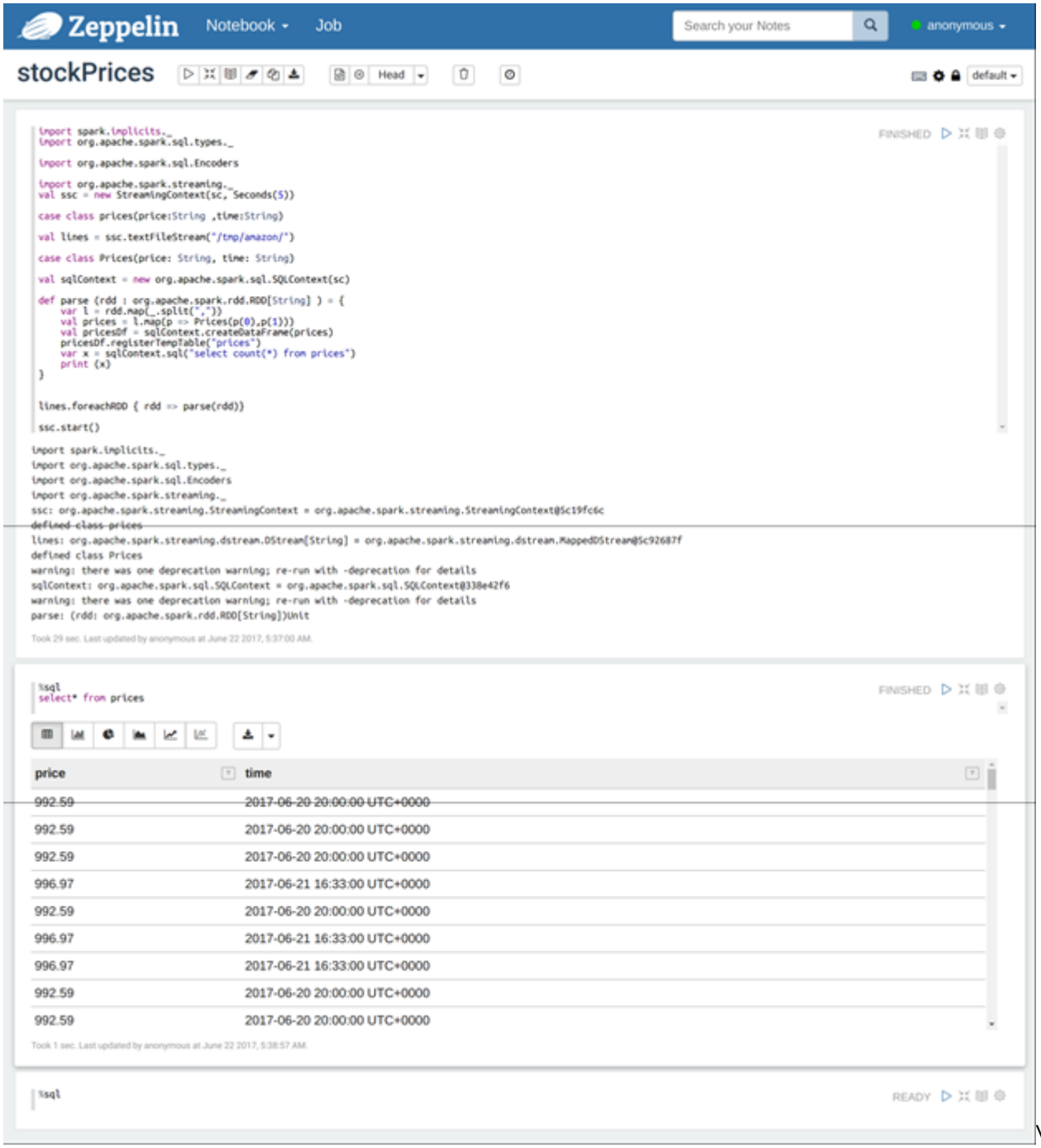
APACHE SPARK: WORKING WITH STREAMS

In the last two posts we wrote, we explained how to read data streaming from Twitter into Apache Spark by way of Kafka. Here we look at a simpler example of reading a text file into Spark as a stream.

We make a simple stock ticker that looks like the screen below when we run the code in Zeppelin.



Working with streaming data is quite different than working with data in batch. Spark Streaming reads streaming data into an object called DStreams. From these it periodically creates regular RDDs.

So it is just like working with RDDs with the important difference being that the input is not a file of some fixed size. Instead it is an unbounded stream, like Twitter tweets, diagnostic information streaming from a jet engine, stock prices coming off NASDAQ, etc.

We will use the stock price example to illustrate. We will use the Yahoo finance API. Install it using:

```
sudo pip install yahoo-finance
```

Then:

```
mkdir /tmp/amazon/
```

Copy this python code into a text file yahoo.py and run it:

```
python yahoo.py
```

This will download the Amazon stock price and write it to a file every five seconds.

```
from yahoo_finance import Share
import time
f = open('/tmp/amazon/amzn.csv', 'w')
while True:
amzn=Share("AMZN")
s = amzn.get_price() + "," + amzn.get_trade_datetime() + '\n'
print (s)
f.write (s)
f.flush()
time.sleep(5)
del amzn
```

Below is the entire Scala program that we will use to process this data as a stream. It is only a simple example. You are more likely to process streaming data by first writing it into Kafka, as we did with the Twitter example. But this simple example will help you understand how to work with streaming data.

Spark Streaming actually does not handle text files very well, as we will see. Below we explain that it is not able to keep track of changes to a text file. So we have to copy these files to a new folder in order to cause Spark to pick up the changes. That is not very practical for a streaming application. But again this is just a teaching example. You are more likely to use Kafka or read a data socket than process a text file. That said, however, this code would be suitable for a situation where an application rolls over log files, like a web server or LDAP server. Because there it periodically closes the open file and creates a new one.

Having started the Python program above, run **spark-shell** and post the entire code below into the Spark command-line interpreter. You will see that it will take a few minutes to start working. Then it will take over the screen and start echoing output.

Here we break the code up into sections to explain key parts.

Scala code

The key commands here are **new StreamingContext** and **ssc.textFileStream**.

The first takes the **SparkContext** (called **sc**, when using spark-shell) and creates a **StreamingContext**. It is the same idea as SparkContext, except it is for streaming data.

ssc.textFileStream looks inside the indicated folder and reads that data into DStreams without regards to any kind of format. So it is just reading bytes of information upon which we have not created any RDDs or other structures yet.

```
import spark.implicits._
import org.apache.spark.sql.types._
import org.apache.spark.sql.Encoders
import org.apache.spark.streaming._
val ssc = new StreamingContext(sc, Seconds(5))
val lines = ssc.textFileStream("/tmp/amazon/")
```

The Python program writes the stock price and time into the text file in this comma-delimited format:

992.59,2017-06-20 20:00:00 UTC+0000
992.59,2017-06-20 20:00:00 UTC+0000
992.59,2017-06-20 20:00:00 UTC+0000
992.59,2017-06-20 20:00:00 UTC+0000

The next section is familiar-looking code that we use to read each line in this type of text file and split each line by the comma using **rdd.map**. However, there is not notable difference which is that Spark creates multiple RDDs when you work with streaming data. That is why we have written the function **parse** with parameter type **org.apache.spark.rdd.RDD** so that **lines.foreachRDD { rdd => parse(rdd)}** shown below will process each RDD.

In other words, here is the crucial distinction. Spark returns a whole bunch of RDDs and we use **dstream.foreachRDD** to process each of them.

The rest of the code creates a **case class** so that we can turn it into a SQL dataframe using **sqlContext.createDataFrame**.

```
case class Prices(price: String, time: String)
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
def parse (rdd : org.apache.spark.rdd.RDD ) = {
var l = rdd.map(_.split(","))
val prices = l.map(p => Prices(p(0),p(1)))
val pricesDf = sqlContext.createDataFrame(prices)
pricesDf.registerTempTable("prices")
pricesDf.show()
var x = sqlContext.sql("select count(*) from prices")
println (x)
}
```

We call the function **parse** for each rdd created, as shown below.

lines.print() echoes the first 10 lines of the RDD in its raw format.

In the command line interpreter, nothing will happen until we run **ssc.start()**. At that point the program will continually look for new files added to /tmp/amazon and process each of them. It will stop accepting input from the keyboard and echo results to the screen.

```
lines.foreachRDD { rdd => parse(rdd)}
lines.print()
ssc.start()
```

When you run this code, wait a few seconds as it will take a while to start.

At first the program will echo the data it finds because we used **pricesDf.show()**.

```
+------+----------------------------+
|price|time|
+------+----------------------------+
|992.59|2017-06-20 20:00:...|
|992.59|2017-06-20 20:00:...|
|992.59|2017-06-20 20:00:...|
```

992.59	2017-06-20 20:00:...
992.59	2017-06-20 20:00:...
992.59	2017-06-20 20:00:...
996.97	2017-06-21 16:33:...

This is where things turn strange as the second iteration will show an empty dataframe. Why is that?

```
+-------+-------+
|price |time|
+-------+-------+
+-------+-------+
```

The reason the data frame is empty, even though the Python program is still writing data, is best understood by looking at the JavaDoc definition of the textFileStream written by Apache Spark.

> **public JavaDStream<String> textFileStream(String directory)**
>
> Create an input stream that monitors a Hadoop-compatible filesystem for new files and reads them as text files (using key as LongWritable, value as Text and input format as TextInputFormat). Files must be written to the monitored directory by "moving" them from another location within the same file system. File names starting with . are ignored.

In other words, it will not find any new data until you copy a new file manually into the /tmp/amazon folder. That is awkward and is something you could do with a shell script. You can try that manually by copying the file /tmp/amazon/amzn.csv to /tmp/amazon/a and then looking at the output of spark-shell again. You will see that it picks up the new data.

So we have given you an example of how to work with streaming data using Apache Spark. From here, we suggest to go back and review the Twitter example to reinforce what you have learned. Or write your own example using a tcp socket.