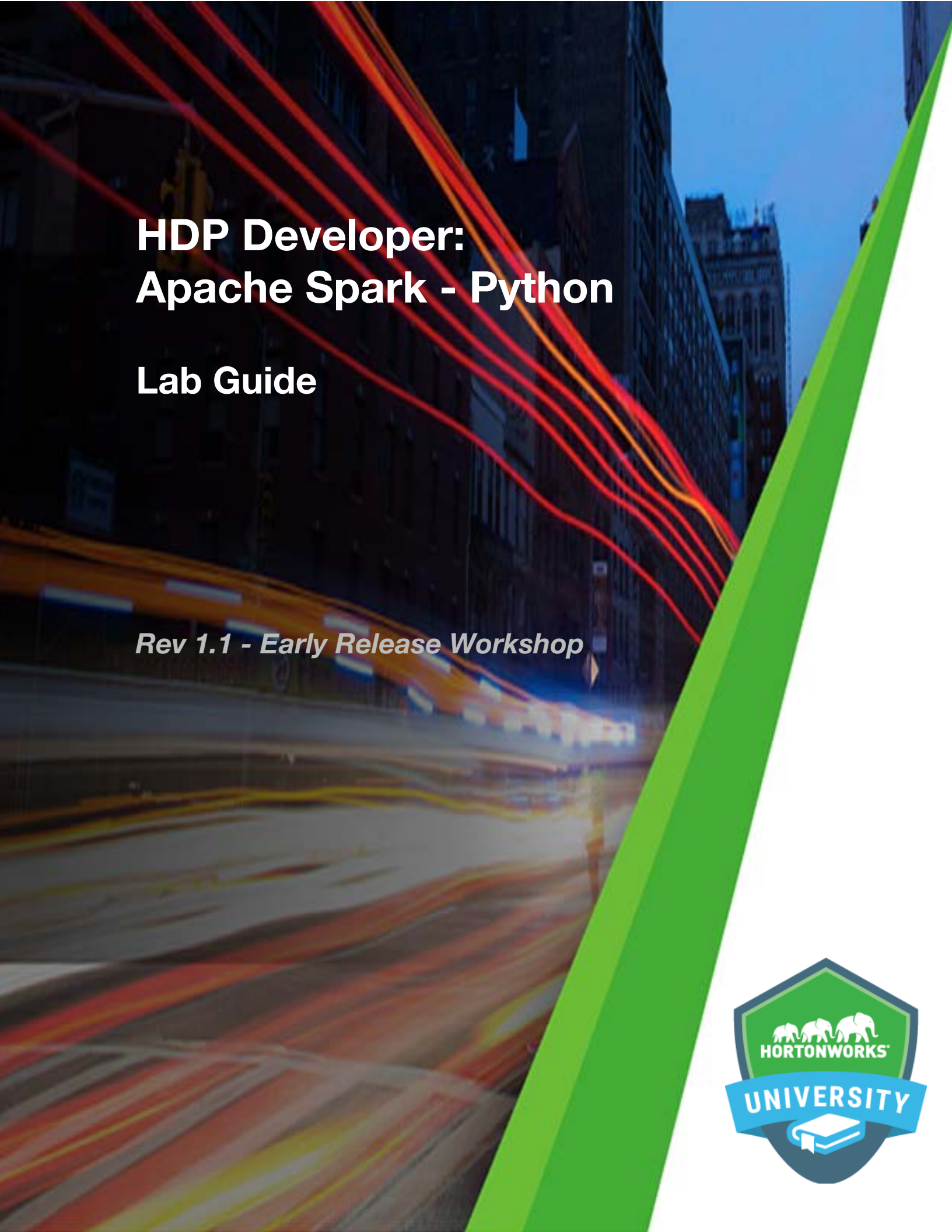


HDP Developer: Apache Spark - Python

Lab Guide

Rev 1.1 - Early Release Workshop

HORTONWORKS UNIVERSITY

Become a ***Hortonworks Certified Professional*** and establish your credentials:

• HDP Certified Developer: for Hadoop developers using frameworks like Pig, Hive, Sqoop and Flume.

• HDP Certified Administrator: for Hadoop administrators who deploy and manage Hadoop clusters.

• HDP Certified Developer: Java: for Hadoop developers who design, develop and architect Hadoop-based solutions written in the Java programming language.

• HDP Certified Developer: Spark: for Hadoop developers who write and deploy applications for the Spark framework.

• HDF Certified Professional: for DataFlow Operators responsible for building and deploying HDF workflows.

How to Register: Visit www.examslocal.com and search for "Hortonworks" to register for an exam. The cost of each exam is $250 USD, and you can take the exam anytime, anywhere using your own computer. For more details, including a list of exam objectives and instructions on how to attempt our practice exams, visit http://hortonworks.com/training/certification/

Earn Digital Badges: Hortonworks Certified Professionals receive a digital badge for each certification earned. Display your badges proudly on your résumé, LinkedIn profile, email signature, etc.

Self Paced Learning Library

On Demand Learning

Hortonworks University Self-Paced Learning Library is an on-demand dynamic repository of content that is accessed using a Hortonworks University account. Learners can view lessons anywhere, at any time, and complete lessons at their own pace. Lessons can be stopped and started, as needed, and completion is tracked via the Hortonworks University Learning Management System.

Hortonworks University courses are designed and developed by Hadoop experts and provide an immersive and valuable real world experience. In our scenario-based training courses, we offer unmatched depth and expertise. We prepare you to be an expert with highly valued, practical skills and prepare you to successfully complete Hortonworks Technical Certifications.

Target Audience: Hortonworks University Self-Paced Learning Library is designed for those new to Hadoop, as well as architects, developers, analysts, data scientists, and IT decision makers. It is essentially for anyone who desires to learn more about Apache Hadoop and the Hortonworks Data Platform.

Duration: Access to the Hortonworks University Self-Paced Learning Library is provided for a 12-month period per individual named user. The subscription includes access to over 400 hours of learning lessons.

The online library accelerates time to Hadoop competency. In addition, the content is constantly being expanded with new material, on an ongoing basis.

Visit: http://hortonworks.com/training/class/hortonworks-university-self-paced-learning-library/

Table of Contents

Pre-Lab Setup

About This Lab

Objective: To setup VM environment
Successful outcome: User will setup the cluster and verify login

Before you begin Get your AWS IP Address

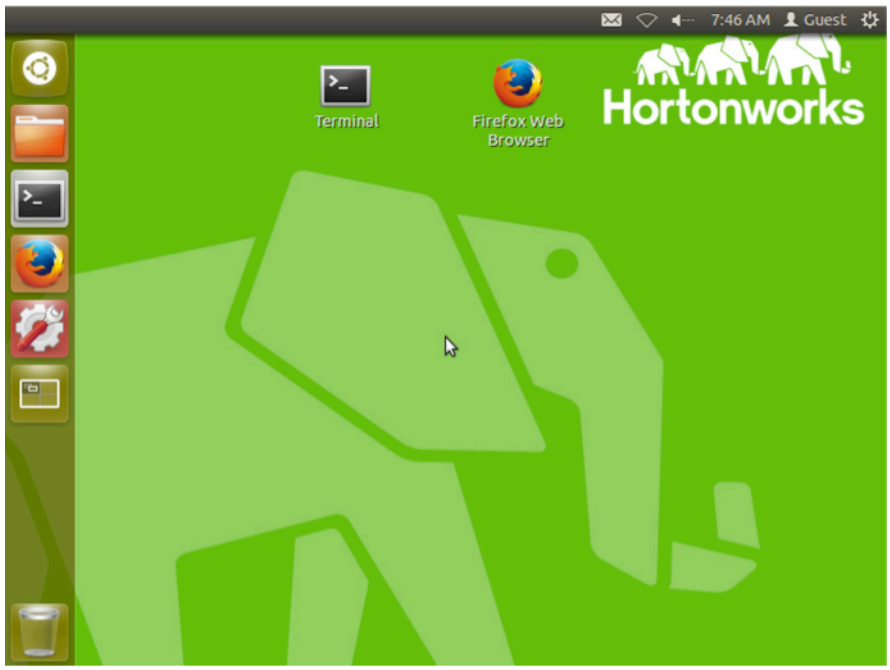
Lab Steps

Perform the following steps:

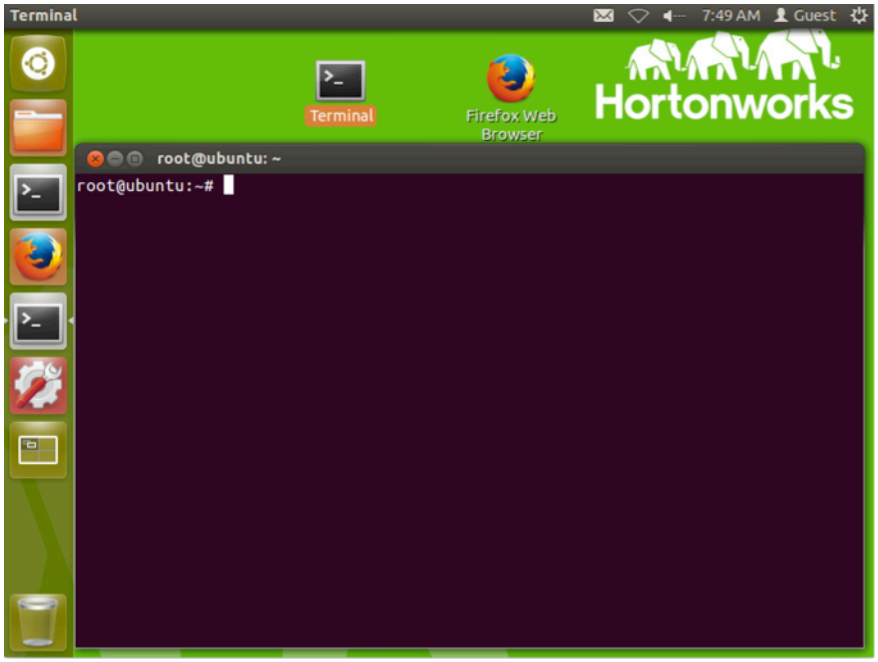
1. **Start the VM**

   a. If applicable, start VMWare Player (or Fusion) on your local machine, select the course VM from the list of virtual machines, then click the Play virtual machine link.

   b. You should see the desktop of your local or cloud-hosted VM:



c. Open a Terminal by double-clicking the shortcut on the desktop:

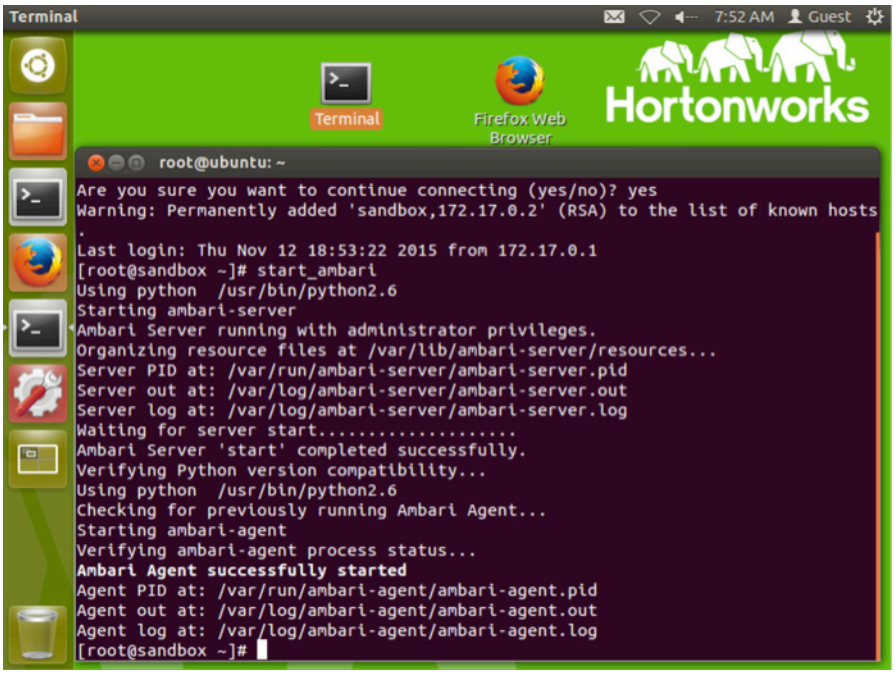


2. Verify that the cluster is running

a. Navigate into the Docker sandbox instance.

Note:
Type "yes" if asked "are you sure you want to continue connecting".

```
root@ubuntu:~# ssh sandbox
[root@sandbox ~]# start_ambari
```



b. From the command line, enter the following command, which displays the usage of the hdfs dfsadmin utility:

```
# su -l hdfs -c "hdfs dfsadmin"
```

Note:
The "dfs" in dfsadmin stands for distributed filesystem, and the dfsadmin utility contains administrative commands for communicating with the Hadoop Distributed File System.

c. Notice the dfsadmin utility has a -report option, which outputs the current health of your cluster. Enter the following command to view this report:

```
# su -l hdfs -c "hdfs dfsadmin -report"
```

d. What is the configured capacity of your distributed filesystem?

Answer: Look for the value of "Configured Capacity" at the start of the output.

e. What is the present capacity?

Answer: Look for the value of "Present Capacity" at the start of the output.

f. How much of your distributed filesystem is used right now?

Answer: Look for the value of "DFS Used."

g. What do you think an "Under-replicated block" is?

Answer: Data in HDFS is chunked into blocks and copied to various nodes in the cluster. If a particular block does not have enough copies, it is referred to as "under replicated."

h. How many available DataNodes does your cluster have?

Answer: 1

3. **View the Processes on the Cluster Nodes**

   a. Enter the jps command, which lists all Java processes running on this machine. While your specific processes and they order they are presented in may look slightly different than the list below, you should still see the NameNode process running:

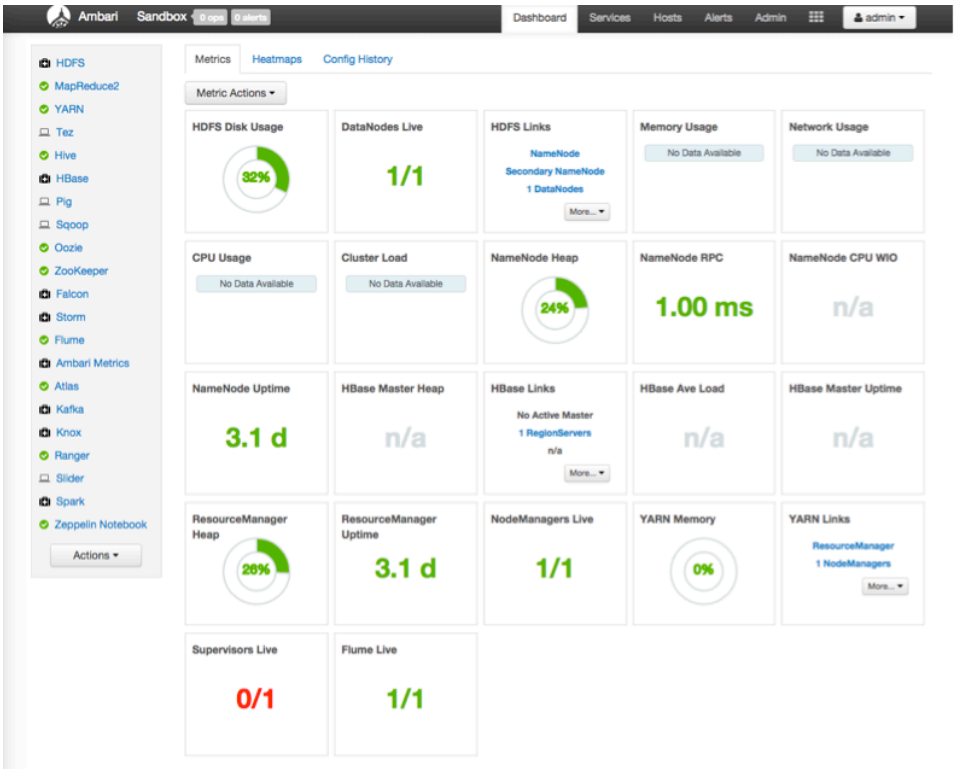
```
# jps
3706 ResourceManager
2988 QuorumPeerMain
3675 RunJar
4032 RunJar
3740 NodeManager
3188 Nfs3
3186 Portmap
3738 JobHistoryServer
2556 DataNode
2557 SecondaryNameNode
2560 NameNode
3712 ApplicationHistoryServer
3511 RunJar
24669 -- process information unavailable
5516 AmbariServer
31813 Jps
3029 Bootstrap
```

4. **Login into the AWS instance**

   a. 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
```

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



Result

We have verified you're able to login, and the cluster is setup and running, we are now ready.

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 directoires. 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**

   a. 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

   b. 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.

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

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

d. Verify the folder was created successfully

```
# hdfs dfs -ls
```

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

e. Create a couple of subdirectories of test:

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

f. 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 -putt 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
```

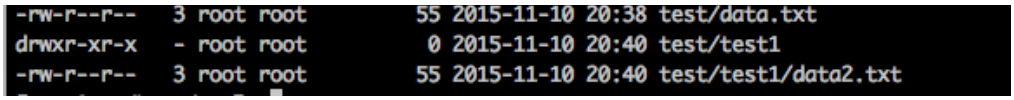
5. Copy a file in the HDFS

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
```



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 follwionig command to view the contest of `data.txt`

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

b. You can also use the 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: Use the REPL to Create and Manipulate RDD's

About This Lab

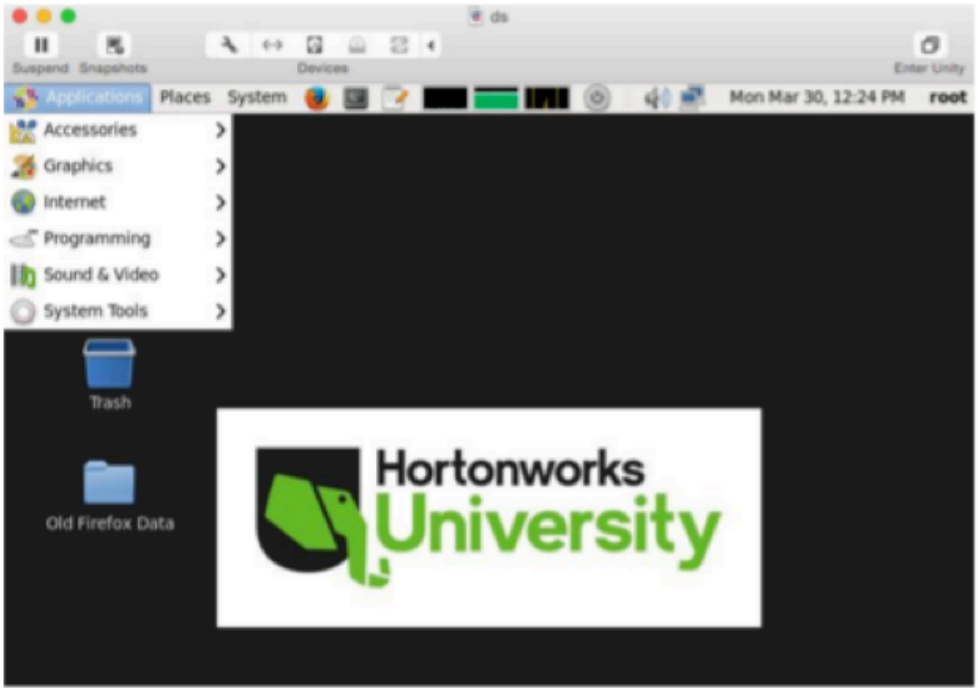
Objective:	Use the REPL to Create and Manipulate RDDs
File locations:	`/root/spark/data/selfishgiant.txt`
Successful outcome:	User will have started the shell and perform some basic RDD transformations and actions.
Before you begin:	Finish Pre-Lab

Lab Steps

Perform the following steps:

1. **Start 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. First, you must be in the sandbox:

```
# ssh sandbox
```

c. To open the spark shell, type 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
```

2. **View the raw data for this lab**

a. In a new terminal window, ssh to sandbox and change directories to the data directory

```
#ssh sandbox
# cd ~/spark/data
```

b. View the data file "selfishgiant.txt"

```
# tail selfishgiant.txt
```

c. This file contains the short story Selfish Giant.

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

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

```
>>> 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)
```

```
>>> baseRdd=sc.textFile("file:///root/spark/data/selfishgiant.txt")
>>> baseRdd.take(1)
15/12/02 16:05:09 WARN DomainSocketFactory: The short-circuit local reads feature ca
[u'The Selfish Giant']
```

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

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

```
>>> splitRdd = baseRdd.flatMap(lambda line: line.split(" "))
>>> splitRdd.take(5)
[u'The', u'Selfish', u'Giant', u'by', u'Oscar']
```

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

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

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

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

f. Run an action to get output.

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

```
>>> reducedRdd.take(20)
[(u'', 30), (u'lived,', 1), (u'all', 11), (u'stole', 1), (u'Through', 1), (u'cried', 3), (u'hand,', 1), (u'when', 9), (u'over', 4), (u'What', 1),
through', 3), (u'"Spring', 1), (u'go', 1), (u'cold', 1), (u'hate', 1), (u'bitterly.', 1), (u'lessons', 1), (u'TRESPASSERS', 1)]
```

4. **Challenge: Find the ten most prominent words**

```
>>> reducedRdd.map(lambda (a,b) : (b,a)).sortByKey(ascending=False).take(10)
[(148, u'the'), (85, u'and'), (44, u'he'), (38, u'to'), (32, u'was'), (30, u''), (28, u'in'), (22, u'a'), (21, u'
were'), (19, u'of')]
```

Result

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

Lab: Advanced RDD Programming

About This Lab

Objective:	To use the advanced RDD transformations that were covered in the previous lesson

File locations:	HDFS:
	`/user/root/flights.csv`
	`/user/root/carriers.csv`
	`/user/root/plane-data.csv`

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 mins Find the most common plane for flights over 1500 miles
Before you begin	You should be logged in to your lab environment
Related lesson:	***Advanced RDD Programming***

Lab Steps

Perform the following steps:

1 . Put the required data for 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 . Find the top 3 airlines with the most flights

a. Create an RDD for `flights.csv`:

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

b. This application looks like a word count. As a general rule of thumb, process the minimal 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.

```
>>> carrierRdd = flightRdd.map(lambda 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`:

```
>>> airportsRdd = sc.textFile("/user/root/airports.csv").map(lambda
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's needed.

```
>>> cityRdd = airportsRdd.map(lambda line: (line[0], line[2]))
>>> flightOrigDestRdd = flightRdd.map(lambda 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: `max(a,b)` returns the greater of the two values, make sure you're comparing `ints`, the data is read in 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), make sure to filter out airplane model records that don't contain 9 fields after it is split into an array.

SOLUTIONS

3. a:

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

3. b:

```
 >>> carrierRdd = flightRdd.map(lambda line: (line[5],1))
>>> carrierRdd.take(1)
```

3. c:

```
>>> cReducedRdd = carrierRdd.reduceByKey(lambda a,b: a+b)
```

3. d:

```
>>> carriersSorted = cReducedRdd.map(lambda (a,b): (b,a)).sortByKey(ascending=False)
>>> carriersSorted.take(3)
```

4.b:

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

4. c. i:

```
 >>> cityRdd = airportsRdd.map(lambda line: (line[0], line[2]))
 >>> flightOrigDestRdd = flightRdd.map(lambda line: (line[12], line[13]))
```

4. c. ii:

```
>>> origJoinRdd = flightOrigDestRdd.join(cityRdd)
>>> destAndOrigJoinRdd = origJoinRdd.map(lambda (a,b): (b[0],b[1])).join(cityRdd)
>>> citiesCleanRdd = destAndOrigJoinRdd.values()
```

4. d:

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

4. e:

```
>>> citiesReducedRdd.map(lambda (a,b): (b,a)).sortByKey(ascending=False).take(5)
```

5:

```
>>> flightRdd.filter(lambda line: int(line[11]) > 15) \
.map(lambda line: (line[5], line[11])).reduceByKey(lambda a,b:
max(int(a),int(b))).take(10)
```

6:

```
>>> airplanesRdd = sc.textFile("/user/root/plane-data.csv") \
.map(lambda line: line.split(",")) \
.filter(lambda line:len(line) == 9)
>>> flight15Rdd = flightRdd \
.filter(lambda line: int(line[14]) > 1500) \
.map(lambda line: (line[7],1))
>>> tailModelRdd = airplanesRdd \
.map(lambda line: (line[0],line[4]))
>>> flight15Rdd.join(tailModelRdd) \
.map(lambda (a,b): (b[1],b[0])) \
.reduceByKey(lambda a,b: a+b) \
.map(lambda (a,b): (b,a)).sortByKey(ascending=False).take(2)
```

Lab: Parallel Programming with 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/flights.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 lab environment
Related lesson:	***Parallel Programming with Spark***

Lab Steps

Perform the following steps:

1 . Navigate to a fresh Spark web UI

a. Close any REPL's currently open

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

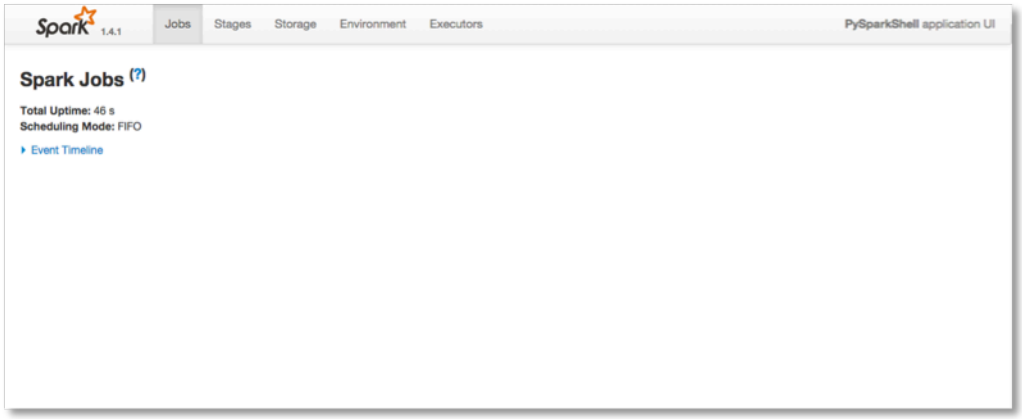
If it seems like the REPL is taking a long time to exit, hit `enter`.

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:



2 . Create two RDDs

a. Create an RDD using `flights.csv`.

i. The application will be joining data, so split the data into K/V using `map` and the `UniqueCarrier` field.

ii. Check the number of partitions:

```
>>>flightRdd=sc.textFile("/user/root/flights.csv") \
.map(lambda line: line.split(","))
>>>flightsKVRdd=flightRdd.map(##Key with 5th index, keep the 6th
value)
>>>flightsKVRdd.getNumPartitions()
```

```
>>> flightRdd = sc.textFile("/user/root/flights.csv").map(lambda line: line.split(","))
>>> flightsKVRdd = flightRdd.map(lambda line: (line[5],line[6]))
>>> flightsKVRdd.getNumPartitions()
16/02/08 11:53:41 WARN DomainSocketFactory: The short-circuit local reads feature cannot be
 libhadoop cannot be loaded.
2
```

b. Create an RDD using `carriers.csv`

i. Split the data into the K/V pairs:

```
>>> carrierRdd = sc.textFile("/user/root/carriers.csv")\
.map(lambda line: line.split(",")) \
.map(lambda 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:

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

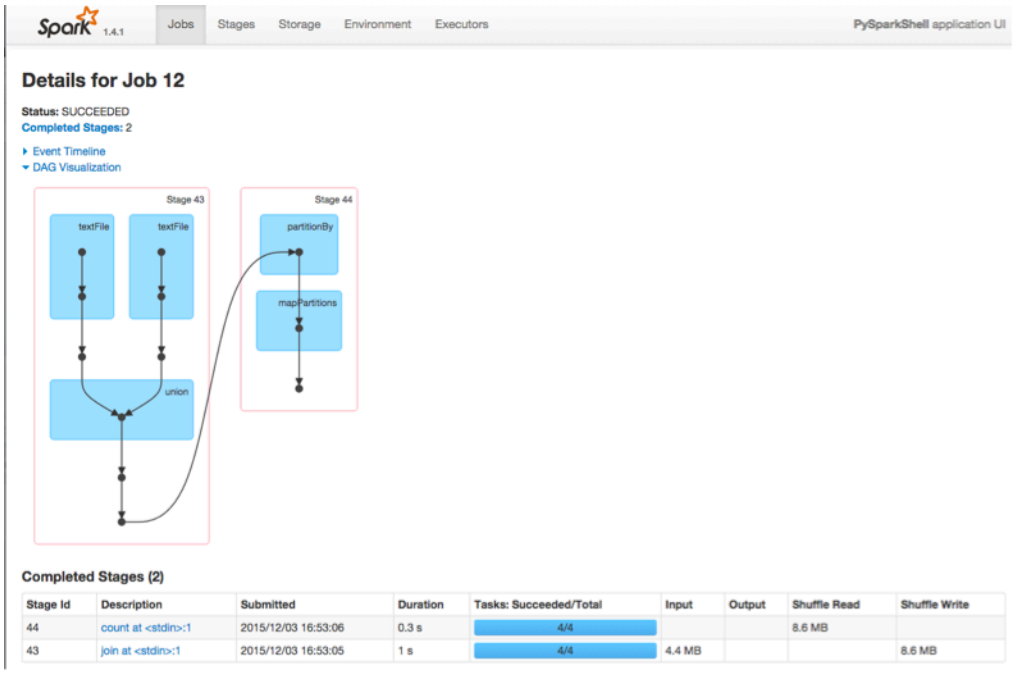
i. Refresh the web UI.

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:



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

```
>>>flightspartKVRdd=flightsKVRdd.repartition(10)
>>>flightspartKVRdd.getNumPartitions()
>>>flightspartKVRdd.join(carrierRdd).count()
```

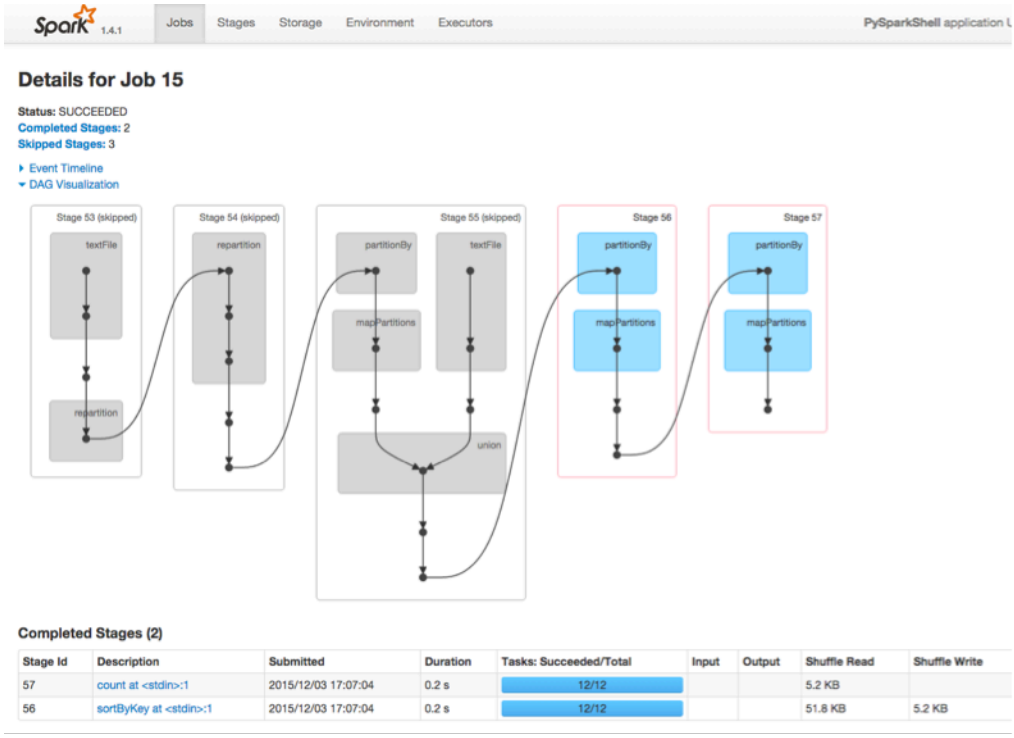
c. Find the number of flights using the 10 partition RDD by unique carrier and sort the list.

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

ii. Collect the results to the driver.

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

NOTE: If you see grey stages like below, it's 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.



SOLUTIONS

2. a:

```
>>>flightRdd=sc.textFile("/user/root/flights.csv") \
.map(lambda line: line.split(","))
>>>flightsKVRdd=flightRdd.map(lambda line: (line[5], line[6]))
>>>flightsKVRdd.getNumPartitions()
```

2. b:

```
>>> carrierRdd = sc.textFile("/user/root/carriers.csv")\
.map(lambda line: line.split(",")) \
.map(lambda line: (line[0], line[1]))
```

3. b:

```
>>>flightspartKVRdd=flightsKVRdd.repartition(10)
>>>flightspartKVRdd.getNumPartitions()
>>>flightspartKVRdd.join(carrierRdd).count()
```

4. c:

```
>>> flightspartKVRdd.map(lambda (a,b): (a,1)) \
.reduceByKey(lambda a,b: a+b).join(carrierRdd) \
.map(lambda (a,b): (b[0],b[1])) \
.sortByKey(ascending=False).collect()
```

Lab: Caching Data with Spark

About This Lab

Objective:	Explore different persisting options and the speed improvements

File locations:	HDFS:
	`/user/root/flights.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 lab environment
Related lesson:	***Caching and Persisting Data***

Lab Steps

Perform the following steps:

1 . Testing caching

a. Perform a count on the RDD `joinedRdd` from the lab (if you deleted, repaste in the code to create it)

i. Note the time it took to complete.

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

```
15/12/03 14:55:17 INFO DAGScheduler: Job 8 finished: count at <stdin>:1, took 8.028970 s
205432
```

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.

```
15/12/03 14:58:16 INFO DAGScheduler: Job 10 finished: count at <stdin>:1, took 0.639217 s
205432
```

2 . Exploring the persist options

a. Restart the REPL to the clear cached RDD.

i. Recreate the `joinedRDD`:

```
>>> flightRdd=sc.textFile("/user/root/flights.csv").map(lambda line:
line.split(",")).map(lambda line: (line[5], line[6]))
>>>carrierRdd = sc.textFile("/user/root/carriers.csv").map(lambda
line: line.split(",")).map(lambda line: (line[0], line[1]))
>>>joinedRdd = flightRdd.join(carrierRdd)
```

b. Import the necessary libraries:

```
>>>from pyspark import StorageLevel
```

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

```
>>> joinedRdd.persist(StorageLevel.MEMORY_ONLY)
PythonRDD[11] at RDD at PythonRDD.scala:43
```

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.

```
>>> joinedRdd.unpersist()
15/12/03 15:22:34 INFO PythonRDD: Removing RDD 11 from persistence list
15/12/03 15:22:34 INFO BlockManager: Removing RDD 11
PythonRDD[11] at RDD at PythonRDD.scala:43
```

e. Persist the data to `DISK_ONLY`.

```
>>> joinedRdd.persist(StorageLevel.DISK_ONLY)
PythonRDD[11] at RDD at PythonRDD.scala:43
```

i. Run a count a couple of times to put the data into memory.

ii. Note the time of the 2nd count.

f. Go ahead and try it with one or two other persistence levels.

Result

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

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 lab environment
Related lesson:	***Caching and Persisting Data***

Lab Steps

Perform the following steps:

1 . Start by pasting in the first line of code

a. This will create an RDD:

```
>>>data = sc.parallelize([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 x in range(100):
. . . data = data.map(lambda i: i+1)
```

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

```
>>> print(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 `toDebugString` on the RDD.

c. Continue doing steps 3a and 3b until an error occurs.

d. When `toDebugString` fails, run a count on the subsequent RDD.

4 . Enabling checkpointing

a. Enable checkpointing:

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

b. It isn't necessary to checkpoint every iteration; figure out a way to checkpoint every 7 iterations.

c. Recreate the base data:

```
>>>data = sc.parallelize([1,2,3,4,5]
```

d. Create the checkpoint:

```
>>>for x in range(1000):
...  ##Create the checkpoint
...##Only do it every 7th iteration of i
...data=data.map(lambda i: i+1)
```

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 x in range(100):
if x%7 == 0:
data.checkpoint()
 data=data.map(lambda i: i+1)
>>>data.take(1)
>>>print(data.toDebugString())
```

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:
	`/root/spark/python/projects/myapp`
	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 lab environment
Related lesson:	***Creating Spark Applications***

Lab Steps

Perform the following steps:

1 . **Develop an application for `pyspark`**

   a. Start by copying the directory `/root/spark/python/projects/myapp/` to your working directory.

   b. This is a simple exercise focusing on building and submitting an application with Spark.

   c. Open the `myapp.py` in a text editor.

      i. The code should look like something that would be copied line by line into the REPL with a basic Python wrapper around it.

   d. Import the correct libraries.

   e. Create the spark `conf`:

      i. Name the application `WordCount`.

      ii. Set `spark.speculation` to `true`.

   f. Create the `spark context`.

   g. Put the `selfishgiants.txt` file in the HDFS if its not already there.

   h. 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 . Submitting an application

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

NOTE: Specify the version of Python you are using by adding it before your submit command: `PYSPARK_PYTHON=/usr/bin/python spark-submit …`

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

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:

```
PYSPARK_PYTHON=/usr/bin/python spark-submit --master yarn-client \
--num-executors 2 --executor-memory 1g myapp.py
```

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 lab environment
Related lesson:	***Advanced Features and Improving Performance***

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:

```
>>> planeRdd=sc.textFile("/user/root/plane-data.csv") \
.map(lambda 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.

i. Create a function to do this, pass an array and the accumulator as the input.

ii. Pass the function into a map.

d. Print the accumulator value to the screen.

i. `foreach` is an action and will trigger data to be processed:

```
>>>print(badData.value)
```

SOLUTIONS

2. b:

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

2. c. i:

```
>>>def dataCheck(line,dataCounter):
if len(line) != 9:
dataCounter += 1
```

2. c. ii:

```
>>>planeRdd.foreach(lambda line: dataCheck(line, badData))
```

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/flights.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 lab environment
Related lesson:	***Advanced Features and Improving Performance***

Lab Steps

Perform the following steps:

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

2 . Create a dictionary of the `carrier.csv` file and broadcast it

a. Navigate to the `/root/spark/python/stubs/` directory and view the `lab9.py` file.

b. Execute the code in the REPL using the `execfile` command:

```
>>>execfile("/root/spark/python/stubs/lab9.py")
```

c. Verify a dictionary named "`result`" was created:

```
>>>print(result)
>>>type(result)
```

```
: 'Aerotaxis De Aguascalientes S.A. De C.V.', 'JK (1)': "Sunworld Int'l Airways Inc.", 'UX': 'Air Europa
', 'UT': 'Union De Transports Aeriens', 'UP': 'Bahamasair Holding Limited', 'US': 'US Airways Inc. (Merg
ed with America West 9/05. Reporting for both starting 10/07.)', 'UR': 'Empire Airlines Inc. (1)', 'LRQ'
: 'Air Liberte', 'UN': 'Transaero Airlines', 'UH': 'US Helicopter Corporation', 'UD': 'Fast Air Carrier
Ltd.', 'UA': 'United Air Lines Inc.', 'UC': 'Ladeco', 'IDQ': 'Intl Air Service Co. Ltd.', 'PMT': 'Pilgri
m Aviation & Airlines', 'CGL': "Viking Int'l Airlines Inc.", 'CGO': 'Chicago Air Taxi', 'CGA': 'Chicago
Airlines', '4B': 'Olson Air Service', 'PMA': 'Pacific Missionary Aviation', 'H5': 'Magadan Airlines', 'C
US': 'Custom Air Service', 'CUQ': 'Caicos Caribbean Airways', 'WIA': 'Wisc Air', 'WIL': "Will's Air", 'W
IR': 'Williams Air Inc.', '0MQ': 'Multi-Aero', 'MST': 'Midwest Aviation', 'CIS': 'Cal Sierra Airlines',
'CIQ': 'A/S Conair', 'SRA': 'Star Aviation', 'AQQ': 'Air Charter (Safa)', 'DAN': 'Danbury Airlines', 'DA
U': 'Dauphin Island Airways', '4Y': 'Yute Air Aka Flight Alaska', '4W': 'Warbelow', '4T': 'Belair Airlin
es Ltd.', '4R': 'Regent Air Corporation', '4S': 'Sol Air (Aero Hunduras)', '4N': 'Air North', '4M': 'LAN
 Argentina', '4H': 'Belize Trans Air', 'MM (1)': 'Sociedad Aeronautica De Med', '4E': 'Tanana Air Servic
e', 'DWN': 'Dawn Air', 'OOZ': 'Sun Aire Lines'}
>>> type(result)
<type 'dict'>
```

d. Broadcast the dictionary created in 2b.

```
>>>carrierbc=sc.broadcast(result)
```

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:

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

```
>>>carrierbc=sc.broadcast(result)
```

3. b:

```
>>>flightUpdate=flightRdd \
.map(lambda (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/flights.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 lab environment
Related lesson:	***Spark SQL and Dataframes***

Lab Steps

Perform the following steps:

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

a. Import the `Row` module from `pyspark`

```
>>>from pyspark import Row
```

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

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

```
>>> flightRdd=sc.textFile("/user/root/flight.csv").map(lambda line:
line.split(","))
```

b. Create a RDD of flight objects:

```
>>> flightORdd=flightRdd.map(lambda f: Row(Month=int(f[0]), \
DayOfMonth=int(f[1]), \
DayofWeek=int(f[2]), \
DepTime=int(f[3]), \
ArrTime=int(f[4]), \
UniqueCarrier=f[5], \
FlightNum=f[6], \
TailNum= f[7], \
ElapsedTime=int(f[8]), \
AirTime=int(f[9]), \
ArrDelay=int(f[10]), \
DepDelay=int(f[11]), \
Origin=f[12], \
Dest=f[13], \
Distance=int(f[14]), \
TaxiIn=int(f[15]), \
TaxiOut=int(f[16]), \
Cancelled=f[17], \
CancellationCode=f[18], \
Diverted=f[19]))
```

c. Use the `createDataFrame` to create a dataframe:

```
>>> flightDF=sqlContext.createDataFrame(flightORdd)
```

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

```
>>> flightDF.printSchema()
root
 |-- AirTime: long (nullable = true)
 |-- ArrDelay: long (nullable = true)
 |-- ArrTime: long (nullable = true)
 |-- CancellationCode: string (nullable = true)
 |-- Cancelled: string (nullable = true)
 |-- DayFfWeek: long (nullable = true)
 |-- DayOfMonth: long (nullable = true)
 |-- DepDelay: long (nullable = true)
 |-- DepTime: long (nullable = true)
 |-- Dest: string (nullable = true)
 |-- Distance: long (nullable = true)
 |-- Diverted: string (nullable = true)
 |-- ElapsedTime: long (nullable = true)
 |-- FlightNum: string (nullable = true)
 |-- Month: long (nullable = true)
 |-- Origin: string (nullable = true)
 |-- TailNum: string (nullable = true)
 |-- TaxiIn: long (nullable = true)
 |-- TaxiOut: long (nullable = true)
 |-- UniqueCarrier: 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 `DataframeReaderAPI`.

```
>>>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(dfflight.Origin, dfflight.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, then select the fields. The UDF will be using an integer and a UDF, so import the libraries:

```
>>>from pyspark.sql.functions import udf
>>>from pyspark.sql.types import IntegerType
>>>def delay_check(x):
     if x > 0:
             return 1
     else:
             return 0
>>> depUDF = udf(delay_check, IntegerType())
```

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

```
>>>delayDF = dfflight.select(dfflight.UniqueCarrier, \
##Use UDF here##.alias("IsDelayed"), dfflight.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

```
>>>delayGroupDF = delayDF \
.groupBy(delayDF.UniqueCarrier).agg(##Add dict here##)
```

iv. Create a UDF to get the percentage of delayed flights, import the Float library as well:

```
>>>from pyspark.sql.types import FloatType
>>>calc_percent = \
udf(lambda s,c: (float(s)/c), FloatType())
```

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

```
>>> delayGroupDF.select(delayGroupDF.UniqueCarrier, \
calc_percent(##Use the correct columns for the 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:

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

5. a:

```
>>> dfflight.select(dfflight.Origin, dfflight.DepDelay) \
.groupBy('Origin').avg() \
.withColumnRenamed("AVG(DepDelay)",  "DelayAvg") \
.sort('DelayAvg', ascending=False).show()
```

5. b. ii:

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

5. b. iii:

```
>>>delayGroupDF = delayDF.groupBy(delayDF.UniqueCarrier) \
.agg({"IsDelayed": "sum", "DepDelay": "count"})
```

5. b. v:

```
>>> delayGroupDF.select(delayGroupDF.UniqueCarrier, \
calc_percent("SUM(IsDelayed)","COUNT(DepDelay)") \
.alias("Percentage")).sort("Percentage", ascending=False).show()
```

5. c:

```
>>> dfflight.select("UniqueCarrier", "Distance") \
.groupBy("UniqueCarrier").avg() \
.sort("AVG(Distance)", ascending=False).show(5)
```

6. a:

```
>>> dfflight.select("Origin", "TaxiIn") \
.groupBy("Origin").avg() \
.sort("AVG(TaxiIn)", ascending=False).show(5)
```

6. b:

```
>>> dfflight.select("Origin", "TaxiOut") \
.groupBy("Origin").avg() \
.sort("AVG(TaxiOut)", ascending=True).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 lab environment
Related lesson:	***Spark SQL and Dataframes***

Lab Steps

Perform the following steps:

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

a Verify the `sqlContext` is of the type `HiveContext`:

```
>>>type(sqlContext)
```

```
>>> type(sqlContext)
<class 'pyspark.sql.context.HiveContext'>
```

2. Use the database "`flight`"

```
>>>sqlContext.sql("USE flight")
```

3. Find all the airplanes that fly the longest route

4. Using the `hivecontext`, create two dataframes. One from the table `flights` and the other from `planes`

5. Sort the `flights` dataframe, using `distance` to find the longest flight, do a `take` to look at the distance of the longest flight

6. Filter all flights on the longest flight distance, and return the tail numbers of those flights

7. Join the `tailnums` to the planes RDD to get the models of the airplanes

8. Perform a `count` to find the most common airplane models

SOLUTIONS

4:

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

5:

```
>>>flights.sort("distance", ascending=False).take(1)
```

6:

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

7:

```
>>>longflightplanes = longflights\
.join(planes, 'tailnum' , 'inner')
```

8:

```
>>> 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 lab environment
Related lesson:	***Spark Streaming***

Lab Steps

Perform the following steps:

1 . Close the REPL

2 . Start a new REPL specifying the following information:

```
#pyspark --master local[2]
```

3 . Create a Spark Streaming application that performs a wordcount on a socket text stream

a. Import the Streaming library:

```
>>>from pyspark.streaming import StreamingContext
```

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

```
>>>ssc = StreamingContext(sc, 5)
```

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

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

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

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

e. Print out the output to the client:

```
>>>wc.pprint()
```

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

```
>>>sc.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 . In a new terminal, run the following command to start outputting data:

```
#nc -lkv 9999
```

a. Start typing words separated by space, press 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 lab environment
Related lesson:	***Spark Streaming***

Lab Steps

Perform the following steps:

1. **Close the REPL**

2. **Start a new REPL specifying the following information:**

```
#pyspark --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:

```
>>>from pyspark.streaming import StreamingContext
```

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

```
>>>ssc = StreamingContext(sc, 2)
```

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

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

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

```
>>>ssc.checkpoint("hdfs:///user/root/checkpointDir")
```

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

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

f. Print the output out:

```
>>>windowDS.pprint()
```

g. To avoid cluttering the output, set the `loglevel` 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, press 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.

Classes Available Worldwide Through Our Partners

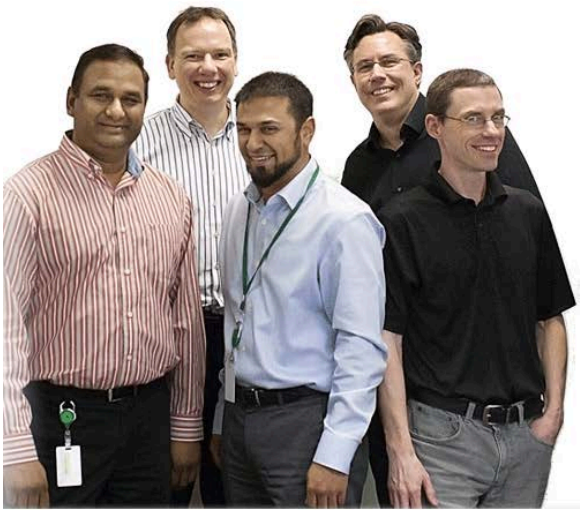
Study Options Worldwide

In combination with our partner providers, classes are often available in numerous locations across the world.

Private On-site Training

Hortonworks training in-house covers all of our basic coursework, and provides a more intimate setting for 6 or more students.
Contact us for more details

Learn from the company focused solely on Hadoop.



What Makes Us Different?

1. Our courses are designed by the **leaders and committers** of Hadoop
2. We provide an **immersive** experience in **real-world** scenarios
3. We prepare you to **be an expert** with highly valued, **fresh skills**
4. Our courses are available **near you**, or accessible **online**

Hortonworks University courses are designed by the leaders and committers of Apache Hadoop. We provide immersive, real-world experience in scenario-based training. Courses offer unmatched depth and expertise available in both the classroom or online from anywhere in the world. We prepare you to be an expert with highly valued skills and for Certification.