Lab: Getting Started with Apache Spark

About This Lab

Objective:	To start up a Spark Shell application, and perform Hello World
File locations:	`/root/spark/data/selfishgiant.txt`
Successful outcome:	User will have started the shell and performed word count on the dataset
Before you begin	Get your AWS IP

Lab Steps

Perform the following steps:

1. Login into the AWS instance
    a. Everyone will be given an IP with their AWS instance started up. Navigate to the web GUI by opening a web browser and going to the location below:

```
<aws_ip>/guacamole
```

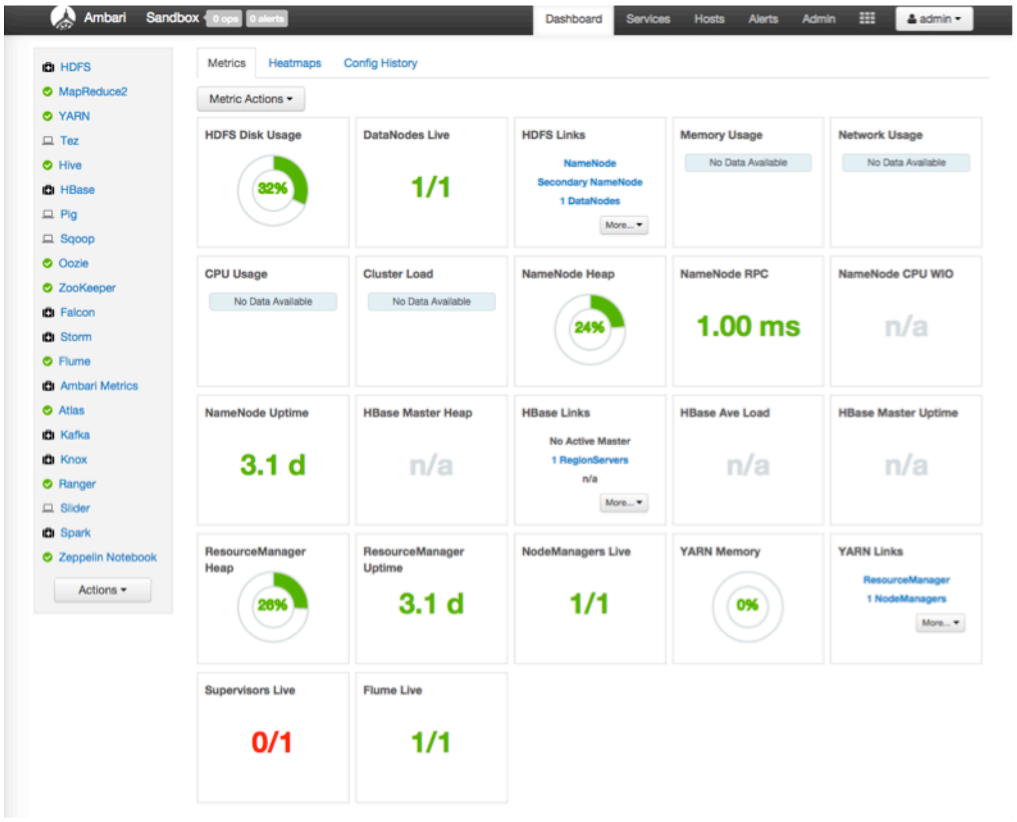
    b. Verify the cluster is running by going to the following url to log into Ambari:

```
<aws_ip>:8080
```

    Log into Ambari using the following credentials:

    Username: `admin`
    Password: `admin`

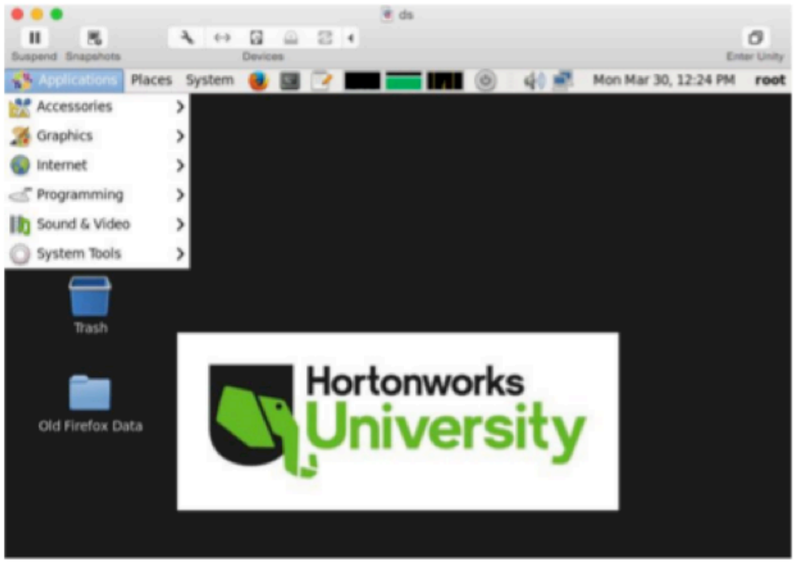
    c. You should now be logged into Ambari and can see the cluster information. Your screen should look something like this:



d. Now that we have verified you're able to login, and the cluster is setup and running, we are now ready to move on to the next step of the lab.

2. Starting up the Spark Shell

a. Open up a Terminal window, either by clicking on the Terminal icon in the top toolbar, or by the `Application->System Tools` pull-down:



b. In the opened Terminal window type the following to log into to the docker sandbox container. Every command should be run from inside the docker sandbox container:

```
# ssh sandbox
```

c. Start the `spark-shell` by typing the following

For Scala:

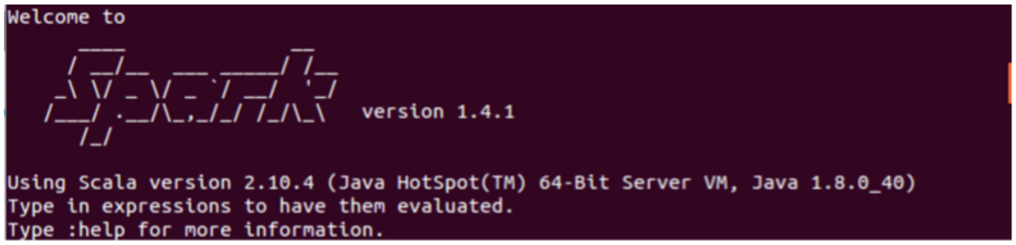
```
# spark-shell
```

For Python:

```
# pyspark
```

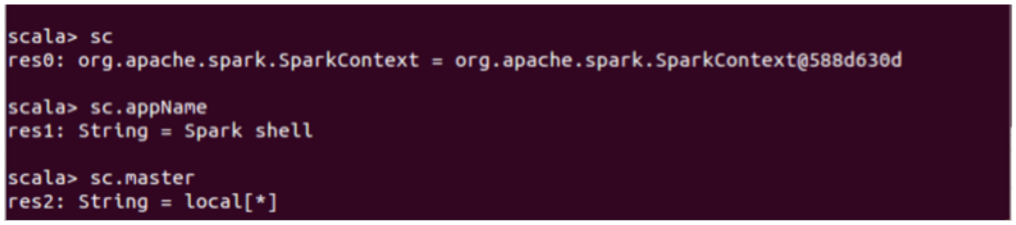
d. Take a look at the Spark context and some attributes

```
> sc
> sc.appName
> sc.master
```



```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.4.1
      /_/

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_40)
Type in expressions to have them evaluated.
Type :help for more information.
```



```
scala> sc
res0: org.apache.spark.SparkContext = org.apache.spark.SparkContext@588d630d

scala> sc.appName
res1: String = Spark shell

scala> sc.master
res2: String = local[*]
```

3. View the raw data for this lab

a. In a new terminal window, change directories to the data directory:

```
# cd ~/spark/data
```

b. View the data file "`selfishgiant.txt`"

```
# tail selfishgiant.txt
```

c. This file contains the short story "Selfish Giant."

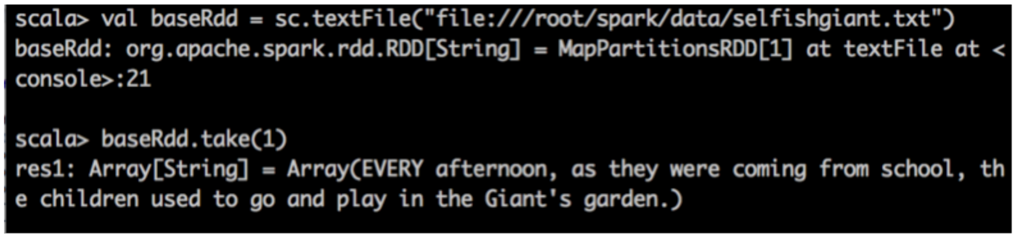
4. From the Spark Shell, write the logic for counting all the words

a. Create an RDD from the file we just viewed above:

```
>>> val baseRdd=sc.textFile("file:///root/spark/data/selfishgiant.txt")
```

b. Verify that you have created and RDD from the correct file using `take(1)`:

```
>>> baseRdd.take(1)
```

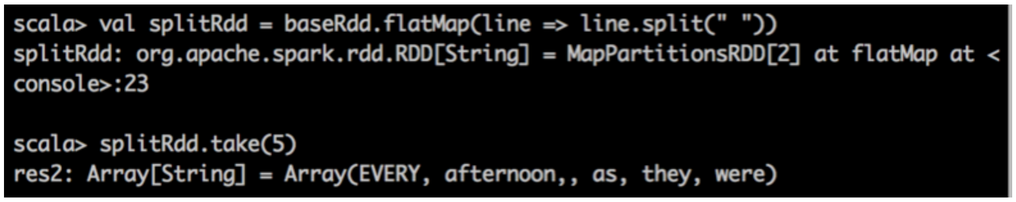


```
scala> val baseRdd = sc.textFile("file:///root/spark/data/selfishgiant.txt")
baseRdd: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[1] at textFile at <
console>:21

scala> baseRdd.take(1)
res1: Array[String] = Array(EVERY afternoon, as they were coming from school, th
e children used to go and play in the Giant's garden.)
```

c. Each element is currently a string, transform the string into arrays and examine the output

```
>>> val splitRdd = baseRdd.flatMap(line => line.split(" "))
>>> splitRdd.take(5)
```



```
scala> val splitRdd = baseRdd.flatMap(line => line.split(" "))
splitRdd: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[2] at flatMap at <
console>:23

scala> splitRdd.take(5)
res2: Array[String] = Array(EVERY, afternoon,, as, they, were)
```

d. Map each element into a key/value pair, with the key being the word and the value being 1. Examine the output.

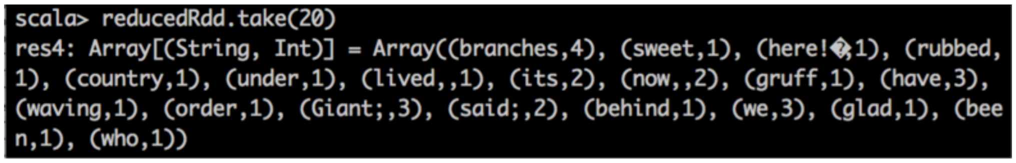
```
>>> val mappedRdd = splitRdd.map(line=> (line,1))
>>> mappedRdd.take(5)
```

e. Reduce the key/value pairs to get the count of each word:

```
>>> val reducedRdd = mappedRdd.reduceByKey((a,b) => a+b)
```

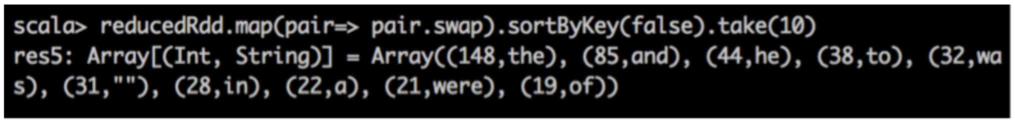
f. Run an action to get output:

```
>>> reducedRdd.take(20)
>>> reducedRdd.collect()
```



```
scala> reducedRdd.take(20)
res4: Array[(String, Int)] = Array((branches,4), (sweet,1), (here!�,1), (rubbed,
1), (country,1), (under,1), (lived,,1), (its,2), (now,,2), (gruff,1), (have,3),
(waving,1), (order,1), (Giant;,3), (said;,2), (behind,1), (we,3), (glad,1), (bee
n,1), (who,1))
```

5. **CHALLENGE:** Find the ten most prominent words.



```
scala> reducedRdd.map(pair=> pair.swap).sortByKey(false).take(10)
res5: Array[(Int, String)] = Array((148,the), (85,and), (44,he), (38,to), (32,wa
s), (31,""), (28,in), (22,a), (21,were), (19,of))
```

RESULT

You should now know how to start the spark shell and perform some basic RDD transformations and actions.

Lab: Using HDFS Commands

About This Lab

Objective:	To become familiar with how files are added to and removed from HDFS, and how to view files in HDFS
File locations:	`/root/spark/data/`
Successful outcome:	You will have added and deleted several files and folders in HDFS
Before you begin	You should be logged in to your AWS instance

Lab Steps

Perform the following steps:

1. View the `hdfs dfs` command
   a. With your AWS instance, open a Terminal window if you do not have one open already.
   b. From the command line, enter the following command to view the usage:

```
# hdfs dfs
```

   c. Notice the usage contains options for performing file system tasks in HDFS, like copying files from a local folder into HDFS, retrieving a file from HDFS, copying an moving files around, and making and removing directories. In this lab, you will perform these commands and many others, to help you become comfortable with working with the HDFS.

2. Create a directory in HDFS

Enter the following `-ls` command to view the contents of the user's root directory in HDFS, which is `/user/root`:

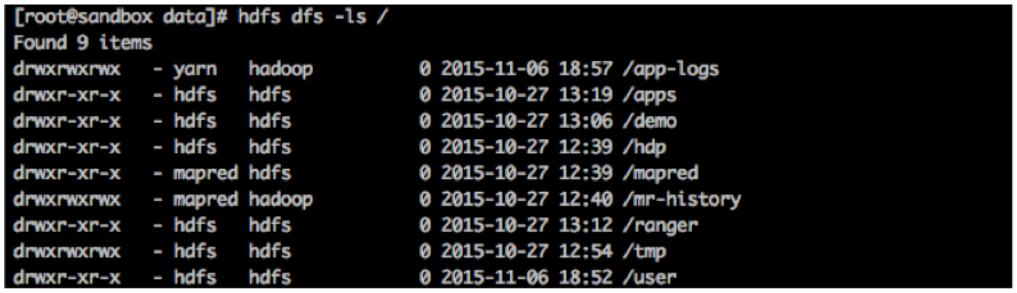
```
# hdfs dfs -ls
```

You do not have any files in `/user/root` yet, so no output is displayed.

   a. Run the `-ls` command, but this time specify the root HDFS folder:

```
# hdfs dfs -ls /
```

The output should looking something like:



```
[root@sandbox data]# hdfs dfs -ls /
Found 9 items
drwxrwxrwx   - yarn   hadoop          0 2015-11-06 18:57 /app-logs
drwxr-xr-x   - hdfs   hdfs            0 2015-10-27 13:19 /apps
drwxr-xr-x   - hdfs   hdfs            0 2015-10-27 13:06 /demo
drwxr-xr-x   - hdfs   hdfs            0 2015-10-27 12:39 /hdp
drwxr-xr-x   - mapred hdfs            0 2015-10-27 12:39 /mapred
drwxrwxrwx   - mapred hadoop          0 2015-10-27 12:40 /mr-history
drwxr-xr-x   - hdfs   hdfs            0 2015-10-27 13:12 /ranger
drwxrwxrwx   - hdfs   hdfs            0 2015-10-27 12:54 /tmp
drwxr-xr-x   - hdfs   hdfs            0 2015-11-06 18:52 /user
```

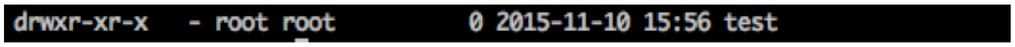
IMPORTANT: Notice how adding the `/` in the `-ls` command caused the contents of the root folder to display, but leaving off the `/` showed the contents of `/user/root`, which is the user root's home directory on `hadoop`. If you do not provide the path for any `hdfs dfs` commands, the user's home on `hadoop` is assumed.

b. Enter the following command to create a directory named `test` in HDFS:

```
# hdfs dfs -mkdir test
```

c. Verify the folder was created successfully:

```
# hdfs dfs -ls
```



```
drwxr-xr-x   - root root          0 2015-11-10 15:56 test
```

d. Create a couple of subdirectories of `test`:

```
# hdfs dfs -mkdir test/test1
# hdfs dfs -mkdir -p test/test2/test3
```

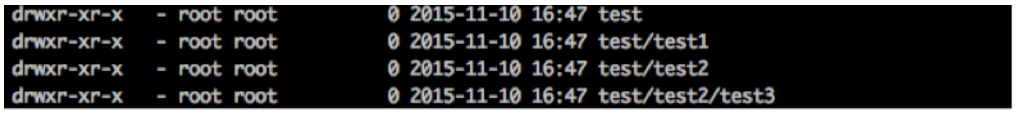
e. Use the `-ls` command to view the contents of `/user/root`:

```
# hdfs dfs -ls
```

Notice you only see the `test` directory. To recursively view the contests of a folder, use `-ls -R`:

```
# hdfs dfs -ls -R
```

The output should look like:



```
drwxr-xr-x   - root root          0 2015-11-10 16:47 test
drwxr-xr-x   - root root          0 2015-11-10 16:47 test/test1
drwxr-xr-x   - root root          0 2015-11-10 16:47 test/test2
drwxr-xr-x   - root root          0 2015-11-10 16:47 test/test2/test3
```

3. Delete a directory

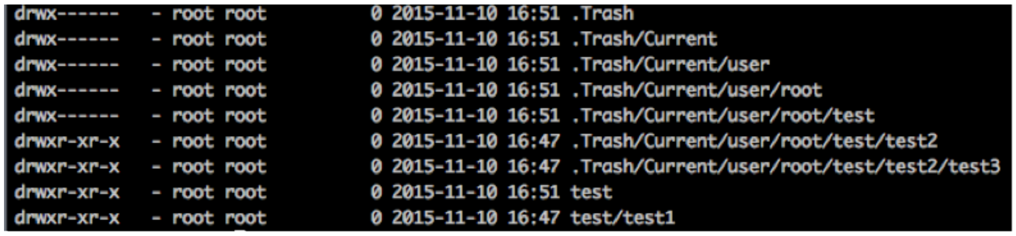
a. Delete the `test2` folder (and recursively its subcontents) using the `-rm -R` command:

```
# hdfs dfs -rm -R test/test2
```

b. Now run the `-ls -R` command:

```
# hdfs dfs -ls -R
```

The directory structure of the output should look like:


```
drwx------   - root root          0 2015-11-10 16:51 .Trash
drwx------   - root root          0 2015-11-10 16:51 .Trash/Current
drwx------   - root root          0 2015-11-10 16:51 .Trash/Current/user
drwx------   - root root          0 2015-11-10 16:51 .Trash/Current/user/root
drwx------   - root root          0 2015-11-10 16:51 .Trash/Current/user/root/test
drwxr-xr-x   - root root          0 2015-11-10 16:47 .Trash/Current/user/root/test/test2
drwxr-xr-x   - root root          0 2015-11-10 16:47 .Trash/Current/user/root/test/test2/test3
drwxr-xr-x   - root root          0 2015-11-10 16:51 test
drwxr-xr-x   - root root          0 2015-11-10 16:47 test/test1
```

NOTE: Notice Hadoop create a `.Trash` folder for the root user and moved the deleted content there. The `.Trash` folder empties automatically after a configured amount of time.

4. Upload a file to the HDFS
   a. Now put a file into the `test` folder.

   Change directories to `/root/spark/data/`:

```
# cd /root/spark/data/
```

   b. Notice this folder contains a file named `data.txt`

```
# tail data.txt
```

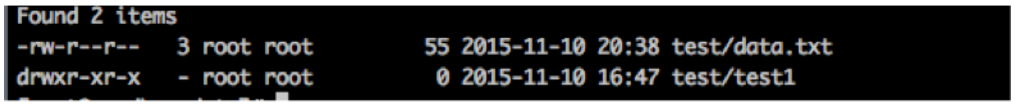
   c. Run the following `-put` command to copy `data.txt` into the `test` folder in HDFS:

```
# hdfs dfs -put data.txt test/
```

   d. Verify the file is in the HDFS by listing the contents of `test`:

```
# hdfs dfs -ls test
```

   The output should look like the following:

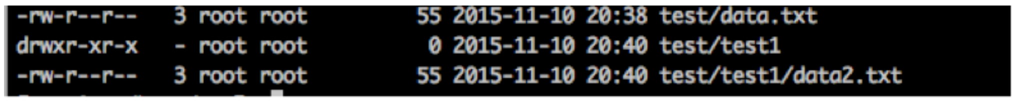

```
Found 2 items
-rw-r--r--   3 root root         55 2015-11-10 20:38 test/data.txt
drwxr-xr-x   - root root          0 2015-11-10 16:47 test/test1
```

5. Copy a file in the HDFS
   a. Now copy the `data.txt` file in test to another folder in HDFS using the `-cp` command:

```
# hdfs dfs -cp test/data.txt test/test1/data2.txt
```

   b. Verify the file is in both places by using the `-ls -R` command on `test`. The output should look like the following:

```
# hdfs dfs -ls -R test
```


```
-rw-r--r--   3 root root         55 2015-11-10 20:38 test/data.txt
drwxr-xr-x   - root root          0 2015-11-10 20:40 test/test1
-rw-r--r--   3 root root         55 2015-11-10 20:40 test/test1/data2.txt
```

   c. Now delete the `data2.txt` file using the `-rm` command

```
# hdfs dfs -rm test/test1/data2.txt
```

   d. Verify the `data2.txt` file is in the `.Trash` folder

6. View the contents of a file in the HDFS

   a. You can use the `-cat` command to view text files in the HDFS.

      Enter the following command to view the contents of `data.txt`:

```
# hdfs dfs -cat test/data.txt
```

   b. You can also use the `-tail` command to view the end of a file:

7. Getting a file from the HDFS

   a. See if you can figure out how to use the `-get` command to copy `test/data.txt` from the HDFS into your local `/tmp` folder.

8. The `getmerge` command

   a. Put the file `/root/spark/data/small_blocks.txt` into the `test` folder in HDFS. You should now have two files in test: `data.txt` and `small_blocks.txt`.

   b. Run the following `-getmerge` command:

```
# hdfs dfs -getmerge test /tmp/merged.txt
```

   c. What did the previous command do? Open the file `merged.txt` to see what happened.

RESULT

You should now be comfortable with executing the various HDFS commands, including creating directories, putting files in the HDFS, copy files out of the HDFS, and deleting files and folders.

Lab: Advanced RDD Programming

About This Lab

Objective:	To use advanced RDD transformations.
File locations:	`/root/spark/data/`
Successful outcome:	Find the top 3 airlines with the most flights Find the top 5 most common routes between cities Find the airline with the most delays over 15 minutes Find the most common plane for flights over 1500 miles
Before you begin	You should be logged in to your AWS instance

Lab Steps

Perform the following steps:

1. Put the required data from the lab from local into the HDFS
   a. From within your AWS instance, open a terminal.
   b. Navigate to the following location:

```
# cd /root/spark/data
```

   c. Put the following files into the HDFS:

```
flights.csv, airports.csv, carriers.csv, plane-data.csv
```

2. Explore the data that was just put into the HDFS, using your local machine
   a. Use the `head/vi/tail` command take a look at the data

```
flights.csv
```

Field	Index	Example data
`Month`	0	1
`DayofMonth`	1	3
`DayOfWeek`	2	4
`DepTime`	3	1738
`ArrTime`	4	1841
`UniqueCarrier`	5	WN

`FlightNum`	6	3948
`TailNum`	7	N467WN
`ElapsedTime`	8	63
`AirTime`	9	49
`ArrDelay`	10	1
`DepDelay`	11	8
`Origin`	12	JAX
`Dest`	13	FLL
`Distance`	14	318
`TaxiIn`	15	6
`TaxiOut`	16	8
`Cancelled`	17	0
`CancellationCode`	18	
`Diverted`	19	0

`carrier.csv`

Field	**Index**	**Example**
`Code`	0	WN
`Description`	1	Southwest

`airports.csv`

Field	**Index**	**Example**
`AirportCode`	0	00M
`Airport`	1	Thigpen
`City`	2	Bay Springs
`State`	3	MS
`Country`	4	USA
`Lat`	5	31.95376472

Long	6	-89.23450472

```
plane-data.csv
```

Field	Index	Example
Tailnum	0	N10156
Type	1	Corporation
Manufacturer	2	EMBRAER
Issue_date	3	02/13/2004
Model	4	EMB-145XR
Status	5	Valid
Aircraft_type	6	Fixed Wing Multi-Engine
Engine_type	7	Turbo-Fan
Year	8	2004

i. The charts above will be helpful when trying to access individual fields

3. The first goal of this lab is to find top 3 airlines with the most flights
   a. Create an RDD for `flight.csv` and split it into arrays:

```
>>> val flightRdd=sc.textFile("/user/root/flights.csv").
    map(line => line.split(","))
```

   b. This application looks like a word count. As a general rule of thumb, process the minimum amount of data to get the answer. Transform the RDD created above to only get the necessary fields, along with anything else needed for a word count:

```
>>> val carrierRdd = flightRdd.map(line => (line(5),1))
>>> carrierRdd.take(1)
```

   c. Reduce the RDD to get the number of flights for each airline.
   d. Using `sortByKey`, find the top 3 airlines.

4. Find the top 5 most common routes, between two cities
   a. This application also looks like a word count, but the key is made up of more then one field. Also, there might be more than one airport for each city, make sure to take that into account.
   b. Reuse the `flightRdd` created in 3a, and create an `airportsRdd` using `airports.csv`:

```
>>> val airportsRdd = sc.textFile("/user/root/airports.csv").
            map(line=> line.split(","))
```

   c. Create a new RDD using the smallest amount of required data, and join the `airportsRdd` to `flightsRdd`.

      i. Prep the `airportsRdd` and `flightRdd` to only keep what is needed:

```
>>> val cityRdd = airportsRdd.
            map(line=> (line(0), line(2)))
>>> val flightOrigDestRdd = flightRdd.
            map(line=> (line(12), line(13)))
```

      ii. Join the RDDs to get the correct city, retaining only the required data.

   d. Map the `citiesRdd` to a new RDD that is then ready to do a `reduceByKey`.

5. **CHALLENGE:** Find the longest departure delay for each airline if its over 15 minutes

   a. This application is similar to a word count, believe it or not.

   b. Filter out all departure delays less then 15 minutes

   c. Instead of adding together values, compare them to find the longest for each key

   HINT: `math.max(a,b)` returns the greater of the two values, make sure you're comparing `ints`, as the data is read as a string until casted.

6. **CHALLENGE:** Find the most common airplane model for flights over 1500 miles

   NOTE: Not all data is perfect (`plane-data.csv` has some missing values), so make sure to filter out airplane model records that don't contain 9 fields after it is split into an array.

SOLUTIONS

3. a:

```
>>> val flightRdd=sc.textFile("/user/root/flights.csv").
map(line => line.split(","))
```

3. b:

```
>>> val carrierRdd = flightRdd.map(line => (line(5),1))
>>> carrierRdd.take(1)
```

3. c:

```
>>> val carrierReduce = carrierRdd.reduceByKey((a,b) => a+b)
```

3. d:

```
>>> val carriersSorted = carrierReduce.map{case (a,b) => (b,a)}.
sortByKey(false)
>>> carriersSorted.take(3)
```

4. b:

```
>>> val airportsRdd = sc.textFile("/user/root/airports.csv").
map(line=> line.split(","))
```

4. c. i:

```
>>> val cityRdd = airportsRdd.map(line=> (line(0), line(2)))
>>> val flightOrigDestRdd = flightRdd.
map(line=> (line(12), line(13)))
```

4. c. ii:

```
>>> val origJoinRdd = flightOrigDestRdd.join(cityRdd)
>>> val destAndOrigJoinRdd = origJoinRdd.
map{case(a,(b,c))=> (b,c)}.join(cityRdd)
>>> val citiesCleanRdd = destAndOrigJoinRdd.values
```

4. d:

```
>>> val citiesReducedRdd = citiesCleanRdd.
map(line=> (line,1)).reduceByKey((a,b)=> a+b)
```

4. e:

```
>>> citiesReducedRdd.map{case (a,b)=> (b,a)}.
sortByKey(false).take(5)
```

5:

```
>>>flightRdd.filter(line=> line(11).toInt > 15).
map(line=> (line(5), line(11).toInt)).
reduceByKey((a,b)=> math.max(a,b)).take(10)
```

6:

```
>>> val airplanesRdd = sc.textFile("/user/root/plane-data.csv").
map(line=> line.split(",")).filter(line=> line.length == 9)
>>> val flight15Rdd = flightRdd.
filter(line=> line(14).toInt > 1500).map(line=> (line(7),1))
>>> val tailModelRdd = airplanesRdd.map(line=> (line(0),line(4)))
>>> flight15Rdd.join(tailModelRdd).
map{case (a,(b,c))=> (c,b)}.reduceByKey((a,b)=> a+b).
map(pair => pair.swap).sortByKey(false).take(2)
```

Lab: Parallel Programming on Spark

About This Lab

Objective:	Explore the Spark UI to see the tasks, stages, and DAG schedule of an application. Explore how partitioning affects number of tasks.

File locations:	HDFS:
	`/user/root/flight.csv`
	`/user/root/carriers.csv`

Successful outcome:	Use the UI to see how their application is performing Repartition data View the DAG schedule
Before you begin	You should be logged in to your AWS instance

Lab Steps

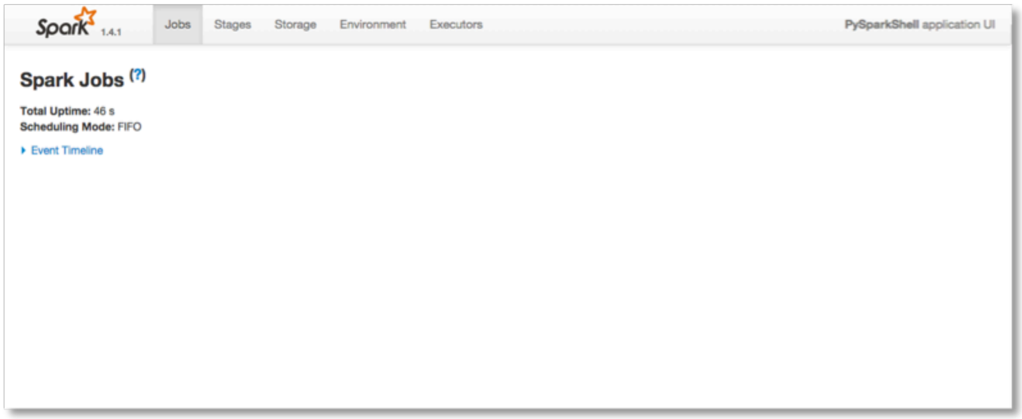
Perform the following steps:

1. Navigate to a fresh Spark web UI
   a. Close any REPL's currently open:

```
>>>exit()
```

   If it seems like the REPL is taking a long time to exit, hit `enter`.

   b. Start a new REPL.
   c. Open a web browser in `guacamole`:
      i. Navigate to `sandbox:4040`
      ii. Verify you see something like the image below:



Spark Jobs (?)

Total Uptime: 46 s
Scheduling Mode: FIFO

▸ Event Timeline

2. Create two RDDs and repartition the largest RDD to 40 partitions
   a. Create an RDD using `flights.csv`:
      i. The application will be joining data so split the data into K/V using `keyBy` and the `UniqueCarrier` field.
      ii. Check the number of partitions:

```
>>>val flightRdd = sc.textFile("/user/root/flights.csv").
map(line=> line.split(","))
>>>val flightsKVRdd=flightRdd.map(//key by 5th index, value 6th index)
>>>flightsKVRdd.partitions.size
```

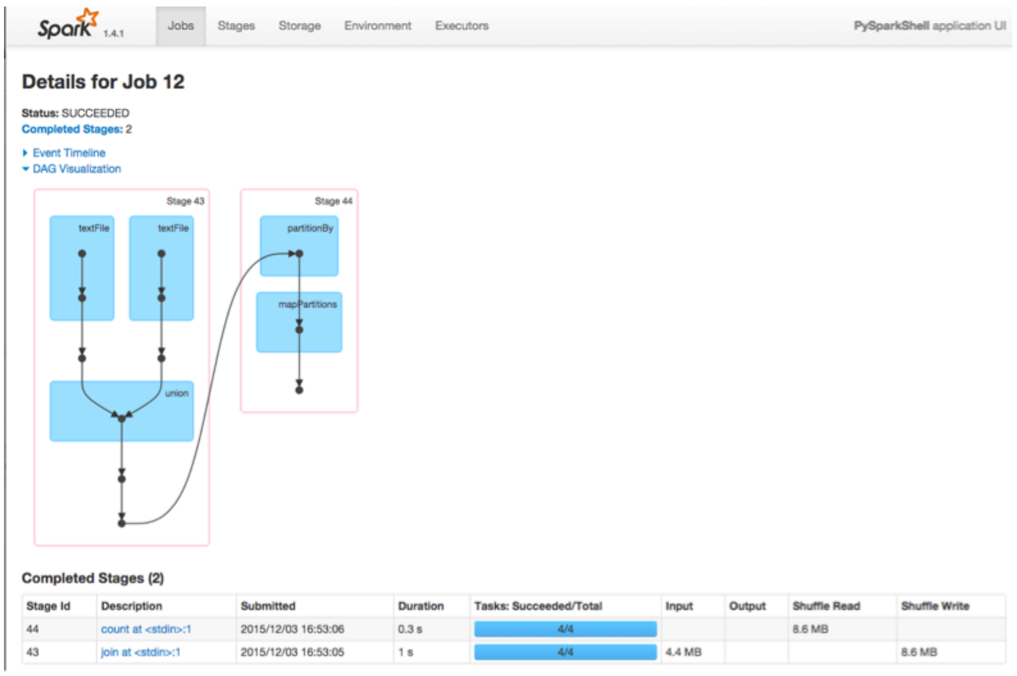
   b. Create an RDD using `carriers.csv`:
      i. Split the data into the K/V pairs:

```
>>> val carrierRdd =
            sc.textFile("/user/root/carriers.csv").
            map(line=> line.split(",")).
            map(line=> (line(0), line(1)))
```

3. Join the `flightRdd` to the `carrierRdd`
   a. Join the two RDDs and run a `count` on the new RDD:

```
>>>val joinedRdd = flightsKVRdd.join(carrierRdd)
>>>joinedRdd.count()
```

      i. Refresh the web UI.
      ii. Click into the stage and view the tasks.
      iii. Click on the dag visualizer to see the DAG created.
      iv. Note the different metrics.
      v. View the event timeline.



Details for Job 12

Status: SUCCEEDED
Completed Stages: 2

Completed Stages (2)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
44	count at <stdin>:1	2015/12/03 16:53:06	0.3 s	4/4			8.6 MB	
43	join at <stdin>:1	2015/12/03 16:53:05	1 s	4/4	4.4 MB			8.6 MB

b. Repeat 2a and 3, but repartition the `flightsKVRdd` to 10 partitions. Explore the tasks of the stages more in this example:

```
>>>val flightspartKVRdd=flightsKVRdd.repartition(10)
>>>flightspartKVRdd.partitions.size
>>>flightspartKVRdd.join(carrierRdd).count()
```

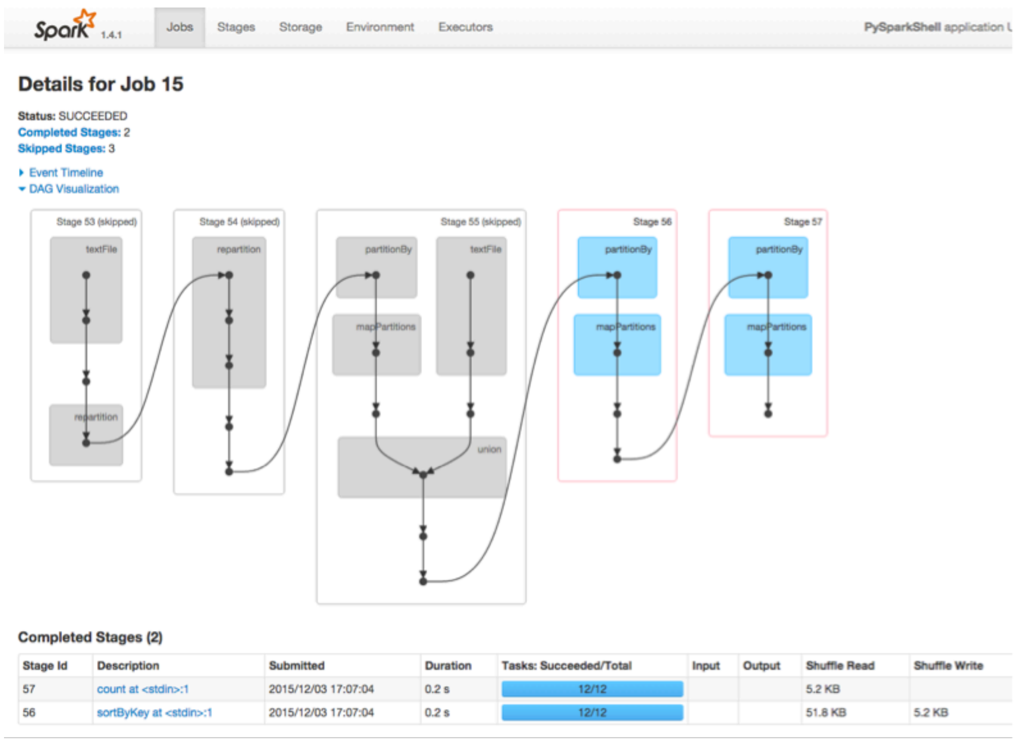
c. Find the number of flights using the 10 partition RDD by unique carrier and sort the list:

   i. Use `reduceByKey`, pattern matching and `sortByKey`.

   ii. Collect the results to the driver.

   iii. View the Web UI, repeating the Web UI steps from 3a.

   NOTE: If you see grey stages like below, its because Spark stores the intermediate files to local disk temporarily, so instead of re-processing all the data, it picks up the intermediate data and skips the stages from previous. Data is stored to disk temporarily during operations that require a shuffle.



Details for Job 15

Status: SUCCEEDED
Completed Stages: 2
Skipped Stages: 3

▸ Event Timeline
▾ DAG Visualization

Completed Stages (2)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
57	count at <stdin>:1	2015/12/03 17:07:04	0.2 s	12/12			5.2 KB	
56	sortByKey at <stdin>:1	2015/12/03 17:07:04	0.2 s	12/12			51.8 KB	5.2 KB

SOLUTIONS:

2. a:

```
>>>val flightRdd = sc.textFile("/user/root/flights.csv").
map(line=> line.split(","))
>>>val flightsKVRdd=flightRdd.map(line=> (line(5),line(6)))
>>>flightsKVRdd.partitions.size
```

2. b:

```
>>>val carrierRdd = sc.textFile("/user/root/carriers.csv").
map(line=> line.split(",")).map(line=> (line(0), line(1)))
```

3. b:

```
>>>val flightspartKVRdd=flightsKVRdd.repartition(10)
>>>flightspartKVRdd.partitions.size
>>>flightspartKVRdd.join(carrierRdd).count()
```

4. c:

```
>>>flightspartKVRdd.map{case (a,b)=> (a,1)}.
reduceByKey((a,b)=> a+b).join(carrierRdd).
map{case (a,(b,c))=> (b,c)}.
sortByKey(false).collect().foreach(println)
```

Lab: Caching Data with Spark

About This Lab

Objective:	Explore different persisting options and the speed improvements

File locations:	HDFS:
	`/user/root/flight.csv`
	`/user/root/carriers.csv`

Successful outcome:	See the benefits of using caching in Spark
Before you begin	You should be logged in to your AWS instance

Lab Steps

Perform the following steps:

1. Testing caching
    a. Perform a count on the RDD `joinedRdd` from the previous lab (if you deleted, repaste in the code to create it).
        i. Note the time it took to complete.
        ii. In the following steps, we will be comparing the time, so make sure to save the time in a notepad or write it down.
    b. Using the `cache` API, cache the `joinedRdd`.
        i. `cache` is not an action, so no data will be processed.
    c. Perform a count on the `joinedRdd` again.
        i. Note the time it took to complete. Was it more or less than in 2b? Why?
    d. Perform a `count` on the `joinedRdd` one more time.
        i. Notice the performance increase.
2. Exploring the `persist` options
    a. Restart the REPL to the clear cached RDD
        i. Recreate the `joinedRDD`:

```
>>> val flightRdd=sc.textFile("/user/root/flights.csv").
map(line=> line.split(",")).map(line=> (line(5),line(6)))

>>>val carrierRdd = sc.textFile("/user/root/carriers.csv").
map(line=> line.split(",")).map(line=> (line(0), line(1)))

>>>val joinedRdd = flightRdd.join(carrierRdd)
```

    b. Import the necessary libraries:

```
>>>import org.apache.spark.storage.StorageLevel._
```

c. Using the `persist` API, persist the RDD with `MEMORY_ONLY`:

```
>>>rdd.persist(MEMORY_ONLY)
```

   i. Run a `count()` a couple of times to put the data into memory.
   ii. Note the time of the 2nd `count()`.

d. Using the `unpersist` API, unpersist the dataset.

e. Persist the data to `DISK_ONLY`.

   i. Run a count a couple of times to put the data into memory.
   ii. Note the time of the 2nd count.
   Go ahead and try it with one or two other persistence levels.

RESULT

You have successfully used to caching and persistence to realize performance benefits.

Lab: Checkpointing and RDD Lineage

About This Lab

Objective:	Create a long iterative application that breaks lineage, and use checkpointing to fix the issue.
File locations:	No files
Successful outcome:	Successfully checkpoint an iterative application
Before you begin	You should be logged in to your AWS instance

Lab Steps

Perform the following steps:

1. Start by pasting the first line of code in
    a. This will create an RDD:

```
>>>var data = sc.parallelize(Array(1,2,3,4,5))
```

    b. Using the `toDebugString()` API, take a look at the lineage.

2. Creating an iterative application
    a. Paste the `for` loop in, notice the iterations that are being done:

```
>>> for(i <- 1 until 100){
            data = data.map(x=> x+1)
       }
```

    b. Notice the last RDD is still called `data`, and run a `toDebugString` and take a look at the lineage:

```
>>> data.toDebugString
```

    c. Perform a `count` on the same RDD above.

3. Increasing the length of the lineage
    a. Modify the `for` loop above by 50 iterations.
    b. Run a `count` on the RDD.
    c. Continue doing A and B until an error happens.
    d. When `count` fails, run a `toDebugString` on the subsequent RDD.

4. Enabling checkpointing
    a. Enable checkpointing:

```
>>> sc.setCheckpointDir("checkpointDir")
```

    b. Don't necessarily need to checkpoint every iteration, figure out a way to checkpoint every 7 iterations.

c. Recreate the base data:

```
>>>var data = sc.parallelize(Array(1,2,3,4,5))
```

d. Create the checkpoint, and perform an action to force Spark to checkpoint:

```
>>>for (i <- 1 to 1000){
            data=data.map(v=> v+1)
      //Create the checkpoint
            //Run an action, like count
            //Only do this every 7th iteration of I

      }
```

e. Modify the `for` loop back to 100 and perform an action:

```
>>> data.take(1)
```

f. Modify it to the point where it broke in 4c.

g. Use the `toDebugString` on the above code to see what checkpointing is doing.

h. It works!

SOLUTIONS

4. d:

```
>>>for (i <- 1 to 1000){
      data=data.map(i=> i+1)
   if (i%7 == 0) {
            data.checkpoint()
            data.count()
      }
}
>>>data.take(1)
>>>data.toDebgugString
```

Lab: Build and Submit an Application to YARN

About This Lab

Objective:	Create a Standalone application and submit the application to YARN
File locations:	Project: `/spark/scala/projects/WordCount/src/main/scala/stub` HDFS: `/user/root/selfishgiants.txt`
Successful outcome:	Standalone Application should complete and print out the words from a file
Before you begin	You should be logged in to your AWS instance

Lab Steps

Perform the following steps:

1. Develop an application for `spark-shell`
    a. Start by changing the directory to the application directory:

```
$cd ~/spark/scala/projects/WordCount/src/main/scala/stub
```

    b. This is a simple exercise focusing on building and submitting an application with Spark
    c. In a text editor (like vi or gedit), open the file `WordCount.scala`

```
        $vi myapp.scala
```

    d. Import the correct libraries.
    e. Create an object called `WordCount` and set up the `main`.
    f. Create the `spark conf`.
        i. Name the application `MyApp`.
        ii. Set up Kryo Serialization, don't worry about registering a class for this exercise.
        iii. Set `spark.speculation` to `true`.
    g. Create the Spark Context.
    h. Put the `selfishgiants.txt` file in the HDFS if its not already there.
        i. In the application, perform a wordcount on the `sleepinggiants.txt` file and print out the final value of the top 10 most said words.

i. Stop the Spark Context.

2. Build the application

    a. Navigate to the root directory of the application and type the following:

```
$ cd ~/ spark/scala/projects/WordCount/
$ mvn package
```

3. Submitting an application

    a. Using `spark-submit`, submit the application to the cluster.

        i. Use `yarn-client master`, with number of executors as 2, and executor memory of 1GB.

        ii. Once submitted, open Firefox and navigate to the YARN History server at `sandbox:18080` and find your application.

SOLUTIONS

Sample solution code for this lab is contained within the VM.

Spark-Submit code:

```
spark-submit --class stub.WordCount --master yarn-client --num-executors 2 --
executor-memory 1g target/WordCount-1.0-SNAPSHOT.jar
```

Lab: Using Accumulators to Check Data Quality

About This Lab

Objective:	Work with Spark Accumulators
File locations:	HDFS: `/user/root/plane-data.csv`
Successful outcome:	Developer should create an accumulator to check data quality
Before you begin	You should be logged in to your AWS instance

Lab Steps

Perform the following steps:

1. Open up the REPL
2. Count the number of planes that don't have all the data filled out
    a. Create an RDD from the `plane-data.csv` file and split it out:

```
>>> val planeRdd=sc.textFile("/user/root/plane-data.csv").
map(line => line.split(","))
```

    b. Create an accumulator to do the counting.
    c. Using `foreach`, check to see if the size of the resulting array is 9, if not increment the accumulator:

```
>>> planeRdd.foreach(line =>
            //Add the logic here
            )
```

    d. Print the accumulator value to the screen.
        i. `foreach` is an action and will trigger data to be processed.

```
>>>badData.value
```

SOLUTIONS

2. b:

```
>>>val badData=sc.accumulator(0)
```

2. c:

```
>>>planeRdd.foreach(line =>
      if(line.length != 9){
            badData+=1
      })
```

Lab: Using Broadcast Variables

About This Lab

Objective:	Join a large file in the HDFS efficiently to a small local lookup file using a broadcast variable.
File locations:	HDFS: `/user/root/flight.csv` Local: `/root/spark/data/carriers.csv`
Successful outcome:	Developers will successfully use a broadcast variable to join a large table to a lookup table.
Before you begin	You should be logged in to your AWS instance.

Lab Steps

Perform the following steps:

1. Open up the REPL if not still open from the previous lab
2. Create a dictionary of the `carrier.csv` file and broadcast it
    a. Paste the following code into the REPL. This will create a `map` to broadcast:

```
>>> import scala.io.Source
>>> val list =
Source.fromFile("/root/spark/data/carriers.csv").getLines.map(line=>line.spli
t(",")).map(line => (line(0),line(1))).toMap
```

    b. Broadcast the map created in 3a.

3. Join the broadcast variable and the `flights.csv` file
    a. Create an RDD of `flights.csv` and `split` the flights into an array of elements keeping the flight number and unique carrier:

```
>>> val flightRdd=sc.textFile("/user/root/flights.csv").
      map(line=> line.split(",")).map(line=>(line(6),line(5)))
```

    b. Using the `broadcast.value` API, create a new RDD with the flight number and carrier name, this is called a broadcast join.
    c. Verify the broadcast join worked by running a take and return a few records.

SOLUTIONS

2. b:

```
>>>val carrierbc=sc.broadcast(list)
```

3. b:

```
>>> val flightUpdate=flightRdd.map{case (a,b)=> (a,carrierbc.value(b))}
```

Lab: Spark SQL Using UDFs

About This Lab

Objective:	Read a text file from the HDFS, create a Dataframe, query the Dataframe with a UDF and Dataframe operations
File locations:	HDFS: `/user/root/flight.csv`
Successful outcome:	Developer should work heavily with dataframes, including creating, saving, loading, and manipulating. Developer should also be able to use UDFs.
Before you begin	You should be logged in to your AWS instance

Lab Steps

Perform the following steps:

1. Open up the REPL if not still open from the previous lab

   NOTE: when the `spark-shell` is started, a `sqlContext` is created. Import the `implicits` library to allow Scala to do some conversions for you:

```
>>>import sqlContext.implicits._
```

2. Create a dataframe from the `flights.csv` file

   a. Create an RDD from the `flights.csv` file:

```
>>> val flightRdd=sc.textFile("/user/root/flights.csv").map(line =>
line.split(","))
```

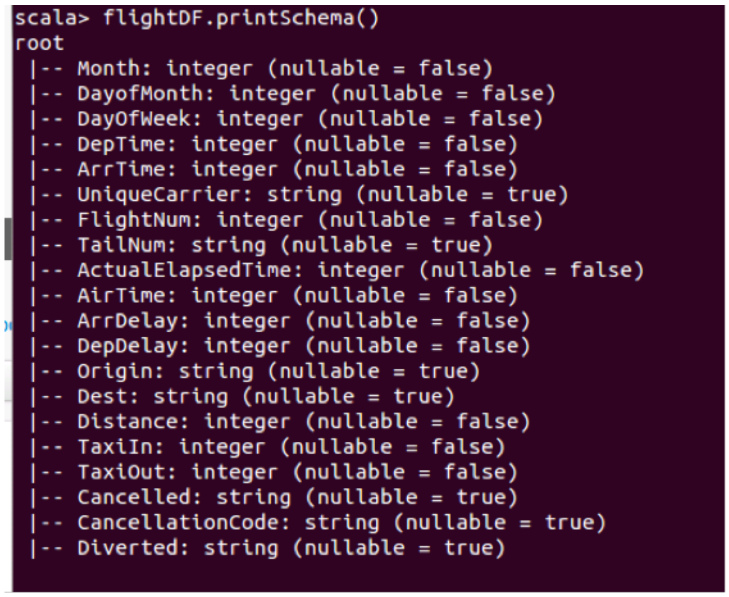
   b. Create the `case class` to be used in a DataFrame. Copy and paste the code below:

```
>>> case class Flight(
Month: Int,
DayofMonth: Int,
DayOfWeek: Int,
DepTime: Int,
ArrTime: Int,
UniqueCarrier: String,
FlightNum: Int,
TailNum: String,
ActualElapsedTime: Int,
AirTime: Int,
ArrDelay: Int,
DepDelay: Int,
Origin: String,
Dest: String,
Distance: Int,
TaxiIn: Int,
TaxiOut: Int,
Cancelled: String,
CancellationCode: String,
Diverted: String)
```

c. Map the `flightRdd` into the `case class`, and convert to dataframe:

```
>>> val flightDF = flightRdd.map(f => Flight(f(0).toInt,
f(1).toInt,
f(2).toInt,
f(3).toInt,
f(4).toInt,
f(5).toString,
f(6).toInt,
f(7).toString,
f(8).toInt,
f(9).toInt,
f(10).toInt,
f(11).toInt,
f(12).toString,
f(13).toString,
f(14).toInt,
f(15).toInt,
f(16).toInt,
f(17).toString,
f(18).toString,
f(19).toString)).toDF()
```

d. Using the `printSchema()` API, examine the schema that was just created for the dataframe.



```
scala> flightDF.printSchema()
root
 |-- Month: integer (nullable = false)
 |-- DayofMonth: integer (nullable = false)
 |-- DayOfWeek: integer (nullable = false)
 |-- DepTime: integer (nullable = false)
 |-- ArrTime: integer (nullable = false)
 |-- UniqueCarrier: string (nullable = true)
 |-- FlightNum: integer (nullable = false)
 |-- TailNum: string (nullable = true)
 |-- ActualElapsedTime: integer (nullable = false)
 |-- AirTime: integer (nullable = false)
 |-- ArrDelay: integer (nullable = false)
 |-- DepDelay: integer (nullable = false)
 |-- Origin: string (nullable = true)
 |-- Dest: string (nullable = true)
 |-- Distance: integer (nullable = false)
 |-- TaxiIn: integer (nullable = false)
 |-- TaxiOut: integer (nullable = false)
 |-- Cancelled: string (nullable = true)
 |-- CancellationCode: string (nullable = true)
 |-- Diverted: string (nullable = true)
```

3. Save the dataframe as a parquet file to the HDFS

a. Use the `DataframeWriter` API:

```
>>>flightDF.write.format("parquet").save("/user/root/flights.parquet")
```

b. In a new terminal window, verify the file was written to the HDFS.

4. Create a new dataframe from the saved parquet file in 3a
    a. Use the `DataframeReader` API:

```
>>>val dfflight=sqlContext.read.//Try to finish
```

    b. Explore the schema to see what's created, it should look familiar.
5. Explore flights with Departure Delays using dataframe operations
    a. Find the highest average delays by airport origin:

```
>>> dfflight.select($"Origin", $"DepDelay").//Try to finish
```

    b. Find the percentage of flights delayed/total flights for each airline, and sort the list to get the most delayed airlines, by airline code.
        i. Create a UDF to check if the flight is delayed or not:

```
>>>def delay_check(x:Int) : Int = {
            if(x>0) return 1
            else return 0
      }

>>> val depUDF = udf((x: Int) => delay_check(x))
```

        ii. Select the columns using the UDF to check if a flight was delayed or not:

```
>>>val delayDF = dfflight.select($"UniqueCarrier",
/* Use the UDF here */.alias("IsDelayed"), $"DepDelay")
```

        iii. Using `groupBy`, and the `agg` operator, create a count of the `DepDelay` to get total number of flights, and a sum of the `IsDelayed` Column:

```
>>>val delayGroupDF = delayDF.
groupBy($"UniqueCarrier").
agg("/* Add the mapping here */")
```

        iv. Create a UDF to get the percentage of delayed flights:

```
>>>val calc_percent = udf((s: Int, c: Int) => s.toFloat/c)
```

        v. Create the final DataFrame by using a `select`, the UDF, and a `sort`, then show it:

```
>>> delayGroupDF.select($"UniqueCarrier",
      calc_percent(/* use the correct columns for udf*/).
      alias("Percentage")).sort(/*sort on percent*/).
      show()
```

    c. **CHALLENGE:** Find the top 5 airlines with longest average flight distance.
6. **CHALLENGE:** Explore taxi times
    a. Find the top 5 airports with the largest average taxi time in.
    b. Find the top 5 airports with the shortest average taxi time out.

SOLUTIONS

4. c:

```
>>>val dfflight=sqlContext.read.format("parquet").
      load("/user/root/flights.parquet")
```

5. a:

```
 >>> dfflight.select($"Origin", $"DepDelay").groupBy($"Origin").avg().
      withColumnRenamed("AVG(DepDelay)","DelayAvg").sort($"DelayAvg".desc).sh
ow()
```

5. b. ii:

```
>>>val delayDF = dfflight.select($"UniqueCarrier",depUDF($"DepDelay").
      alias("IsDelayed"), $"DepDelay")
```

5. b. iii:

```
>>>val delayGroupDF = delayDF.groupBy($"UniqueCarrier").
      agg("IsDelayed" -> "sum", "DepDelay" -> "count")
```

5. b. v:

```
>>> delayGroupDF.select($"UniqueCarrier",
calc_percent($"SUM(IsDelayed)".cast("int"),$"COUNT(DepDelay)".cast("int")).
alias("Percentage")).sort($"Percentage".desc).show()
```

5. c:

```
>>>dfflight.select($"UniqueCarrier",$"Distance").
groupBy($"UniqueCarrier").avg().sort($"AVG(Distance)".desc).show(5)
```

6. a:

```
>>> dfflight.select($"Origin", $"TaxiIn").
groupBy($"Origin").avg().sort($"AVG(TaxiIn)".desc).show(5)
```

6. b:

```
>>> dfflight.select($"Origin", $"TaxiOut").
groupBy($"Origin").avg().sort($"AVG(TaxiOut″.asc).show(5)
```

Lab: Spark SQL with Hive

About This Lab

Objective:	Using tables already existing in Hive, perform analytics.
File locations:	Data is stored in Hive
Successful outcome:	Developer should interact with Hive metastore and be able to query data
Before you begin	You should be logged in to your AWS instance

Lab Steps

Perform the following steps:

1. Open up the REPL if not still open from the previous lab
2. Find all the airplanes that fly the longest route
3. Using the `hivecontext`, create two dataframes. One from the table `flight.flights` and the other from `flight.planes`
4. Sort the flights dataframe using distance to find the longest flight, do a take to look at the distance of the longest flight
5. Filter all flights on the longest flight distance, and return the tail numbers of those flights
6. Join the `tailnums` to the planes RDD to get the models of the airplanes
7. Perform a `count` to find the most common airplane models

SOLUTIONS

3:

```
>>>sqlContext.sql("Use flight")
>>>val flights = sqlContext.table("flights")
>>>val planes = sqlContext.table("planes")
```

4:

```
>>>flights.select($"distance").sort($"distance".desc).show(5)
```

5:

```
>>>val longflights = flights.filter($"distance"===4962)
.select($"tailnum").distinct
```

6:

```
>>>val longflightplanes = longflights.join(planes,
longflights("tailnum")===planes("tailnum") , "inner")
```

7:

```
>>> longflightplanes.select($"model").groupBy($"model")
.count.show()
```

Lab: Spark Streaming WordCount

About This Lab

Objective:	Create a Streaming application that outputs all words said in a Dstream, utilize the `nc` command to simulate a data source
File locations:	No files
Successful outcome:	Output words from simulated source to screen
Before you begin	You should be logged in to your AWS instance

Lab Steps

Perform the following steps:

1. Close the REPL
2. Start a new REPL specifying the following information:

```
#spark-shell --master local[2]
```

3. Create a Spark Streaming application that performs a wordcount on a socket text stream
    a. Import the Streaming library:

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

    b. Create the streaming context, with a 5 second batch duration:

```
>>>val ssc = new StreamingContext(sc, Seconds(5))
```

    c. Create the Dstream using `sandbox` and `port 9999`:

```
>>>val inputDS = ssc.socketTextStream("sandbox",9999)
```

    d. Transform the RDD to create a wordcount application, split on spaces:

```
>>>val wc = inputDS.flatMap(line=> line.split(" ")).
map(word=> (word,1)).reduceByKey((a,b) => a+b)
```

    e. Print out the output to the client:

```
>>>wc.print()
```

    f. Set the log level to `ERROR` to avoid clutter:

```
>>>setLogLevel("ERROR")
```

    g. Start the streaming application:

```
>>>ssc.start()
```

NOTE: You will see an error when it starts, it's waiting for an input connection.

4. If a new terminal run the following command to start outputting data:

```
#nc -lkv 9999
```

a. Start typing words separated by `space`, hit `return` occasionally to submit them.
b. Look at the other terminal where the streaming application is running.
c. While the application is running, navigate to the web UI in Firefox and explore the web UI tabs:

```
sandbox:4040
```

d. To quit the streaming application, press `control-d`, `control-c` for the terminal running `nc`.

RESULT

You have now successfully created and run a stateless application.

Lab: Spark Streaming with Windows

About This Lab

Objective:	Create a Spark Streaming utilizing a `window` function to find words read in the previous 10 seconds
File locations:	No files
Successful outcome:	Developer will use the `window` function to create a windowed wordcount.
Before you begin	You should be logged in to your AWS instance

Lab Steps

Perform the following steps:

1. Close the REPL
2. Start a new REPL specifying the following information:

```
#spark-shell --master local[2]
```

3. Create a Spark Streaming application that performs a wordcount on a socket text stream using the window function `reduceByKeyAndWindow`. Set a 10 second window with a 2 second sliding duration

    a. Import the Streaming library:

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

    b. Create the streaming context, with a 2 second batch duration:

```
>>>val ssc = new StreamingContext(sc, Seconds(2))
```

    c. Create the Dstream using `sandbox` and `port 9999`:

```
>>>val inputDS = ssc.socketTextStream("sandbox",9999)
```

    d. For this lab, enable checkpointing the lazy way:

```
>>>ssc.checkpoint("checkpointDir")
```

    e. Transform the `inputDS` to use a window and then a `reducebykey`

```
>>>val windowDS = inputDS.window(Seconds(10),Seconds(2)).
flatMap(line => line.split(" ")).map(word=>(word,1)).
reduceByKey((a,b)=> a+b)
```

    f. Print the output out:

```
>>>windowDS.print()
```

    g. To avoid clutter the output, set the log level to `ERROR`:

```
>>>sc.setLogLevel("ERROR")
```

h. Start the streaming application:

```
>>>ssc.start()
```

4. In a new terminal run the following command to start outputting to the stream:

```
#nc -lkv 9999
```

a. Start typing words separated by `space`, hit `return` occasionally to submit them.

b. Look at the other terminal where the streaming application is running.

c. While the application is running, navigate to the web UI in Firefox and explore the web UI tabs:

```
sandbox:4040
```

d. To quit the streaming application, press `control-d`, `control-c` for the terminal running `nc`.

RESULT

You have now successfully created an application that utilizes the `window` function.