Spark Fundamentals

Working with RDD operations

Contents

Resilient Distributed Datasets (RDD)

The primary abstraction of Spark is RDD. Remember, there are two types of RDD operations: transformations and actions. Transformations return a pointer to the new RDD. Actions return the value of the operation. RDD transformations are lazy evaluations. This means nothing is processed until an action occurs. Each transformation updates the direct acyclic graph (DAG), which is only executed after an action is called. Due to this inherent behavior, Spark's applications are fault tolerant. The DAG can be used to rebuild the data within a node.

In this lab exercise, you will work with various RDD operations and learn some of the APIs from Scala and Python.

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

- Create a RDD from an external dataset
- View the Direct Acyclic Graph (DAG) of an RDD
- Work with various RDD operations including shared variables and key-value pairs

Allow 60 minutes to complete this section of lab.

1.1 Uploading files to the HDFS

__1. Open up a docker terminal. If you closed the terminal from the previous exercise, run the commands to start up the same bdu_spark container:

docker start bdu_spark

docker attach bdu_spark

Note: If the docker terminal doesn't return back with a prompt, just hit the Enter key and the prompt should appear.

__2. You'll need to start up the Hadoop service if you started a new terminal.

/etc/bootstrap.sh

Note: The Hadoop name node may be in safe mode. Exit the safe mode by entering in:

hdfs dfsadmin -safemode leave

__3. Copy a log file to the HDFS. You can copy the log file from the /opt/ibm/labfiles/ into the HDFS or you can upload any log file that you like for this exercise. Type in this command to load and rename the log file name to something shorter such as sparkLog.out

```
hdfs dfs –put /opt/ibm/labfiles/<log filename> input/tmp/sparkLog.out
```

__4. You should have the README.md file already in your image if you continued with the bdu-spark container. If not, use this command to copy it in.

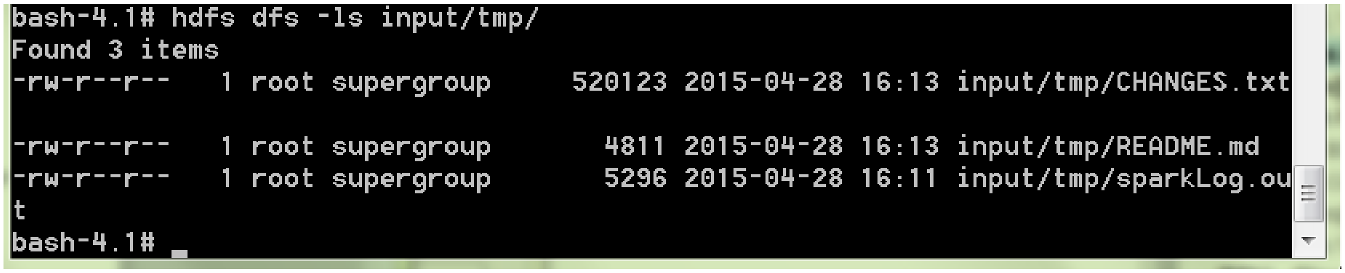
hdfs dfs -put /opt/ibm/labfiles/README.md input/tmp

__5. Copy the CHANGES.txt file located from the /opt/ibm/labfiles directory to the HDFS as well, under "input/tmp"

hdfs dfs -put /opt/ibm/labfiles/CHANGES.txt input/tmp

__6. Verify that the files were copied over by doing a listing of the HDFS

```
hdfs dfs -ls input/tmp/
```



```
bash-4.1# hdfs dfs -ls input/tmp/
Found 3 items
-rw-r--r--   1 root supergroup     520123 2015-04-28 16:13 input/tmp/CHANGES.txt

-rw-r--r--   1 root supergroup       4811 2015-04-28 16:13 input/tmp/README.md
-rw-r--r--   1 root supergroup       5296 2015-04-28 16:11 input/tmp/sparkLog.ou
t
bash-4.1# _
```

1.2 RDD operations using Scala

1.2.1 Analyzing a log file

__1. Start the Scala Spark shell:

```
$SPARK_HOME/bin/spark-shell
```

__2. Create a RDD by loading in that log file:

```
val logFile = sc.textFile("input/tmp/sparkLog.out")
```

__3. Filter out the lines that contains INFO (or ERROR, if the particular log has it)

```
val info = logFile.filter(line => line.contains("INFO"))
```

__4. Count the lines:

```
info.count()
```

__5. Count the lines with Spark in it by combining transformation and action.

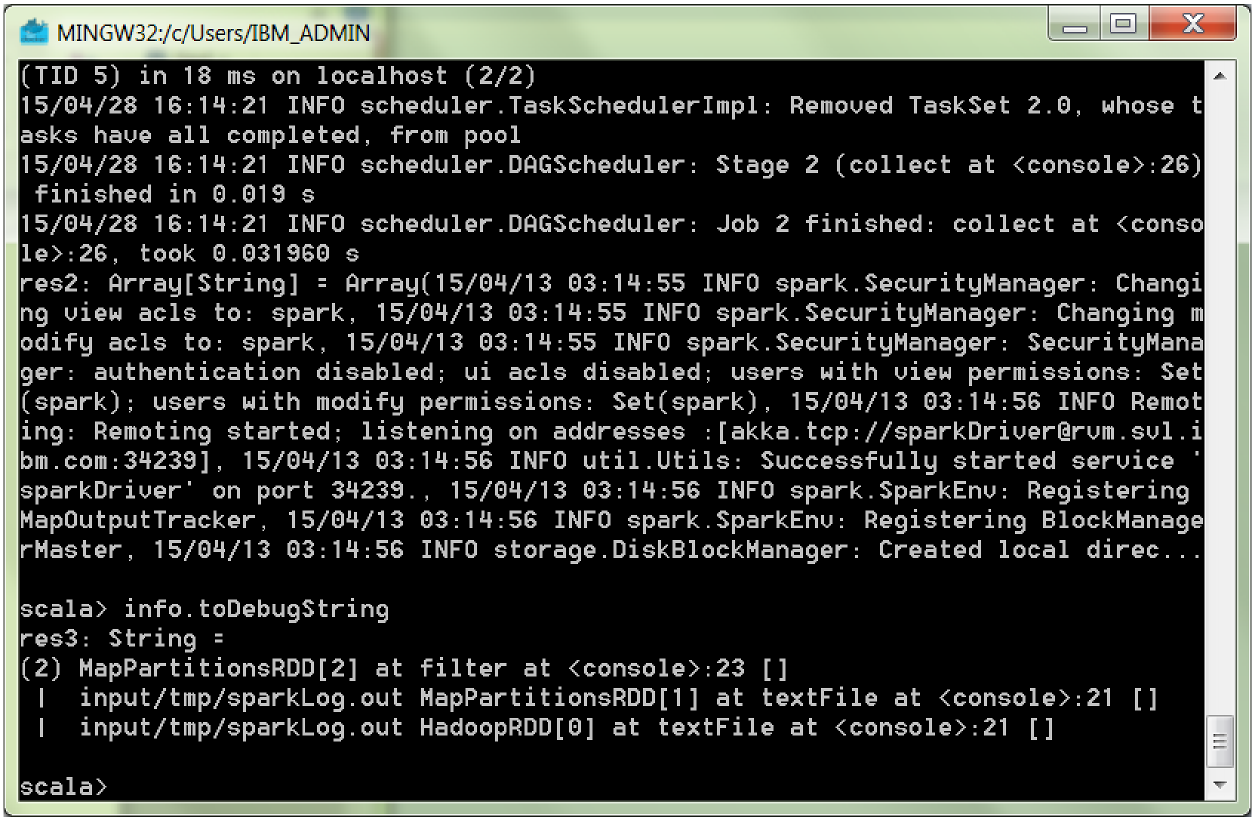
```
info.filter(line => line.contains("spark")).count()
```

__6. Fetch those lines as an array of Strings

```
info.filter(line => line.contains("spark")).collect()
```

__7. Remember that we went over the DAG. It is what provides the fault tolerance in Spark. Nodes can re-compute its state by borrowing the DAG from a neighboring node. You can view the graph of an RDD using the toDebugString command.

```
info.toDebugString
```



```
(TID 5) in 18 ms on localhost (2/2)
15/04/28 16:14:21 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 2.0, whose t
asks have all completed, from pool
15/04/28 16:14:21 INFO scheduler.DAGScheduler: Stage 2 (collect at <console>:26)
 finished in 0.019 s
15/04/28 16:14:21 INFO scheduler.DAGScheduler: Job 2 finished: collect at <conso
le>:26, took 0.031960 s
res2: Array[String] = Array(15/04/13 03:14:55 INFO spark.SecurityManager: Changi
ng view acls to: spark, 15/04/13 03:14:55 INFO spark.SecurityManager: Changing m
odify acls to: spark, 15/04/13 03:14:55 INFO spark.SecurityManager: SecurityMana
ger: authentication disabled; ui acls disabled; users with view permissions: Set
(spark); users with modify permissions: Set(spark), 15/04/13 03:14:56 INFO Remot
ing: Remoting started; listening on addresses :[akka.tcp://sparkDriver@rvm.svl.i
bm.com:34239], 15/04/13 03:14:56 INFO util.Utils: Successfully started service '
sparkDriver' on port 34239., 15/04/13 03:14:56 INFO spark.SparkEnv: Registering
MapOutputTracker, 15/04/13 03:14:56 INFO spark.SparkEnv: Registering BlockManage
rMaster, 15/04/13 03:14:56 INFO storage.DiskBlockManager: Created local direc...

scala> info.toDebugString
res3: String =
(2) MapPartitionsRDD[2] at filter at <console>:23 []
 |  input/tmp/sparkLog.out MapPartitionsRDD[1] at textFile at <console>:21 []
 |  input/tmp/sparkLog.out HadoopRDD[0] at textFile at <console>:21 []

scala>
```

Note: Occasionally, it may seem like the shell has not returned back the prompt. In that case, just hit the Enter key and the prompt will show back up.

1.2.2 Joining RDDs

__8. Next, you are going to create RDDs for the README and the CHANGES file.

```
val readmeFile = sc.textFile("input/tmp/README.md")

val changesFile = sc.textFile("input/tmp/CHANGES.txt")
```

__9. How many Spark keywords are in each file?

```
readmeFile.filter(line => line.contains("Spark")).count()

changesFile.filter(line => line.contains("Spark")).count()
```

__10. Now do a WordCount on each RDD so that the results are (K,V) pairs of (word,count)

```
val readmeCount = readmeFile.flatMap(line => line.split(" ")).map(word
=> (word, 1)).reduceByKey(_ + _)
```

```
val changesCount = changesFile.flatMap(line => line.split("
")).map(word => (word, 1)).reduceByKey(_ + _)
```

__11. To see the array for either of them, just call the collect function on it.

```
readmeCount.collect()
```

```
changesCount.collect()
```

__12. Now let's join these two RDDs together to get a collective set. The join function combines the two datasets (K,V) and (K,W) together and get (K, (V,W)). Let's join these two counts together

```
val joined = readmeCount.join(changesCount)
```

__13. Cache the joined dataset.

```
joined.cache()
```

__14. Print the value to the console

```
joined.collect.foreach(println)
```

__15. Let's combine the values together to get the total count. The operations in this command tells Spark to combine the go from (K,V) and (K,W) to (K, V+W). The ._ notation is a way to access the value on that particular index of the key value pair.

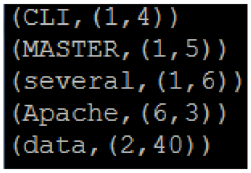
```
val joinedSum = joined.map(k => (k._1, (k._2)._1 + (k._2)._2))
```

```
joinedSum.collect()
```

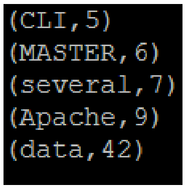
__16. To check if it is correct, print the first five elements from the joined and the joinedSum RDD

```
joined.take(5).foreach(println)
```

```
joinedSum.take(5).foreach(println)
```



```
(CLI,(1,4))
(MASTER,(1,5))
(several,(1,6))
(Apache,(6,3))
(data,(2,40))
```



```
(CLI,5)
(MASTER,6)
(several,7)
(Apache,9)
(data,42)
```

1.2.3 Shared variables

Broadcast variables are useful for when you have a large dataset that you want to use across all the worker nodes. Instead of having to send out the entire dataset, only the variable is sent out.

__17. In the same shell from the last section, create a broadcast variable. Type in

```
val broadcastVar = sc.broadcast(Array(1,2,3))
```

__18. To get the value, type in:

```
broadcastVar.value
```

Accumulators are variables that can only be added through an associative operation. It is used to implement counters and sum efficiently in parallel. Spark natively supports numeric type accumulators and standard mutable collections. Programmers can extend these for new types. Only the driver can read the values of the accumulators. The workers can only invoke it to increment the value.

__19. Create the accumulator variable. Type in:

```
val accum = sc.accumulator(0)
```

__20. Next parallelize an array of four integers and run it through a loop to add each integer value to the accumulator variable. Type in:

```
sc.parallelize(Array(1,2,3,4)).foreach(x => accum += x)
```

__21. To get the current value of the accumulator variable, type in:

```
accum.value
```

You should get a value of 10.

This command can only be invoked on the driver side. The worker nodes can only increment the accumulator.

1.2.4 Key-value pairs

You have already seen a bit about key-value pairs in the Joining RDD section. Here is a brief example of how to create a key-value pair and access its values. Remember that certain operations such as map and reduce only works on key-value pairs.

__22. Create a key-value pair of two characters. Type in:

```
val pair = ('a', 'b')
```

__23. To access the value of the first index, type in:

```
pair._1
```

__24. To access the value of the second index, type in:

```
pair._2
```

__25. Quit the Scala shell by typing in

CTRL +D

1.3 RDD operations using Python

This section goes over the same tasks using Python.

1.3.1 Analyzing a log file

__1. Start the Python Spark shell:

```
$SPARK_HOME/bin/pyspark
```

__2. Create a RDD by loading in that log file:

```
logFile = sc.textFile("input/tmp/sparkLog.out")
```

__3. Filter out the lines that contains INFO (or ERROR, if the particular log has it)

```
info = logFile.filter(lambda line: "INFO" in line)
```

__4. Count the lines:

```
info.count()
```

__5. Count the lines with Spark in it by combining transformation and action.

```
info.filter(lambda line: "spark" in line).count()
```

__6. Fetch those lines as an array of Strings

```
info.filter(lambda line: "spark" in line).collect()
```

__7. View the graph of an RDD using this command:

```
info.toDebugString
```

> **Note:** Occasionally, it may seem like the shell has not returned back the prompt. In that case, just hit the Enter key and the prompt will show back up.

1.3.2 Joining RDDs

__8. Next, you are going to create RDDs for the README and the CHANGES file.

```
readmeFile = sc.textFile("input/tmp/README.md")
```

```
changesFile = sc.textFile("input/tmp/CHANGES.txt")
```

__9. How many Spark keywords are in each file?

```
readmeFile.filter(lambda line: "Spark" in line).count()
```

```
changesFile.filter(lambda line: "Spark" in line).count()
```

__10. Now do a WordCount on each RDD so that the results are (K,V) pairs of (word,count)

```
readmeCount = readmeFile.flatMap(lambda line: line.split("
")).map(lambda word: (word, 1)).reduceByKey(lambda a, b: a + b)
```

```
changesCount = changesFile.flatMap(lambda line: line.split("
")).map(lambda word: (word, 1)).reduceByKey(lambda a, b: a + b)
```

__11. To see the array for either of them, just call the collect function on it.

```
readmeCount.collect()
```

```
changesCount.collect()
```

__12. The join function combines the two datasets (K,V) and (K,W) together and get (K, (V,W)). Let's join these two counts together

```
joined = readmeCount.join(changesCount)
```

__13. Cache the joined dataset.

```
joined.cache()
```

__14. Print the value to the console

```
joined.collect()
```

__15. Let's combine the values together to get the total count

```
joinedSum = joined.map(lambda k: (k[0], (k[1][0]+k[1][1])))
```

__16. To check if it is correct, print the first five elements from the joined and the joinedSum RDD

```
joined.take(5)
```

```
joinedSum.take(5)
```

1.3.3 Shared variables

Broadcast variables are useful for when you have a large dataset that you want to use across all the worker nodes. Instead of having to send out the entire dataset, only the variable is sent out.

__17. In the same shell from the last section, create a broadcast variable. Type in

```
broadcastVar = sc.broadcast(list(range(1,4)))
```

__18. To get the value, type in:

```
broadcastVar.value
```

Accumulators are variables that can only be added through an associative operation. It is used to implement counters and sum efficiently in parallel. Spark natively supports numeric type accumulators and standard mutable collections. Programmers can extend these for new types. Only the driver can read the values of the accumulators. The workers can only invoke it to increment the value.

__19. Create the accumulator variable. Type in:

```
accum = sc.accumulator(0)
```

__20. Next parallelize an array of four integers and run it through a loop to add each integer value to the accumulator variable. Type in:

```
rdd = sc.parallelize([1,2,3,4])

def f(x):

     global accum

     accum += x
```

__21. Next, iterate through each element of the rdd and apply the function f on it:

```
rdd.foreach(f)
```

__22. To get the current value of the accumulator variable, type in:

```
accum.value
```

You should get a value of 10.

This command can only be invoked on the driver side. The worker nodes can only increment the accumulator.

1.3.4 Key-value pairs

You have already seen a bit about key-value pairs in the Joining RDD section.

__23. Create a key-value pair of two characters. Type in:

```
pair = ('a', 'b')
```

__24. To access the value of the first index, type in:

```
pair[0]
```

__25. To access the value of the second index, type in:

```
pair[1]
```

__26. Quit the pyspark shell.

CTRL + D

1.4 Sample Application using Scala

In this section, you will be using a subset of a data for taxi trips that will determine the top 10 medallion numbers based on the number of trips. You will be doing this using the Spark shell with Scala.

__1. For this exercise, you will have to load additional taxi data into the HDFS. Under the input/tmp directory, create two additional directories under it. input/tmp/labdata/sparkdata/

```
hdfs dfs -mkdir input/tmp/labdata/

hdfs dfs -mkdir input/tmp/labdata/sparkdata/
```

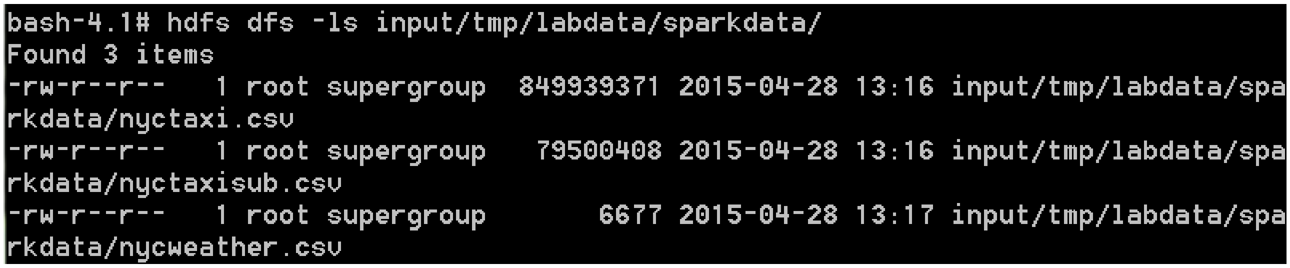
__2. Next, upload three csv files under sparkdata: nyctaxi.csv, nyctaxisub.csv, and nycweather.csv.

```
hdfs dfs -put /opt/ibm/labfiles/nyctaxi/nyctaxi.csv
input/tmp/labdata/sparkdata/

hdfs dfs -put /opt/ibm/labfiles/nyctaxisub/nyctaxisub.csv
input/tmp/labdata/sparkdata/

hdfs dfs -put /opt/ibm/labfiles/nycweather/nycweather.csv
input/tmp/labdata/sparkdata/
```

It is going to take a while to upload the nyctaxi.csv data. It is a fairly large file. For this lab exercise, you will only use the nyctaxi data. The others will be used at a later time.

__3. Do a listing of the directory to make sure all three files were uploaded:

```
hdfs dfs -ls input/tmp/labdata/sparkdata/
```



```
bash-4.1# hdfs dfs -ls input/tmp/labdata/sparkdata/
Found 3 items
-rw-r--r--   1 root supergroup  849939371 2015-04-28 13:16 input/tmp/labdata/spa
rkdata/nyctaxi.csv
-rw-r--r--   1 root supergroup   79500408 2015-04-28 13:16 input/tmp/labdata/spa
rkdata/nyctaxisub.csv
-rw-r--r--   1 root supergroup       6677 2015-04-28 13:17 input/tmp/labdata/spa
rkdata/nycweather.csv
```

__4. Start up the spark shell

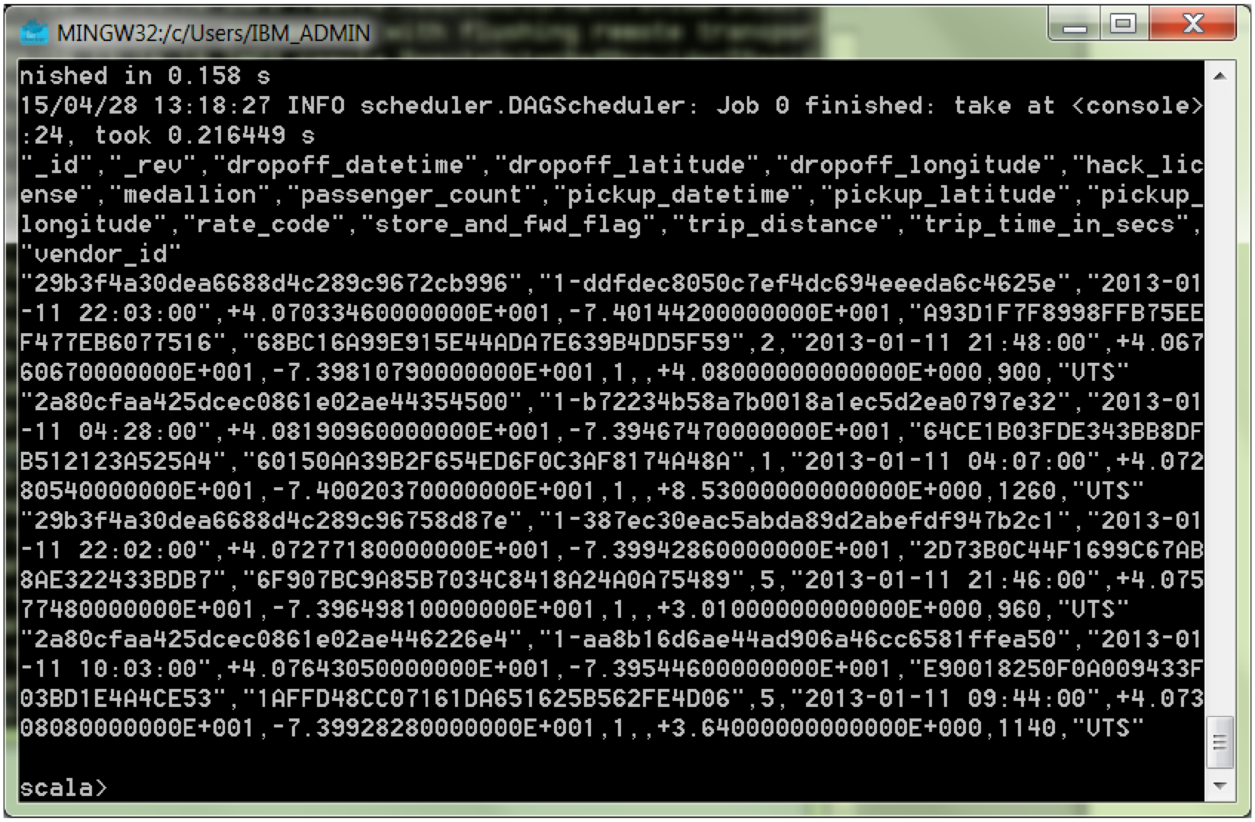
$SPARK_HOME/bin/spark-shell

__5. Create an RDD from the HDFS data in 'input/tmp/labdata/sparkdata/nyctaxi.csv'

```
val taxi = sc.textFile("input/tmp/labdata/sparkdata/nyctaxi.csv")
```

__6. To view the five rows of content, invoke the take function. Type in:

```
taxi.take(5).foreach(println)
```



```
nished in 0.158 s
15/04/28 13:18:27 INFO scheduler.DAGScheduler: Job 0 finished: take at <console>
:24, took 0.216449 s
"_id","_rev","dropoff_datetime","dropoff_latitude","dropoff_longitude","hack_lic
ense","medallion","passenger_count","pickup_datetime","pickup_latitude","pickup_
longitude","rate_code","store_and_fwd_flag","trip_distance","trip_time_in_secs",
"vendor_id"
"29b3f4a30dea6688d4c289c9672cb996","1-ddfdec8050c7ef4dc694eeeda6c4625e","2013-01
-11 22:03:00",+4.07033460000000E+001,-7.40144200000000E+001,"A93D1F7F8998FFB75EE
F477EB6077516","68BC16A99E915E44ADA7E639B4DD5F59",2,"2013-01-11 21:48:00",+4.067
60670000000E+001,-7.39810790000000E+001,1,,+4.08000000000000E+000,900,"VTS"
"2a80cfaa425dcec0861e02ae44354500","1-b72234b58a7b0018a1ec5d2ea0797e32","2013-01
-11 04:28:00",+4.08190960000000E+001,-7.39467470000000E+001,"64CE1B03FDE343BB8DF
B512123A525A4","60150AA39B2F654ED6F0C3AF8174A48A",1,"2013-01-11 04:07:00",+4.072
80540000000E+001,-7.40020370000000E+001,1,,+8.53000000000000E+000,1260,"VTS"
"29b3f4a30dea6688d4c289c96758d87e","1-387ec30eac5abda89d2abefdf947b2c1","2013-01
-11 22:02:00",+4.07277180000000E+001,-7.39942860000000E+001,"2D73B0C44F1699C67AB
8AE322433BDB7","6F907BC9A85B7034C8418A24A0A75489",5,"2013-01-11 21:46:00",+4.075
77480000000E+001,-7.39649810000000E+001,1,,+3.01000000000000E+000,960,"VTS"
"2a80cfaa425dcec0861e02ae446226e4","1-aa8b16d6ae44ad906a46cc6581ffea50","2013-01
-11 10:03:00",+4.07643050000000E+001,-7.39544600000000E+001,"E90018250F0A009433F
03BD1E4A4CE53","1AFFD48CC07161DA651625B562FE4D06",5,"2013-01-11 09:44:00",+4.073
08080000000E+001,-7.39928280000000E+001,1,,+3.64000000000000E+000,1140,"VTS"

scala>
```

Note that the first line is the headers. Normally, you would want to filter that out, but since it will not affect our results, we can leave it in.

__7. To parse out the values, including the medallion numbers, you need to first create a new RDD by splitting the lines of the RDD using the comma as the delimiter. Type in:

```
val taxiParse = taxi.map(line=>line.split(","))
```

__8. Now create the key-value pairs where the key is the medallion number and the value is 1. We use this model to later sum up all the keys to find out the number of trips a particular taxi took and in particular, will be able to see which taxi took the most trips. Map each of the medallions to the value of one. Type in:

```
val taxiMedKey = taxiParse.map(vals=>(vals(6), 1))
```

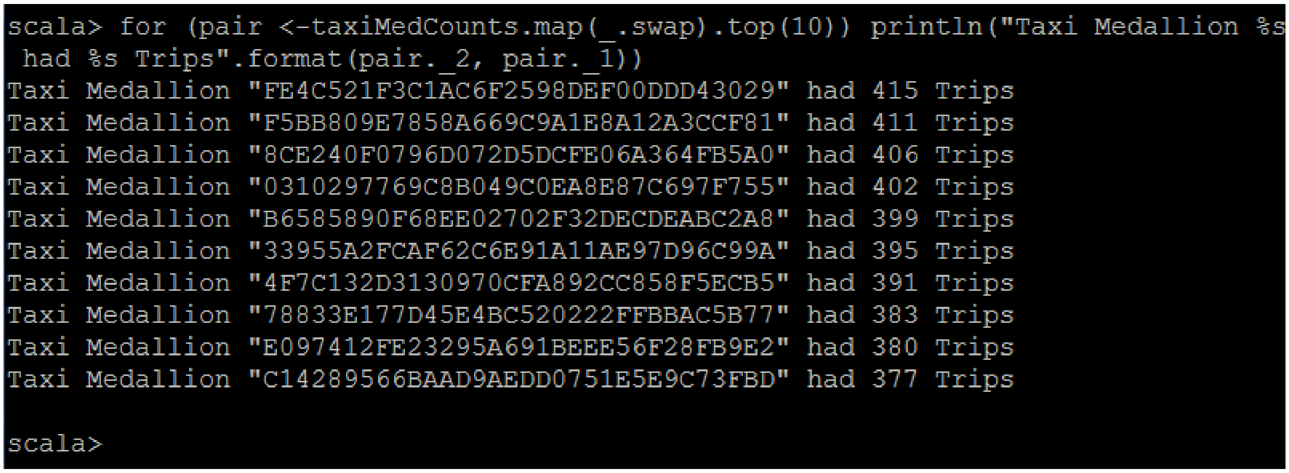
vals(6) corresponds to the column where the medallion key is located

__9. Next use the reduceByKey function to count the number of occurrence for each key.

```
val taxiMedCounts = taxiMedKey.reduceByKey((v1,v2)=>v1+v2)
```

__10. Finally, the values are swapped so they can be ordered in descending order and the results are presented correctly.

```
for (pair <-taxiMedCounts.map(_.swap).top(10)) println("Taxi Medallion
%s had %s Trips".format(pair._2, pair._1))
```



```
scala> for (pair <-taxiMedCounts.map(_.swap).top(10)) println("Taxi Medallion %s
 had %s Trips".format(pair._2, pair._1))
Taxi Medallion "FE4C521F3C1AC6F2598DEF00DDD43029" had 415 Trips
Taxi Medallion "F5BB809E7858A669C9A1E8A12A3CCF81" had 411 Trips
Taxi Medallion "8CE240F0796D072D5DCFE06A364FB5A0" had 406 Trips
Taxi Medallion "0310297769C8B049C0EA8E87C697F755" had 402 Trips
Taxi Medallion "B6585890F68EE02702F32DECDEABC2A8" had 399 Trips
Taxi Medallion "33955A2FCAF62C6E91A11AE97D96C99A" had 395 Trips
Taxi Medallion "4F7C132D3130970CFA892CC858F5ECB5" had 391 Trips
Taxi Medallion "78833E177D45E4BC520222FFBBAC5B77" had 383 Trips
Taxi Medallion "E097412FE23295A691BEEE56F28FB9E2" had 380 Trips
Taxi Medallion "C14289566BAAD9AEDD0751E5E9C73FBD" had 377 Trips

scala>
```

__11. While each step above was processed one line at a time, you can just as well process everything on one line:

```
val taxiMedCountsOneLine =
taxi.map(line=>line.split(',')).map(vals=>(vals(6),1)).reduceByKey(_ +
_)
```

__12. Run the same line as above to print the taxiMedCountsOneLine RDD.

```
for (pair <-taxiMedCountsOneLine.map(_.swap).top(10)) println("Taxi
Medallion %s had %s Trips".format(pair._2, pair._1))
```

__13. Let's cache the taxiMedCountsOneLine to see the difference caching makes. Run it with the logs set to INFO and you can see the output of the time it takes to execute each line. First, let's cache the RDD

```
taxiMedCountsOneLine.cache()
```

__14. Next, you have to invoke an action for it to actually cache the RDD. Note the time it takes here (either empirically using the INFO log or just notice the time it takes)

```
taxiMedCountsOneLine.count()
```

__15. Run it again to see the difference.

```
taxiMedCountsOneLine.count()
```

The bigger the dataset, the more noticeable the difference will be. In a sample file such as ours, the difference may be negligible.

Summary

Having completed this exercise, you should now be able perform various RDD operations, view the DAG of an RDD and also perform join operations to combine two datasets into one. You should be able to user shared variables such as broadcast and accumulators. Finally, you saw how to use Spark and Scala to determine the top ten taxis based on the numbers of trips from a sample dataset.

NOTES

NOTES

© Copyright IBM Corporation 2015.

The information contained in these materials is provided for informational purposes only, and is provided AS IS without warranty of any kind, express or implied. IBM shall not be responsible for any damages arising out of the use of, or otherwise related to, these materials. Nothing contained in these materials is intended to, nor shall have the effect of, creating any warranties or representations from IBM or its suppliers or licensors, or altering the terms and conditions of the applicable license agreement governing the use of IBM software. References in these materials to IBM products, programs, or services do not imply that they will be available in all countries in which IBM operates. This information is based on current IBM product plans and strategy, which are subject to change by IBM without notice. Product release dates and/or capabilities referenced in these materials may change at any time at IBM’s sole discretion based on market opportunities or other factors, and are not intended to be a commitment to future product or feature availability in any way.

IBM, the IBM logo and ibm.com are trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml.

Please Recycle