Spark Fundamentals - II

Joining RDDs using partitioning

Contents

Joining RDDs using partitioning

In this lab you'll use a bike trip dataset of 2 files. The data is in CSV format so you'll first parse the two files, then join the trips with the stations get the start and end stations details.

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

- o Join RDDs efficiently using partitioning
- o Use the FAIR Schedule to run concurrent jobs
- o Understand the lineage of RDDs, how input is partitioned, and the impact of partitioning when joining RDDs

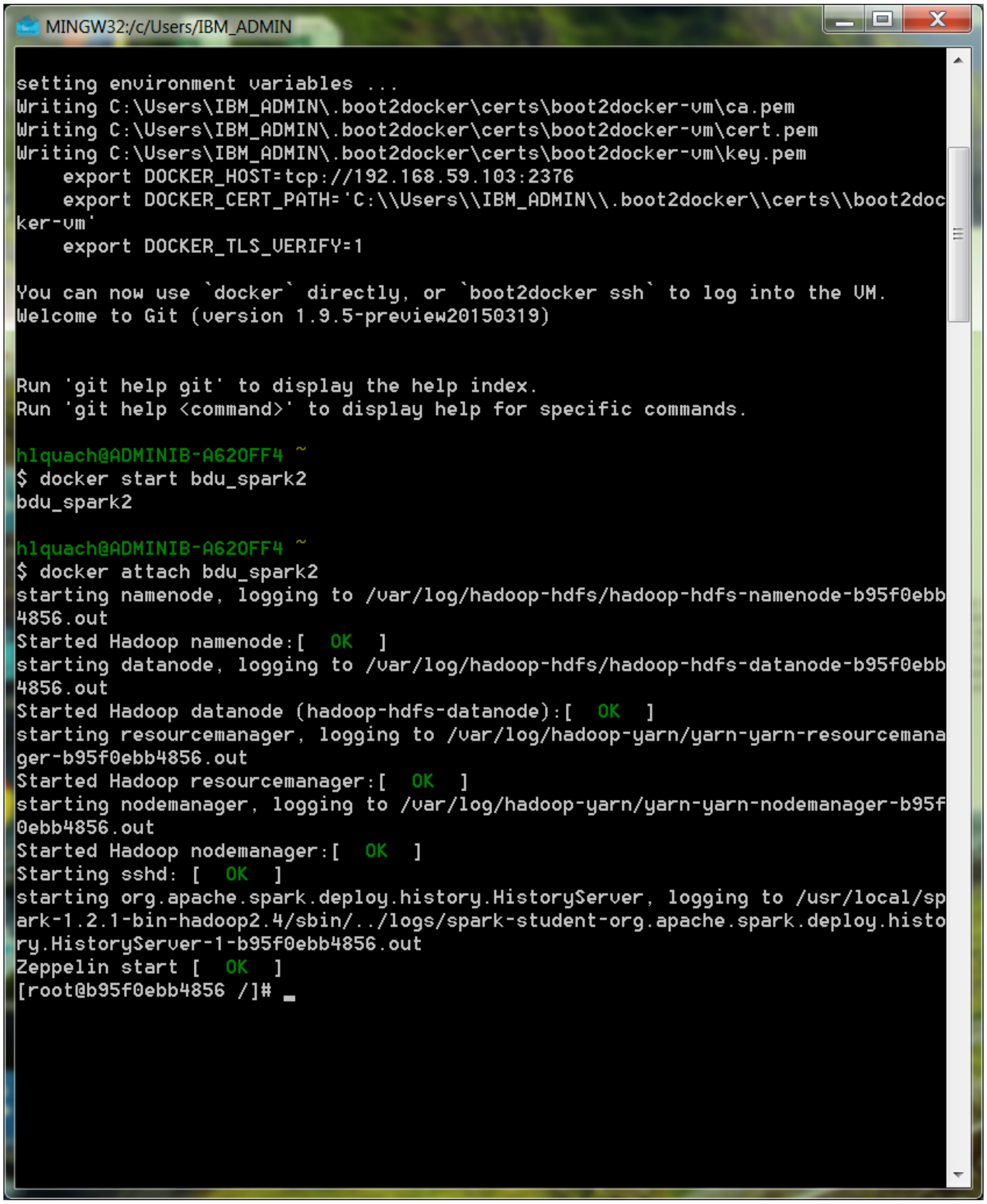
Allow 30 minutes to complete this section of lab.

1.1 Joining RDDs using partitioning

Make sure your docker and the Zeppelin is running. If not, get it started before continuing with this lab. Use the instructions here:

https://registry.hub.docker.com/u/bigdatauniversity/spark2/

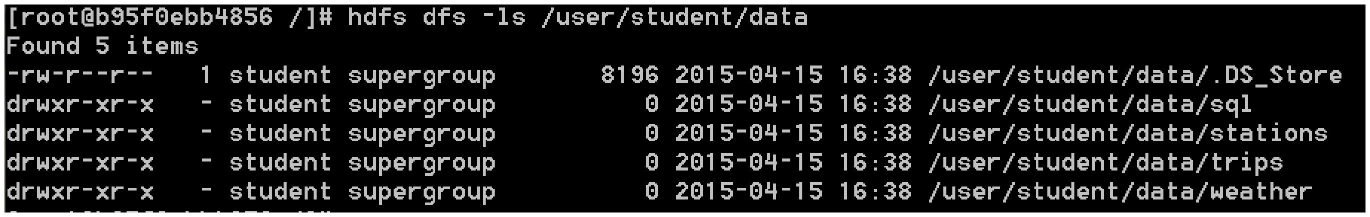
__1. Here I show how you would restart the container if you need to. Otherwise, continue from the previous instance of the boot2docker terminal from lab 1.



```
setting environment variables ...
Writing C:\Users\IBM_ADMIN\.boot2docker\certs\boot2docker-vm\ca.pem
Writing C:\Users\IBM_ADMIN\.boot2docker\certs\boot2docker-vm\cert.pem
Writing C:\Users\IBM_ADMIN\.boot2docker\certs\boot2docker-vm\key.pem
    export DOCKER_HOST=tcp://192.168.59.103:2376
    export DOCKER_CERT_PATH='C:\\Users\\IBM_ADMIN\\.boot2docker\\certs\\boot2doc
ker-vm'
    export DOCKER_TLS_VERIFY=1

You can now use `docker` directly, or `boot2docker ssh` to log into the VM.
Welcome to Git (version 1.9.5-preview20150319)


Run 'git help git' to display the help index.
Run 'git help <command>' to display help for specific commands.

hlquach@ADMINIB-A620FF4 ~
$ docker start bdu_spark2
bdu_spark2

hlquach@ADMINIB-A620FF4 ~
$ docker attach bdu_spark2
starting namenode, logging to /var/log/hadoop-hdfs/hadoop-hdfs-namenode-b95f0ebb
4856.out
Started Hadoop namenode:[  OK  ]
starting datanode, logging to /var/log/hadoop-hdfs/hadoop-hdfs-datanode-b95f0ebb
4856.out
Started Hadoop datanode (hadoop-hdfs-datanode):[  OK  ]
starting resourcemanager, logging to /var/log/hadoop-yarn/yarn-yarn-resourcemana
ger-b95f0ebb4856.out
Started Hadoop resourcemanager:[  OK  ]
starting nodemanager, logging to /var/log/hadoop-yarn/yarn-yarn-nodemanager-b95f
0ebb4856.out
Started Hadoop nodemanager:[  OK  ]
Starting sshd: [  OK  ]
starting org.apache.spark.deploy.history.HistoryServer, logging to /usr/local/sp
ark-1.2.1-bin-hadoop2.4/sbin/../logs/spark-student-org.apache.spark.deploy.histo
ry.HistoryServer-1-b95f0ebb4856.out
Zeppelin start [  OK  ]
[root@b95f0ebb4856 /]# _
```

__2. There are two datasets that you will be using for this lab. The data is from the San Francisco bay area bike share program. It is a public dataset that contains the trips and the bike stations over roughly a two years period. They are simple CSV files. The files for this lab have already been placed on the HDFS. Take a look at the two datasets using the boot2docker terminal, type in:

```
hdfs dfs -ls /user/student/data
```



```
[root@b95f0ebb4856 /]# hdfs dfs -ls /user/student/data
Found 5 items
-rw-r--r--   1 student supergroup       8196 2015-04-15 16:38 /user/student/data/.DS_Store
drwxr-xr-x   - student supergroup          0 2015-04-15 16:38 /user/student/data/sql
drwxr-xr-x   - student supergroup          0 2015-04-15 16:38 /user/student/data/stations
drwxr-xr-x   - student supergroup          0 2015-04-15 16:38 /user/student/data/trips
drwxr-xr-x   - student supergroup          0 2015-04-15 16:38 /user/student/data/weather
```

__3. The respective CSV files are located in **stations** and **trips.** You can review the files if you wish.

__4. Open up Lab 2 in the Zeppelin notebook in your web browser (http://192.168.59.103:8080). Click the **Notebook** dropdown and select **Lab 2**.

__5. The first task in lab 2 is to create two new RDDs by joining the trips with the stations by their start and end terminals. First you create the RDD of the trips. This has been done in the notebook for you:

```
val input1 = sc.textFile("data/trips/*")
val header1 = input1.first // to skip the header row
val trips = input1.filter(_ != header1).map(_.split(","))
```

First line grabs the text file from the HDFS. Second line extracts the header column. Third line contains two transformations strung together. The first transformation filters out the header column and then for each line, it splits up by the comma to get the individual cell / value to make up the trips RDD.

__6. Create the RDD for the stations. This has been done for you in the notebook:

```
val input2 = sc.textFile("data/stations/*")
val header2 = input2.first // to skip the header row
val stations = input2.filter(_ != header2).map(_.split(","))
```

__7. Create a new RDD to join the *trips* and *stations* RDD. Use the *keyBy* function to convert the original RDDs to a Pair RDD. Append this transformation to the end of the *stations* RDD: **keyBy(_(0).toInt)** The new *stations* RDD should look like this:

```
val stations = input2.filter(_ != header2).map(_.split(",")).keyBy(_(0).toInt)
```

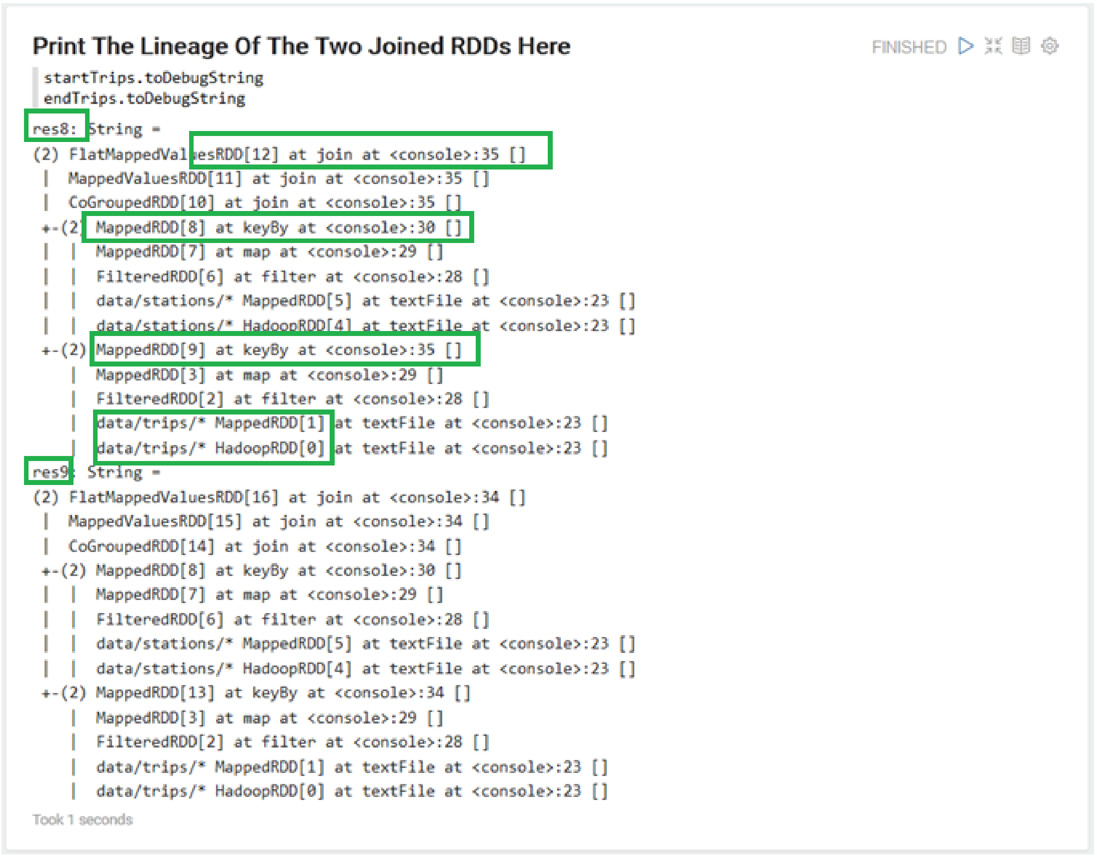
Now the stations RDD is keyed by the id field for the station, which is located at index 0.

__8. To create the RDD of the **startTrips** and the **endTrips,** join the stations RDD with each of the trips RDD. Remember to use the *keyBy* function on each of the original trips RDD. The index of the station id of each the start and end trip is 4 and 7, respectively.

```
val startTrips = stations.join(trips.keyBy(_(4).toInt))
val endTrips = stations.join(trips.keyBy(_(7).toInt))
```

__9. Run the panel to execute the Spark tasks. This will create the RDDs. Remember that because these are only transformation, no actual values are returned, rather just the pointers to them. Click on the **Play** button to execute the tasks.

__10. Examine the lineage of the RDDs by using the *toDebugString* function. The next task is to identify how many tasks will run and the stage boundaries. Print the linage of the two joined RDD (startTrips and endTrips) by typing in the code and running the panel in the notebook.

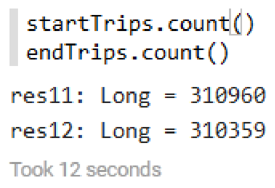


```
startTrips.toDebugString
endTrips.toDebugString
res8: String =
(2) FlatMappedValuesRDD[12] at join at <console>:35 []
 |  MappedValuesRDD[11] at join at <console>:35 []
 |  CoGroupedRDD[10] at join at <console>:35 []
 +-(2) MappedRDD[8] at keyBy at <console>:30 []
 |  |  MappedRDD[7] at map at <console>:29 []
 |  |  FilteredRDD[6] at filter at <console>:28 []
 |  |  data/stations/* MappedRDD[5] at textFile at <console>:23 []
 |  |  data/stations/* HadoopRDD[4] at textFile at <console>:23 []
 +-(2) MappedRDD[9] at keyBy at <console>:35 []
    |  MappedRDD[3] at map at <console>:29 []
    |  FilteredRDD[2] at filter at <console>:28 []
    |  data/trips/* MappedRDD[1] at textFile at <console>:23 []
    |  data/trips/* HadoopRDD[0] at textFile at <console>:23 []
res9: String =
(2) FlatMappedValuesRDD[16] at join at <console>:34 []
 |  MappedValuesRDD[15] at join at <console>:34 []
 |  CoGroupedRDD[14] at join at <console>:34 []
 +-(2) MappedRDD[8] at keyBy at <console>:30 []
 |  |  MappedRDD[7] at map at <console>:29 []
 |  |  FilteredRDD[6] at filter at <console>:28 []
 |  |  data/stations/* MappedRDD[5] at textFile at <console>:23 []
 |  |  data/stations/* HadoopRDD[4] at textFile at <console>:23 []
 +-(2) MappedRDD[13] at keyBy at <console>:34 []
    |  MappedRDD[3] at map at <console>:29 []
    |  FilteredRDD[2] at filter at <console>:28 []
    |  data/trips/* MappedRDD[1] at textFile at <console>:23 []
    |  data/trips/* HadoopRDD[0] at textFile at <console>:23 []
Took 1 seconds
```

res8 is the startTrips lineage. From there, you read bottom up and see that when you loaded the textFile, it creates a HadoopRDD wrapper then it maps the value from the file into the RDD. The next task then shows the keyBy operation. The same type of operations happens for both files, and if there are enough cores, they are done in parallel. Finally, the third stage is where the RDDs are joined.

res9 shows the endTrips lineage.

__11. Run the next task in the notebook to invoke an action on the joined RDDs.



```
startTrips.count()
endTrips.count()
res11: Long = 310960
res12: Long = 310359
Took 12 seconds
```

__12. How could we optimize this? Because we're keying all the RDDs by the station id and joining against the same station RDD, we should use a HashPartitioner on the stations RDD to allow for effective partitioning. To do so, use the partitionBy function along with the HashPartitioner to partition by the same number as the trips RDD.

```
partitionBy(new HashPartitioner(trips.partitions.size))
```

Append that transformation to the end of the stations RDD and run that panel.

__13. Finally, Zeppelin has been configured to use the Fair Scheduler. This will allow for running concurrent jobs. A way to test this would be to use the async actions using Scala futures. Run the panel to see that the tasks run successfully.

Summary

Having completed this exercise, you should now understand how and when to use effective partitioning for join operations. By partitioning on RDDs that are being joined multiple times, you reduce the amount of shuffling required for the join operation. You also saw that the Fair Scheduler allows for asynchronous or concurrent jobs.

NOTES

NOTES