Spark Fundamentals

Creating applications using Spark SQL, MLlib, Spark Streaming, and GraphX

Contents

Creating applications using Spark SQL, MLlib, Spark Streaming, and GraphX

This lab exercise will show you how to create various applications using the Spark libraries. The advantage of these libraries is that they are tightly integrated with Spark, with very little overhead. Any time the Spark core API is updated, the new features and performance enhancements are passed down to these libraries as well. In this lab exercise, you will work with these various libraries to use Spark for specialized use cases.

After completing this hands-on lab, you should be able to:

   o   Create applications using the Spark built-in libraries

Allow 60 minutes to complete this section of lab.

1.1   Creating a Spark application using Spark SQL

Spark SQL provides the ability to write relational queries to be run on Spark. There is the abstraction SchemaRDD which is to create an RDD in which you can run SQL, HiveQL, and Scala. In this lab section, you will use SQL to find out the average weather and precipitation for a given time period in New York. The purpose is to demonstrate how to use the Spark SQL libraries on Spark.

__1.   The nycweather data is already on the HDFS under /tmp/labdata/sparkdata/.

__2.   Take a look at the nycweather data. Type in:

```
hadoop fs -cat /tmp/labdata/sparkdata/nycweather.csv
```

There are three columns in the dataset, the date, the mean temperature in Celsius, and the precipitation for the day. Since we already know the schema, we will infer the schema using reflection.

__3.   Launch the Scala shell: `$SPARK_HOME/bin/spark-shell`

__4.   You will first need to define the SparkSQL context. Do so by creating it from an existing SparkContex. Type in:

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

Remember that sc is the existing SparkContext that is created when the shell started.

__5.   Next, you need to import a library for creating a SchemaRDD. Type this in the spark-shell:

```
import sqlContext.createSchemaRDD
```

__6.   Create a case class in Scala that defines the schema of the table. Type in:

```
case class Weather(date: String, temp: Int, precipitation: Double)
```

__7.   Create the RDD of the Weather object:

```
val weather =
sc.textFile("/tmp/labdata/sparkdata/nycweather.csv").map(_.split(",")).
map(w => Weather(w(0), w(1).trim.toInt, w(2).trim.toDouble))
```

You first load in the file, and then you map it by splitting it up by the commas and then another mapping to get it into the Weather class.

__8.   Next you need to register the RDD as a table. Type in:

```
weather.registerTempTable("weather")
```

__9.   At the point, you are ready to create and run some queries on the RDD. You want to get a list of the hottest dates with some precipitation. Type in:

```
val hottest_with_precip = sqlContext.sql("SELECT * FROM weather WHERE
precipitation > 0.0 ORDER BY temp DESC")
```

__10. Normal RDD operations will work. Print the top hottest days with some precipitation out to the console:

```
hottest_with_precip.map(x => ("Date: " + x(0), "Temp : " + x(1),
"Precip: " + x(2))).top(10).foreach(println)
```

1.2   Creating a Spark application using MLlib

In this section, the Spark shell will be used to acquire the K-Means clustering for drop-off latitudes and longitudes of taxis for 3 clusters. The sample data contains a subset of taxi trips with hack license, medallion, pickup date/time, drop off date/time, pickup/drop off latitude/longitude, passenger count, trip distance, trip time and other information. As such, this may give a good indication of where to best to hail a cab.

The data file can be found on the HDFS under /tmp/labdata/sparkdata/nyctaxisub.csv. Remember, this is only a subset of the file that you used a previous exercise. If you ran this exercise on the full dataset, it would take a long time as we are only running on a test environment with limited resources.

__1. Start up the Spark shell.

__2. Import the needed packages for K-Means algorithm and Vector packages:

```
import org.apache.spark.mllib.clustering.KMeans

import org.apache.spark.mllib.linalg.Vectors
```

```
scala> import org.apache.spark.mllib.clustering.KMeans
import org.apache.spark.mllib.clustering.KMeans

scala> import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.linalg.Vectors
```

__3. Create an RDD from the HDFS data in '/tmp/labdata/sparkdata/nyctaxisub.csv'

```
val taxiFile = sc.textFile("/tmp/labdata/sparkdata/nyctaxisub.csv")
```

```
scala> val taxiFile = sc.textFile("/labdata/sparkdata/nyctaxisub/nyctaxisub.csv"
)
taxiFile: org.apache.spark.rdd.RDD[String] = /labdata/sparkdata/nyctaxisub/nycta
xisub.csv MappedRDD[1] at textFile at <console>:14
```

__4. Determine the number of rows in taxiFile.

```
taxiFile.count()
```

```
scala> taxiFile.count()
res0: Long = 250000
```

__5.   Cleanse the data.

```
val
taxiData=taxiFile.filter(_.contains("2013")).filter(_.split(",")(3)!=""
).filter(_.split(",")(4)!="")
```

The first filter limits the rows to those that occurred in the year 2013. This will also remove any header in the file. The third and fourth columns contain the drop off latitude and longitude. The transformation will throw exceptions if these values are empty.

__6.   Do another count to see what was removed.

```
taxiFile.count()
```

In this case, if we had used the full set of data, it would have been filtered out.

__7.   To fence the area roughly to New York City, copy in this command:

```
val taxiFence=taxiData.filter(_.split(",")(3).toDouble>40.70).

filter(_.split(",")(3).toDouble<40.86).

filter(_.split(",")(4).toDouble>(-74.02)).

filter(_.split(",")(4).toDouble<(-73.93))
```

__8.   Determine how many are left in taxiFence:

```
taxiFence.count()
```

```
scala> taxiFence.count()
res2: Long = 206646
```

Approximately, 43,354 rows were dropped since these drop-off points are outside of New York City.

__9.   Create Vectors with the latitudes and longitudes that will be used as input to the K-Means algorithm.

```
val
taxi=taxiFence.map{line=>Vectors.dense(line.split(',').slice(3,5).map(_
.toDouble))}
```

__10.   Run the K-Means algorithm. To make sure it works properly, copy and paste one line at a time.

```
val iterationCount=10
```

```
val clusterCount=3
```

```
val model=KMeans.train(taxi,clusterCount,iterationCount)

val clusterCenters=model.clusterCenters.map(_.toArray)

val cost=model.computeCost(taxi)

clusterCenters.foreach(lines=>println(lines(0),lines(1)))
```

```
scala> val iterationCount=10
iterationCount: Int = 10

scala> val clusterCount=3
clusterCount: Int = 3

scala> val model=KMeans.train(taxi,clusterCount,iterationCount)
model: org.apache.spark.mllib.clustering.KMeansModel = org.apache.spark.mllib.cl
ustering.KMeansModel@2390a585

scala> val clusterCenters=model.clusterCenters.map(_.toArray)
clusterCenters: Array[Array[Double]] = Array(Array(40.72487692697609, -73.995853
48942547), Array(40.78714676115601, -73.95704954430427), Array(40.75705853798479
, -73.98081347887664))

scala> val cost=model.computeCost(taxi)
cost: Double = 63.23322592304786

scala>

scala> clusterCenters.foreach(lines=>println(lines(0),lines(1)))
(40.72487692697609,-73.99585348942547)
(40.78714676115601,-73.95704954430427)
(40.75705853798479,-73.98081347887664)
```

Again, this may take about a minute on a single node VMware image or about 10 minutes if the larger data set is used. Not surprisingly, the second point is between the Theater District and Grand Central. The third point is in The Village, NYU, Soho and Little Italy area. The first point is the Upper East Side, presumably where people are more likely to take cabs than subways.

1.3   Creating a Spark application using Spark Streaming

This section focuses on Spark Streams, an easy to build, scalable, stateful (e.g. sliding windows) stream processing library. Streaming jobs are written the same way Spark batch jobs are coded and support Java, Scala and Python. In this exercise, taxi trip data will be streamed using a socket connection and then analyzed to provide a summary of number of passengers by taxi vendor. This will be implemented in the Spark shell using Scala.

There are two files under /home/virtuser/labdata/streams. The first one is the nyctaxi100.csv which will serve as the source of the stream. The other file is a python file, taxistreams.py, which will feed the csv file through a socket connection to simulate a stream.

Once started, the program will bind and listen to the localhost socket 7777. When a connection is made, it will read 'nyctaxi100.csv' and send across the socket. The sleep is set such that one line will be sent every 0.5 seconds, or 2 rows a second. This was intentionally set to a high value to make it easier to view the data during execution.

__1.   Open a new terminal.

__2.   Create a new folder, PythonStreams under /home/virtuser.

mkdir /home/virtuser/PythonStreams

__3.   Copy the taxistream.py and the nyctaxi100.csv file.

cp /home/virtuser/labfiles/streams/nyctaxi100.csv /home/virtuser/PythonStreams

cp /home/virtuser/labfiles/streams/taxistreams.py /home/virtuser/PythonStreams

__4.   Update the contents of the taxistreams.py file to reflect the path of the nyctaxi100.csv. The file current looks for the nyctaxi100.csv file under: /home/biadmin/PythonStreams/. Change it to /home/virtuser/PythonStreams/

__5.   Change into the PythonStreams folder

__6.   To invoke the standalone Python program, issue the following command:

python taxistreams.py

The program has been started and is awaiting Spark Streams to connect and receive the data.

__7.   Start a new terminal. The existing one is now waiting for the Spark Streams.

__8.   Connect to the spark-shell

$SPARK_HOME/bin/spark-shell

__9.   Turn off logging for this shell so that you can see the output of the application:

```
import org.apache.log4j.Logger

import org.apache.log4j.Level

Logger.getLogger("org").setLevel(Level.OFF)

Logger.getLogger("akka").setLevel(Level.OFF)
```

__10.   Import the required libraries. Copy and paste this into the shell.

```
import org.apache.spark._

import org.apache.spark.streaming._

import org.apache.spark.streaming.StreamingContext._
```

```
scala> import org.apache.spark._
import org.apache.spark._

scala> import org.apache.spark.streaming._
import org.apache.spark.streaming._

scala> import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.streaming.StreamingContext._
```

__11.   Create the StreamingContext by using the existing SparkContext (sc). It will be using a 1 second window, which means the stream is divided to 1 second batches and each batch becomes a RDD. This is intentional to make it easier to read the data during execution.

```
val ssc = new StreamingContext(sc,Seconds(1))
```

__12.   Create the socket stream that connects to the localhost socket 7777. This matches the port that the Python script is listening. Each stream will be a lines RDD.

```
val lines = ssc.socketTextStream("localhost",7777)
```

__13.   Next, put in the business logic to split up the lines on each comma and mapping pass(15), which is the vendor, and pass(7), which is the passenger count. Then this is reduced by key resulting in a summary of number of passengers by vendor.

```
val pass = lines.map(_.split(",")).

map(pass=>(pass(15),pass(7).toInt)).

reduceByKey(_+_)
```

__14.   Print out to the console:

```
pass.print()
```

```
scala> val ssc = new StreamingContext(sc,Seconds(1))
ssc: org.apache.spark.streaming.StreamingContext = org.apache.spark.streaming.St
reamingContext@5f6f410b

scala> val lines = ssc.socketTextStream("localhost",7777)
lines: org.apache.spark.streaming.dstream.ReceiverInputDStream[String] = org.apa
che.spark.streaming.dstream.SocketInputDStream@5b21541e

scala> val pass = lines.map(_.split(",")).
     | map(pass=>(pass(15),pass(7).toInt)).
     | reduceByKey(_+_)
pass: org.apache.spark.streaming.dstream.DStream[(String, Int)] = org.apache.spa
rk.streaming.dstream.ShuffledDStream@a5da2933

scala> pass.print()

scala>
```

__15.   The next two line starts the stream. Copy and paste both in at once.

```
ssc.start()

ssc.awaitTermination()
```

```
scala> val ssc = new StreamingContext(sc,Seconds(1))
ssc: org.apache.spark.streaming.StreamingContext = org.apache.spark.streaming.St
reamingContext@9c76666c

scala> val lines = ssc.socketTextStream("localhost",7777)
lines: org.apache.spark.streaming.dstream.ReceiverInputDStream[String] = org.apa
che.spark.streaming.dstream.SocketInputDStream@165a5220

scala> val pass = lines.map(_.split(",")).
     | map(pass=>(pass(15),pass(7).toInt)).
     | reduceByKey(_+_)
pass: org.apache.spark.streaming.dstream.DStream[(String, Int)] = org.apache.spa
rk.streaming.dstream.ShuffledDStream@9baf167c

scala> pass.print()

scala> ssc.start()

scala> ssc.awaitTermination()
```

__16.   It will take a few cycles for the connection to be recognized, and then the data is sent. In this
         case, 2 rows per second of taxi trip data is receive in a 1 second window.

```
scala> ssc.awaitTermination()
-------------------------------------------
Time: 1425676921000 ms
-------------------------------------------


-------------------------------------------
Time: 1425676922000 ms
-------------------------------------------


-------------------------------------------
Time: 1425676923000 ms
-------------------------------------------


-------------------------------------------
Time: 1425676924000 ms
-------------------------------------------


-------------------------------------------
Time: 1425676925000 ms
-------------------------------------------
("CMT",1)
("VTS",2)

-------------------------------------------
Time: 1425676926000 ms
-------------------------------------------
("CMT",5)
("VTS",5)

-------------------------------------------
Time: 1425676927000 ms
-------------------------------------------
("VTS",3)


-------------------------------------------
Time: 1425676928000 ms
-------------------------------------------
("CMT",1)
("VTS",3)
```

__17.   In the Python terminal, the contents of the file are printed as they are streamed.

```
A536F07CA7617","23A8ED0AAA1936A28C652B80903B42FB",2,"2013-01-11 10:01:00",+4.074
87490000000E+001,-7.39966050000000E+001,1,,+4.38000000000000E+000,1260,"VTS"

"29b3f4a30dea6688d4c289c9675d2b2e","1-e5d974cc18ee010bdadc938f429ea4b0","2013-01
-11 12:35:00",+4.08235400000000E+001,-7.39494090000000E+001,"57749B2DA60E2F46512
AFAAFD8C116BC","42BC02EC8FC9719B5B8075C3029B9EE9",1,"2013-01-11 12:22:00",+4.084
21130000000E+001,-7.39392850000000E+001,1,,+2.41000000000000E+000,780,"CMT"

"2a80cfaa425dcec0861e02ae4463fd7f","1-0874f169553c33569e767caf2b3214a3","2013-01
-11 20:28:00",+4.07467160000000E+001,-7.39899370000000E+001,"0DF63DB5EFE70702BFD
D877770B21604","EB49CE1B3661EF6100CF9EA1B860932E",3,"2013-01-11 20:13:00",+4.072
85690000000E+001,-7.39755940000000E+001,1,,+1.92000000000000E+000,900,"VTS"

"0ba1d5e3cc0d075c6513a0a1a2d6b221","1-4c86f2988613a685795e8bd753390d29","2013-01
-11 12:51:00",+4.07886810000000E+001,-7.39467320000000E+001,"5F18C262C656CCDE401
E240C849DAE61","DDE6F0B0832FA5CCE8491924E360FB45",1,"2013-01-11 12:36:00",+4.074
50220000000E+001,-7.39784390000000E+001,1,,+3.56000000000000E+000,900,"VTS"

"2a80cfaa425dcec0861e02ae4465114a","1-864e1450f7d337140c245f85ef6972c6","2013-01
-11 07:56:00",+4.07648770000000E+001,-7.39683230000000E+001,"518F5695F2D22A0E231
6C74649787AC5","67D7E407C25038B0CDDE600BD2F5244F",2,"2013-01-11 07:24:00",+4.083
01930000000E+001,-7.39278640000000E+001,1,,+3.80000000000000E+000,1920,"CMT"
```

__18. Use CTRL+C to get out of each terminal window to stop the programs.

This is just a simple example showing how you can take streaming data into Spark and do some type of processing on it. In the case here, the taxi and the number of passengers was extracted from the data stream.

1.4 Creating a Spark application using GraphX

__1. For this exercise, you will need to copy users.txt and follwers.txt from the local image to the HDFS under your own user directory. Issue these commands:

```
hadoop fs -copyFromLocal /home/virtuser/labfiles/users.txt /tmp

hadoop fs -copyFromLocal /home/virtuser/labfiles/followers.txt /tmp
```

__2. Users.txt is a set of users and followers is the relationship between the users. Take a look at the contents of these two files.

```
hadoop fs -cat /tmp/users.txt

hadoop fs -cat /tmp/followers.txt
```

```
[biadmin@cluster-743-1423492039-master data]$ hadoop fs -cat users.txt
1,BarackObama,Barack Obama
2,ladygaga,Goddess of Love
3,jeresig,John Resig
4,justinbieber,Justin Bieber
6,matei_zaharia,Matei Zaharia
7,odersky,Martin Odersky
8,anonsys
[biadmin@cluster-743-1423492039-master data]$ hadoop fs -cat followers.txt
2 1
4 1
1 2
6 3
7 3
7 6
6 7
3 7
[biadmin@cluster-743-1423492039-master data]$
```

__3.   Start up the spark-shell: $SPARK_HOME/bin/spark-shell

__4.   Import the GraphX package:

```
import org.apache.spark.graphx._
```

```
scala> import org.apache.spark.graphx._
import org.apache.spark.graphx._
```

__5.   Create the users RDD and parse into tuples of user id and attribute list:

```
val users = (sc.textFile("/tmp/users.txt").map(line =>
line.split(",")).map(parts => (parts.head.toLong, parts.tail)))
```

__6.   Parse the edge data, which is already in userId -> userId format

```
val followerGraph = GraphLoader.edgeListFile(sc, "/tmp/followers.txt")
```

__7.   Attach the user attributes

```
val graph = followerGraph.outerJoinVertices(users) {

  case (uid, deg, Some(attrList)) => attrList

  case (uid, deg, None) => Array.empty[String]

}
```

__8.   Restrict the graph to users with usernames and names:

```
val subgraph = graph.subgraph(vpred = (vid, attr) => attr.size == 2)
```

__9.   Compute the PageRank

```
val pagerankGraph = subgraph.pageRank(0.001)
```

__10. Get the attributes of the top pagerank users

```
val userInfoWithPageRank =
subgraph.outerJoinVertices(pagerankGraph.vertices) {

  case (uid, attrList, Some(pr)) => (pr, attrList.toList)

  case (uid, attrList, None) => (0.0, attrList.toList)

}
```

__11. Print the line out:

```
println(userInfoWithPageRank.vertices.top(5)(Ordering.by(_._2._1)).mkSt
ring("\n"))
```

```
scala> println(userInfoWithPageRank.vertices.top(5)(Ordering.by(_._2._1)).mkStri
ng("\n"))
(1,(1.453834747463902,List(BarackObama, Barack Obama)))
(2,(1.3857595353443166,List(ladygaga, Goddess of Love)))
(7,(1.2892158818481694,List(odersky, Martin Odersky)))
(3,(0.9936187772892124,List(jeresig, John Resig)))
(6,(0.697916749785472,List(matei_zaharia, Matei Zaharia)))

scala>
```

Summary

Having completed this exercise, you should have some familiarity with using the Spark libraries. IN particular, you use Spark SQL to effectively query data inside of Spark. You used Spark Streaming to process incoming streams of batch data. You used Spark's MLlib to compute the K-Means algorithm to find the best place to hail a cab. Finally, you used Spark's GraphX library to perform and parallel graph calculations on a dataset to find the attributes of the top users.

NOTES

NOTES

IBM Software

IBM Software