the open MQTT test server (test.mosquitto.org), and process all the messages other users are generating to the platform.

The provided file lab2.zip contains already the defined project structure inside the folder **`sparkmqtt.`** The pom.xml is slightly more complex, as the composition of the packaging is different. Fortunately, you don't have to edit anything from this file. The only difference you will perceive is that you need to submit a different jar to start the Spark Streaming computation: **`uber-sparkmqtt-0.0.1-SNAPSHOT.jar`**, also generated in the target folder (this jar includes the libraries of D Stream that are not installed by default in the Cloudera Spark installation).

As we saw during the morning, stream processing is based on accessing an external stream of information. Thus, rather than accessing an HDFS folder, the Streaming program accesses an external data source. In this exercise we will connect to a remote MQTT server (located at test.mosquitto.org), and process all the incoming messages by listening to the topic '#'.

The setup is also slightly different to a batch Spark job. First we setup the Stream RDDs that we will be processing, and then we will invoke the **`start`** and **`awaitTermination`** methods to continuously process messages.

The provided class has a fully working program that performs a simple task: it counts the number of messages that are being generated in each window. The window length is 10 seconds, and computation is performed every 2 seconds. Find the corresponding method in the source code file, and identify these elements.

Compile the project with **`mvn clean package`**, and test the job in Spark. For convenience, we provide a working executable command for this program. This program accepts two arguments with the url of the MQTT broker, and the topic we are connecting to. You can modify these parameters to access any other MQTT system.

```
spark-submit --class bigdata.stream.SparkMQTTTest --master local[2]
target/uber-sparkmqtt-0.0.1-SNAPSHOT.jar tcp://test.mosquitto.org:1883
```

As you run the code, you will see marks at each 2 second slot, with a value outputting the number of messages that have been received over the last 10 seconds. You can close the program by sending the signal **`Control C.`**

Now, we are going to show how we can implement statistical computations based on all the received input on a sliding window. We provide two lines of code that can replace the count computation, with a different type of operation. Based on what you know of Spark, you should be able to interpret what this code will do by simply looking at it.

```
val slidingAverageSize = lines.map( x => (x.length,1) ).reduceByWindow (
(a,b) => (a._1 + b._1, a._2 + b._2), Seconds(10), Seconds(2))

slidingAverageSize.map(a => a._1/a._2).print()
```

Package the jar again, and rerun the program.

Q: What is the computation we are computing now at each window?