Spark Fundamentals

Getting started with Spark

Contents

Getting started with Spark

Spark is built around speed and the ease of use. In this section, you will see for yourself how easy it is to get started using Spark with IBM BigInsights. You will be doing some interactive analysis with the Spark Shell, which comes in two flavors. It is available in either Scala or Python. Scala runs on the Java VM and is thus a good way to use existing Java libraries.

Spark's primary abstraction is a distributed collection of items called a Resilient Distributed Dataset or RDD. In a subsequent lab exercise, you will learn more about the details of RDD. In this lab exercise, you'll get to see quickly how it works. RDDs have actions, which return values, and transformations, which return pointers to new RDD. In this lab exercise, you can decide if you wish to do the exercises in the Scala shell or the Python shell – both goes through the same steps, just slightly different syntax.

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

- Start the Spark shell with Scala and Python
- Perform basic RDD actions and transformations
- Use caching to speed up repeated operations

Allow 30-45 minutes to complete this section of lab.

1.1 Setting up the lab environment

Download the QSE v4 or use the Cloud environment. Review the README file that comes with the QSE for instructions on getting started.

This lab was designed with the Spark cluster on the cloud using an older version of Spark, so the screenshots and texts will be slightly different if you are using the latest QSE. The lab was also designed with Spark 1.1, but tested against the latest QSE (version 4) with Spark 1.2.1.

If you are using the QSE image, set up the following variables in the ~/.bashrc file to make it easier to work with Spark.

```
export SPARK_HOME=/usr/iop/current/spark-client
```

Be sure you refer to the section to get the labfiles onto the VM image. You will need these files throughout the lab exercise.

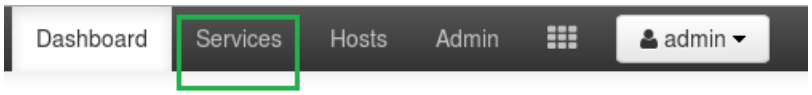
The default user id and password on the QSE image is **virtuser/password**. If you are using the cloud environment, you will use your personal id.

One thing to note: sometimes after you run a command (either Scala or Python), the terminal may seem like got stuck. When that happens, just hit the Enter key to get it back to the prompt.

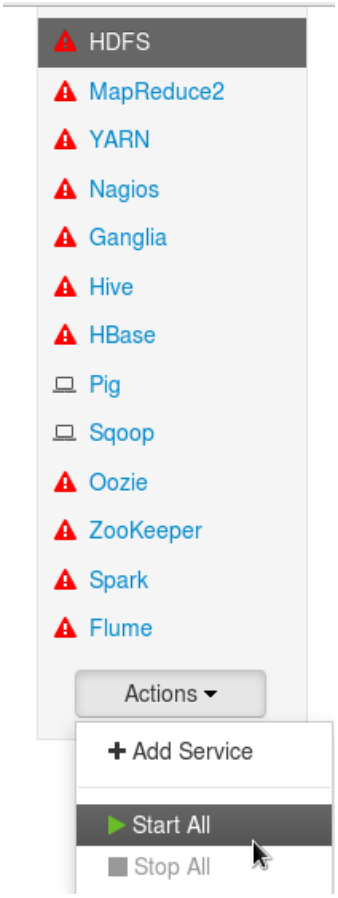
__1. Once you have the VM started and you log in to the OS, you must start up the services. The Firefox browser should start up automatically, if not, open it up and point the browser to: rvm.svl.ibm.com:8080

__2. Log in to the Ambari console using the default user id and password**: admin/admin**

__3. Once logged in, you will be brought to the dashboard. Click on the Services link next to Dashboard:



__4. On the left side of the window, near the bottom, Click Actions and Start All to start up the services:



It may take a while for all the services to start up. Once it does, you may proceed with the rest of the lab.

Troubleshooting:

Browser cannot load the Ambari user interface

If you have problems with the browser not being able to load the Ambari user interface, then the IP address of the machine might have changed due to a network connectivity reset. Network reset can happen during these conditions:

- Changing from wifi to wired or wired to wifi.
- Shutting down the system, and then re-establishing connection.
- Sending the machine into sleep mode. When it comes back up, the network IP address might be reset.

To resolve this issue, run the following scripts as the **`root`** user (root/password):

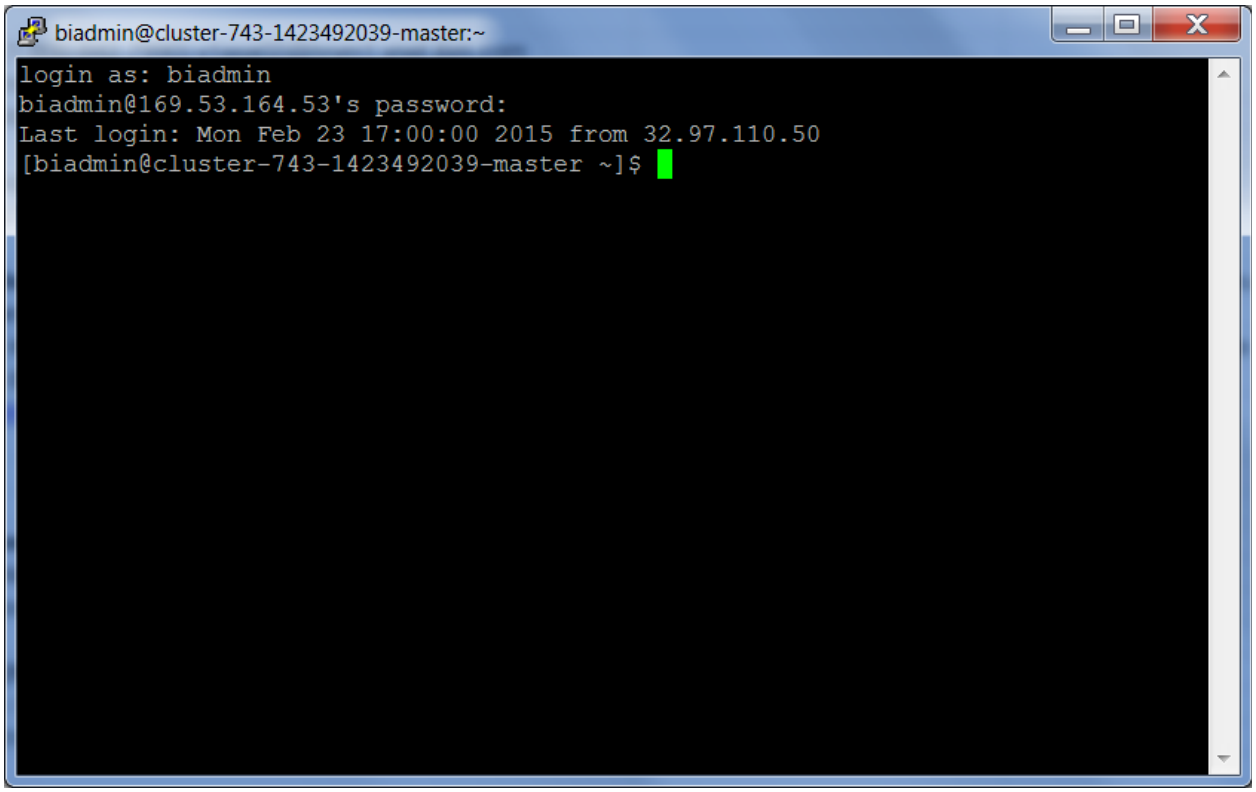
```
/home/virtuser/setHostname.sh
/home/virtuser/start-all.sh
```

Then, try to connect to `http://rvm.svl.ibm.com:8080`. You can also reboot the VM so that the IP address changes are updated as part of the initialization scripts.

1.2 Starting up the Spark Shell

Spark's shell provides a simple way to learn the APIs. In this section, you will get to use both Scala and Python to work with Spark's API. The first half will cover the shell using Scala. The second half will show the same steps, but using Python instead. To use the Spark shell, you will need to SSH to the cluster. Use your favorite SSH tool to connect to the cluster. This lab exercise will use PuTTY.

__5. Launch a new terminal.



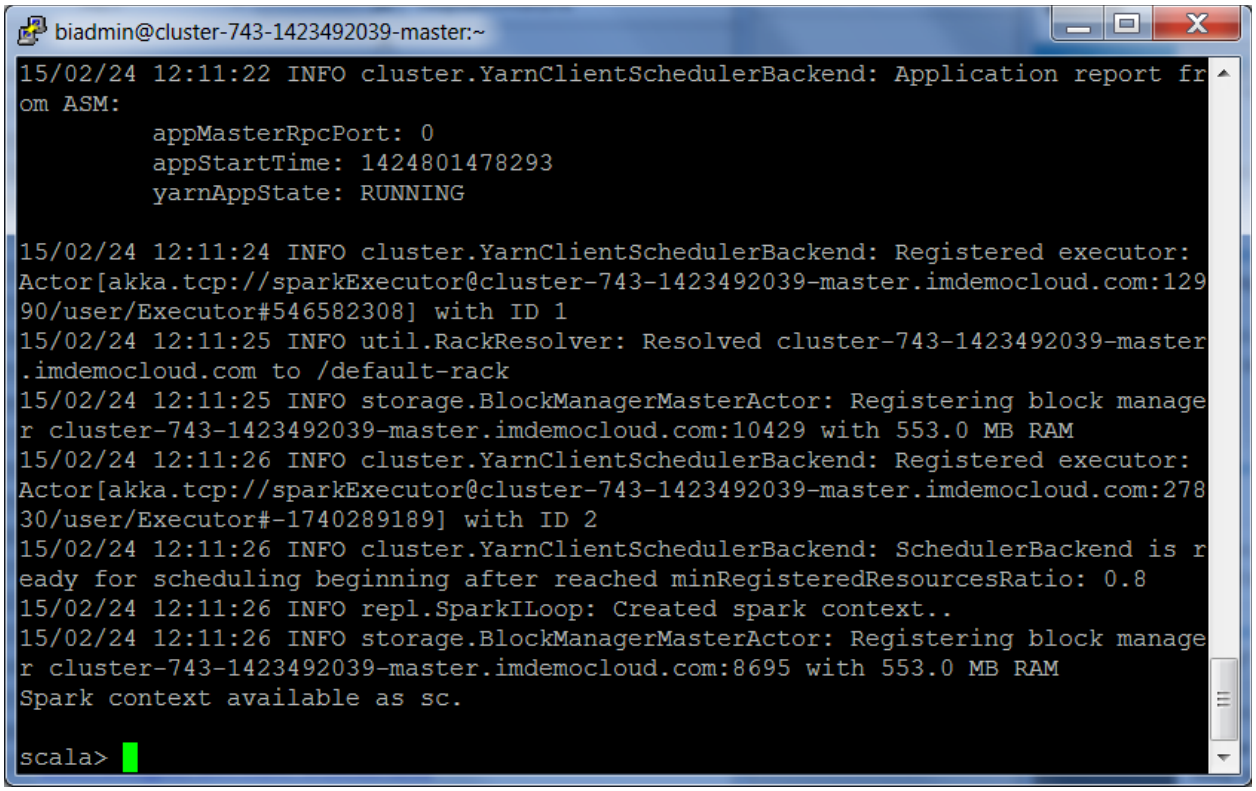
1.2.1 Using Scala

__1. Copy the README.md file from the *labfiles* directory into the /tmp directory on the HDFS. If you don't have the *labfiles* directory, go back to the BDU page and download and set up those files. In the terminal window, execute:

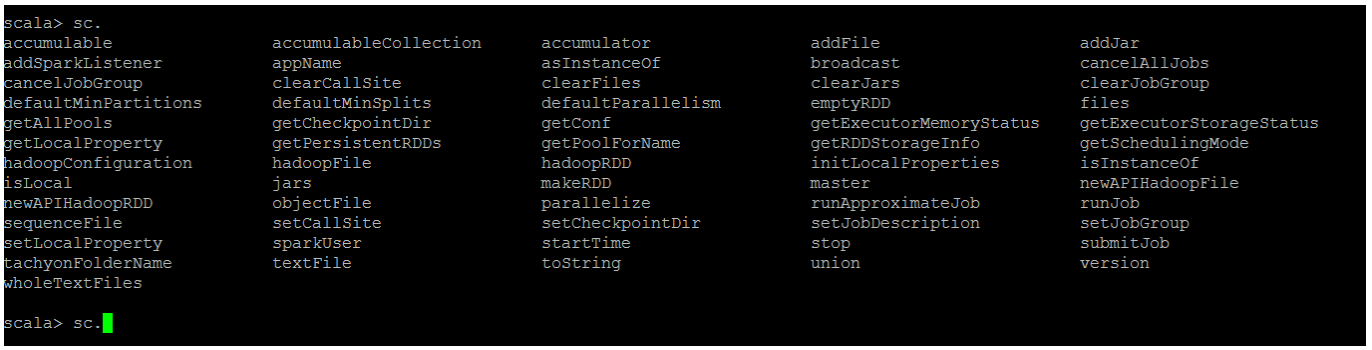
hadoop fs -copyFromLocal /home/virtuser/labfiles/README.md /tmp

__2. Start the Spark shell with this command:

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



__3. The shell provides code assist. For example, type in "sc." followed by the Tab key to get the list of options associated with the spark context:

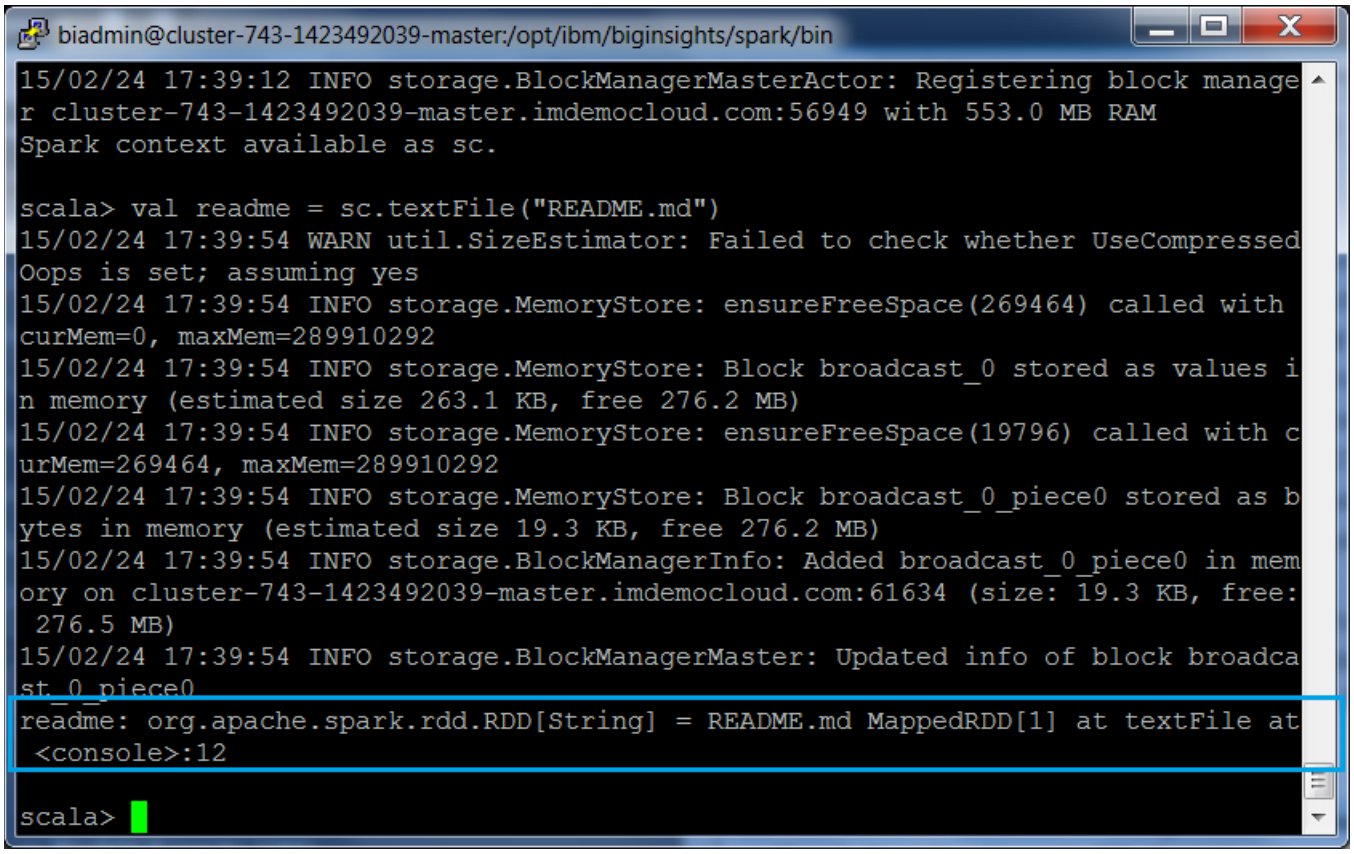


__4. Note the path of the readme text file is on the HDFS. Type in:

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

The parameter of the textFile method is the location within the /tmp directory on the HDFS.

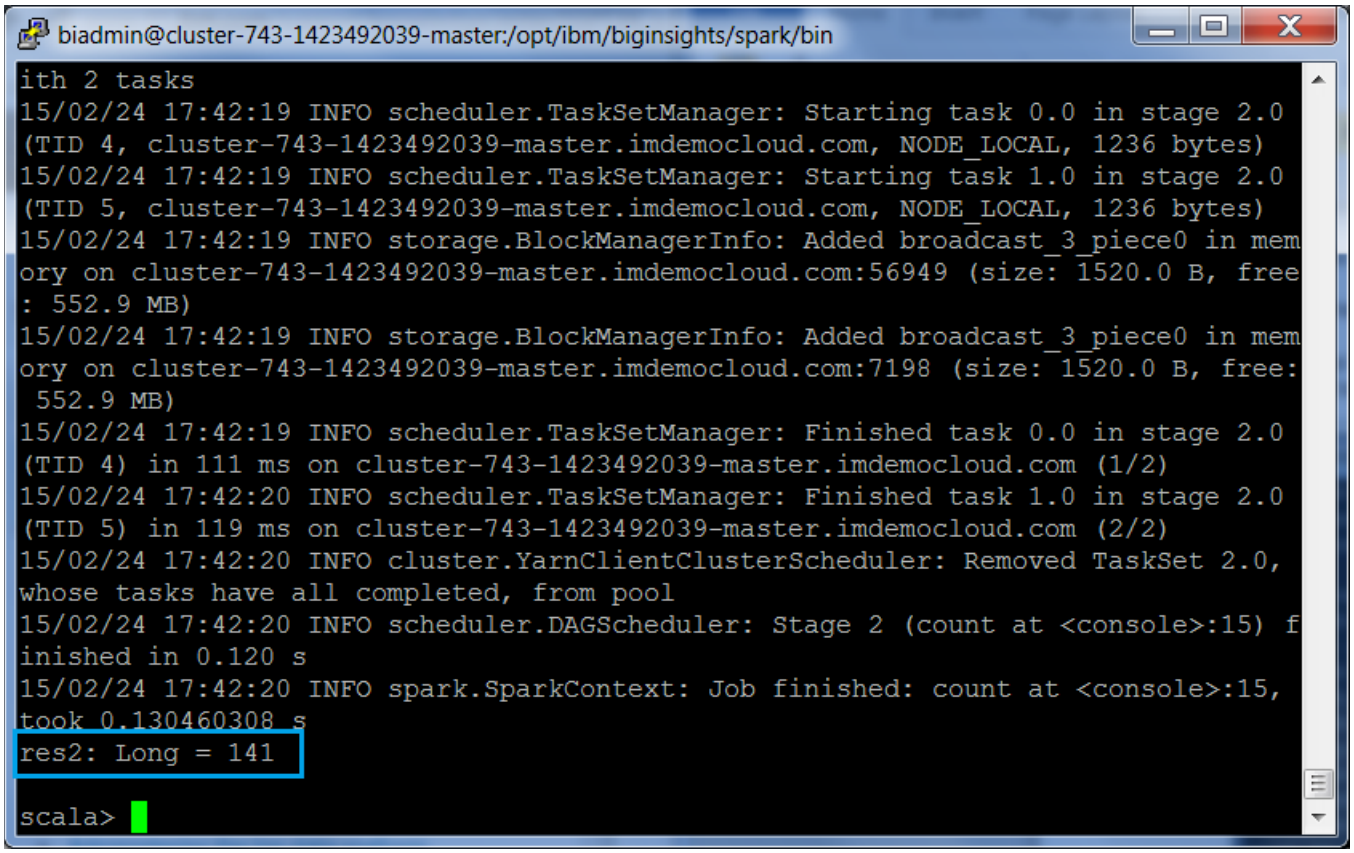
This was a RDD transformation, thus it return a pointer to a RDD, which we have named as *readme*. You can see from the returned line that the readme is a pointer to the RDD.



__5. Let's perform some RDD actions on this text file. Count the number of items in the RDD using this command:

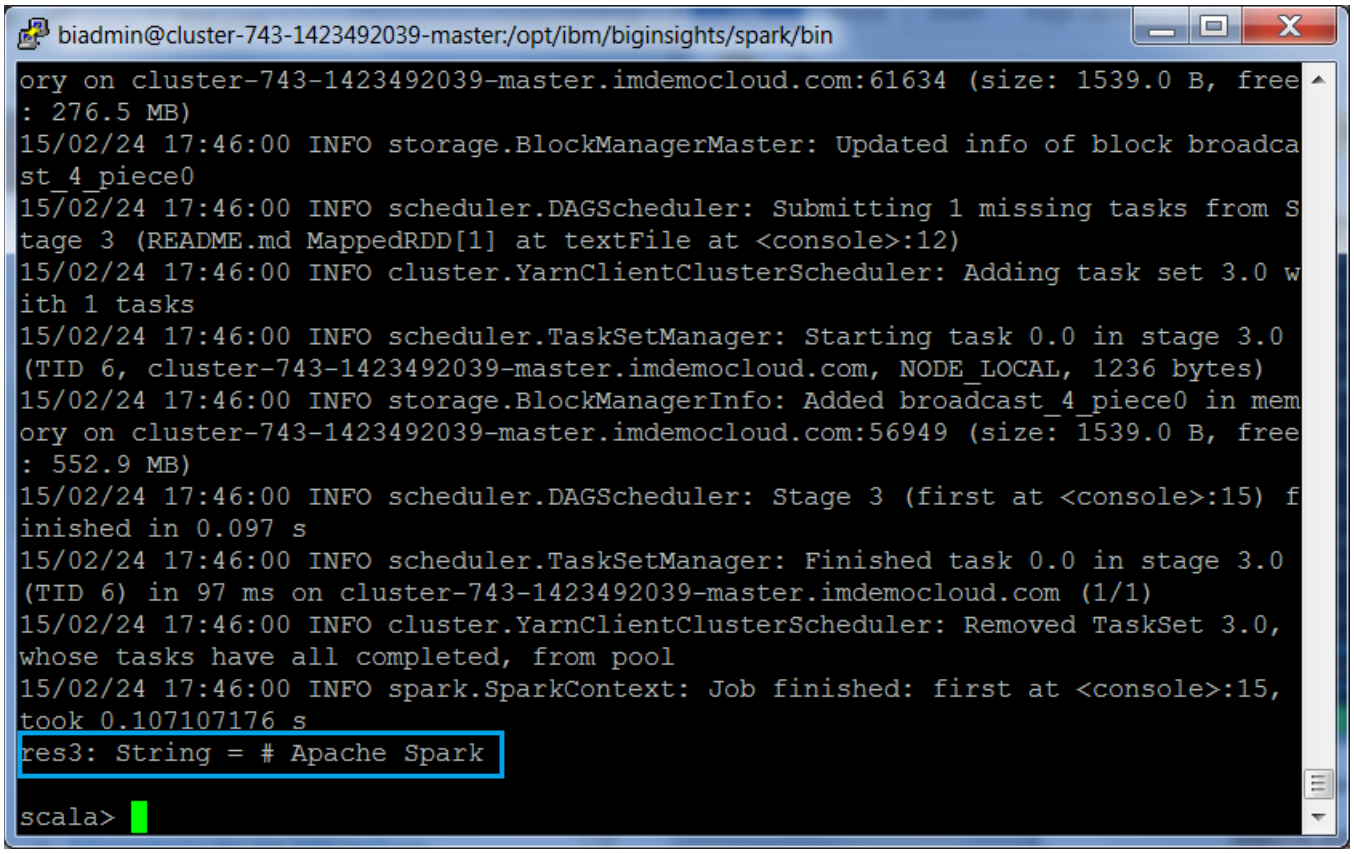
```
readme.count()
```

You should see that this RDD action returned a value of 141.



__6. Let's run another action. Run this command to find the first item in the RDD:

```
readme.first()
```



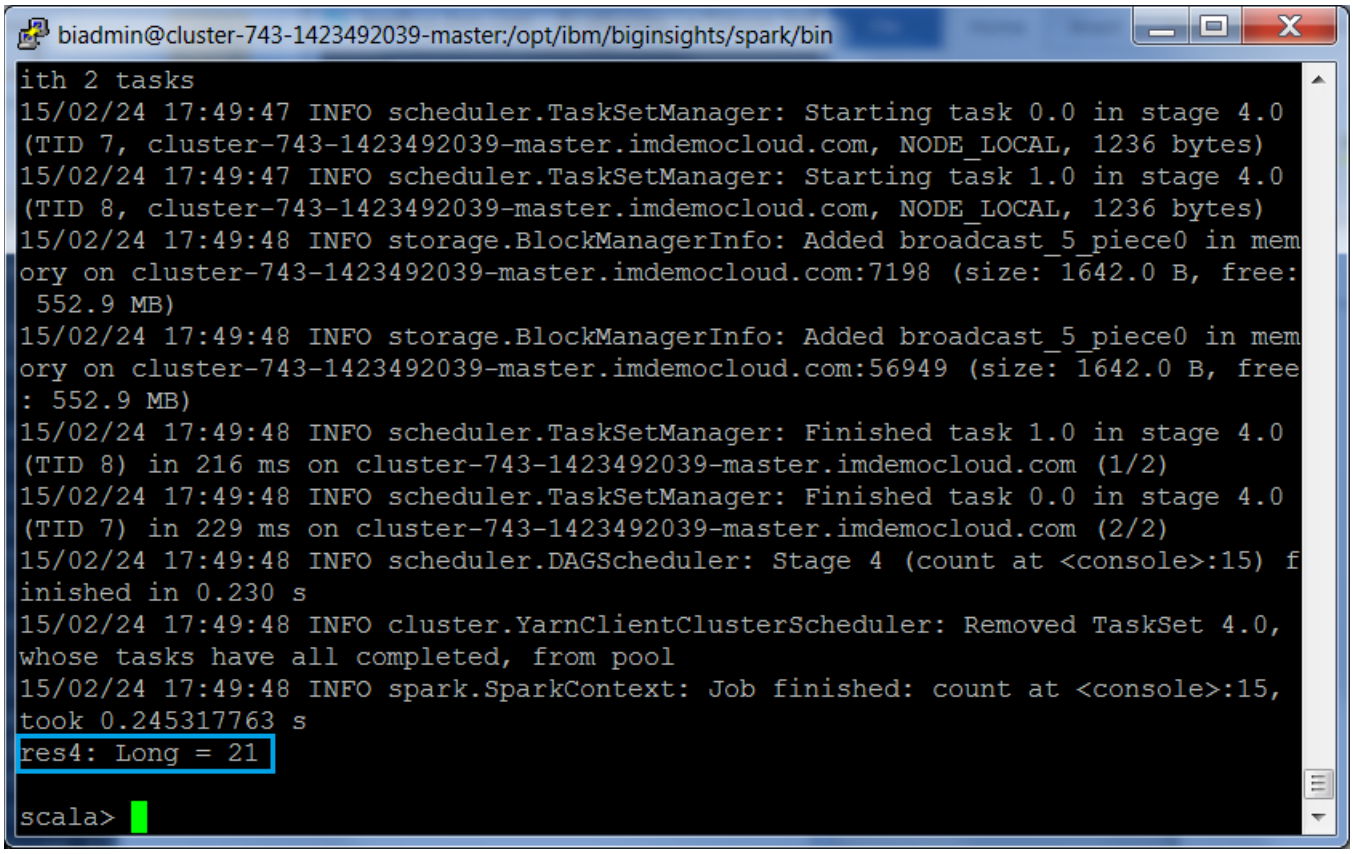
__7. Now let's try a transformation. Use the *filter* transformation to return a new RDD with a subset of the items in the file. Type in this command:

```
val linesWithSpark = readme.filter(line => line.contains("Spark"))
```

Again, this returned a pointer to a RDD with the results of the filter transformation.

__8. You can even chain together transformations and actions. To find out how many lines contains the word "Spark", type in:

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



__9. Do not close the shell. The rest of the exercises will build upon this one. Open up a new terminal and do the Python exercises on that shell.

1.2.2 Using Python

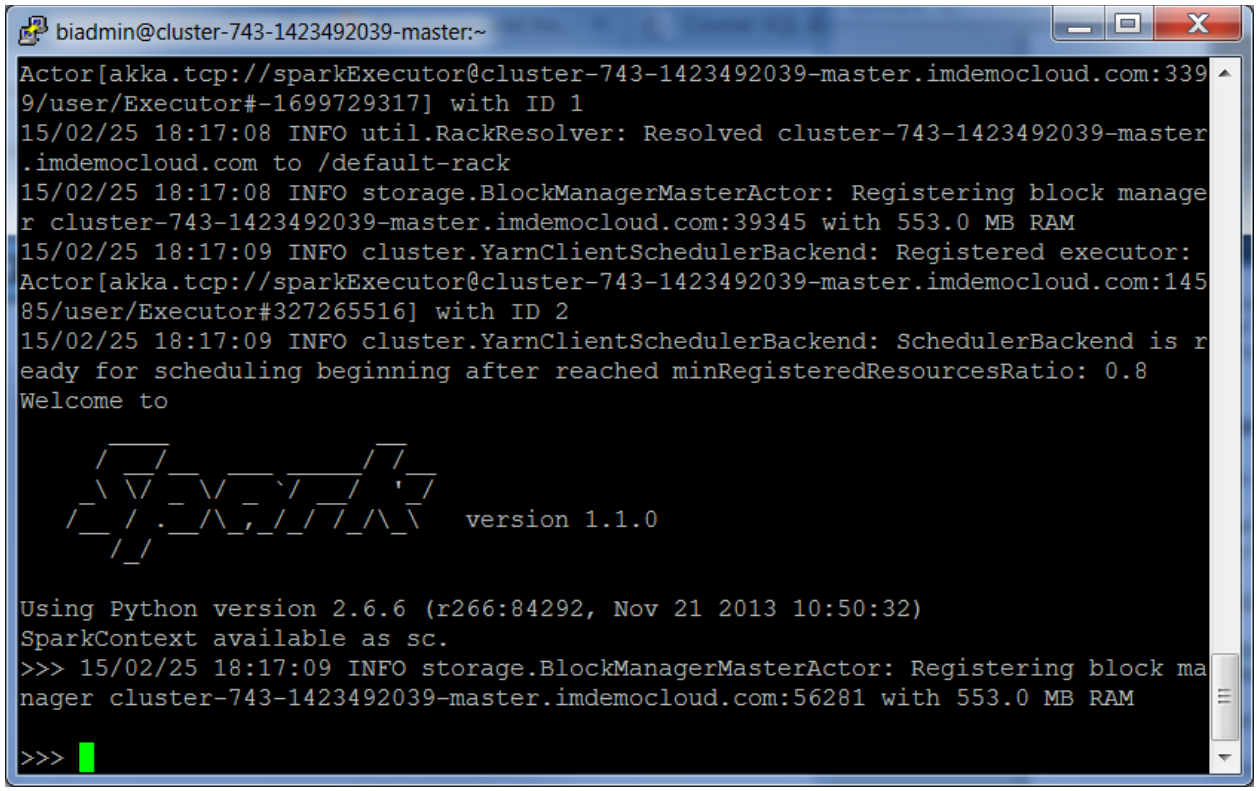
This section goes over the same tasks as the Scala section, but using the Python shell.

__1. Load the README file from the local system onto HDFS. If you had done this already, you can skip this step. Execute this command:

```
hadoop fs -put /home/virtuser/README.md /tmp
```

__2. Start the Python Spark shell with this command:

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



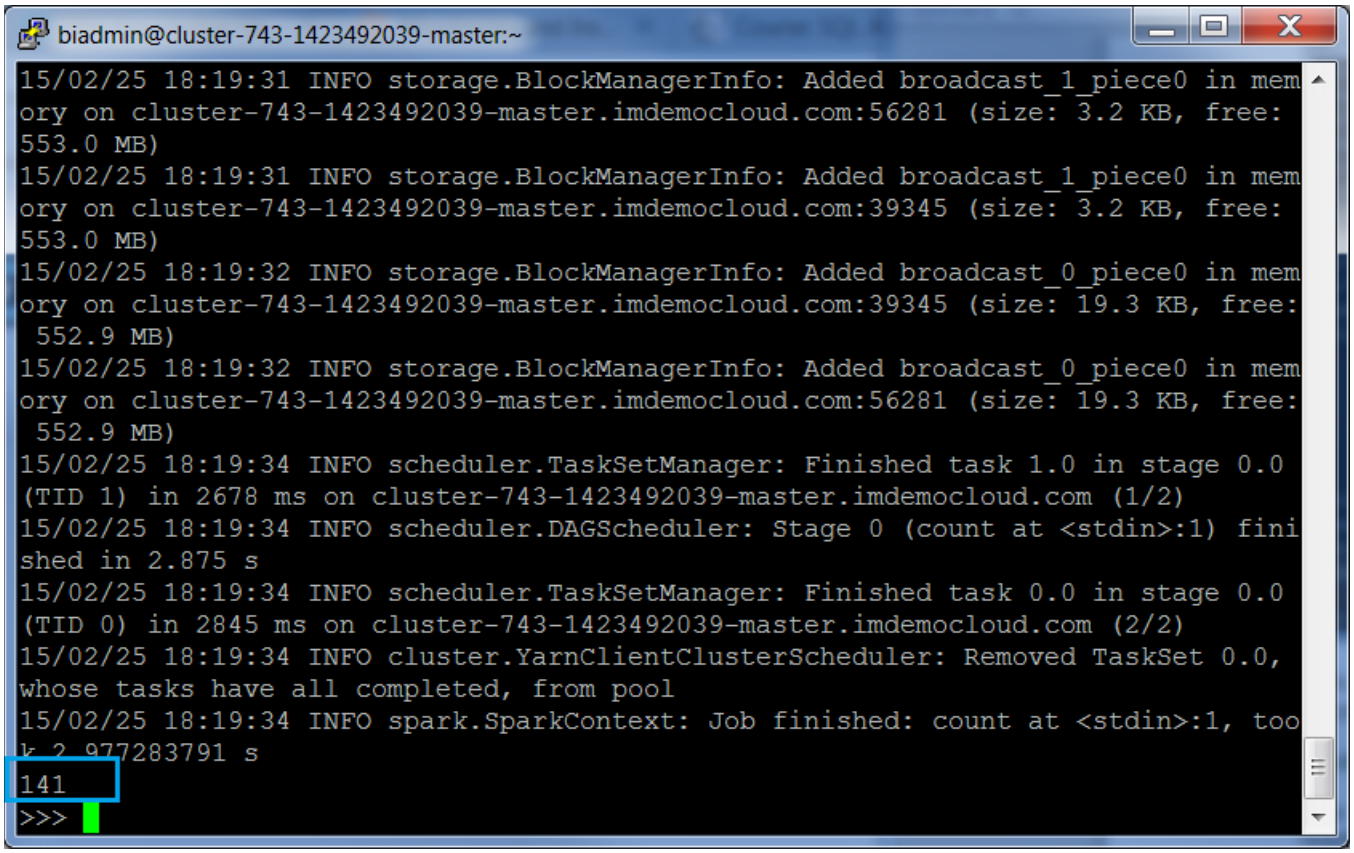
__3. Let's start with the example of reading in a text file, and thus converting it to a RDD. Note that the path of the readme text file is on the HDFS. Type in:

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

This was a RDD transformation, thus it return a pointer to a RDD, which we have named as *readme*.

__4. Let's perform some RDD actions on this text file. Count the number of items in the RDD using this command:

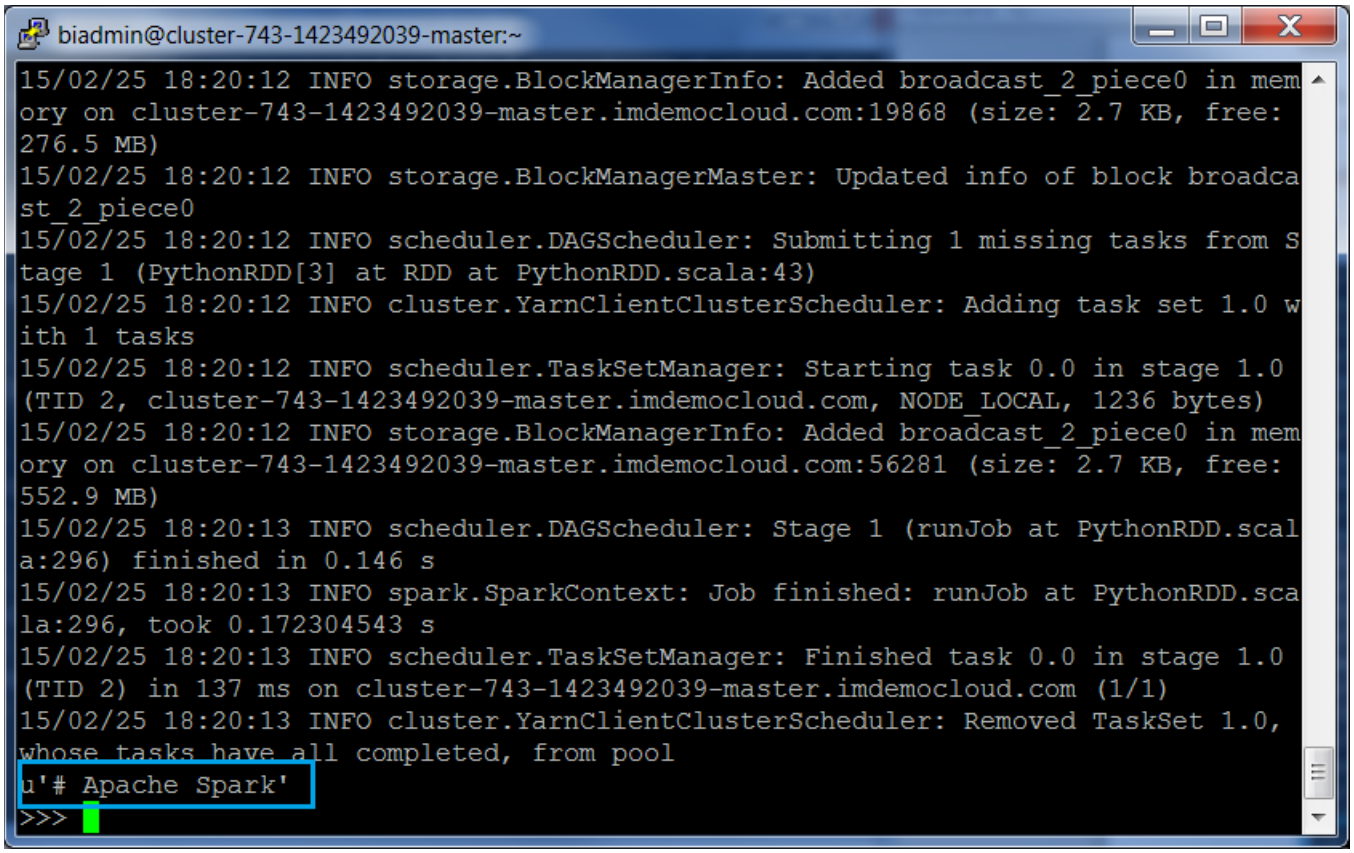
```
readme.count()
```



You see that this RDD action returned a value of 141.

__5. Let's run another action. Run this command to find the first item in the RDD:

```
readme.first()
```

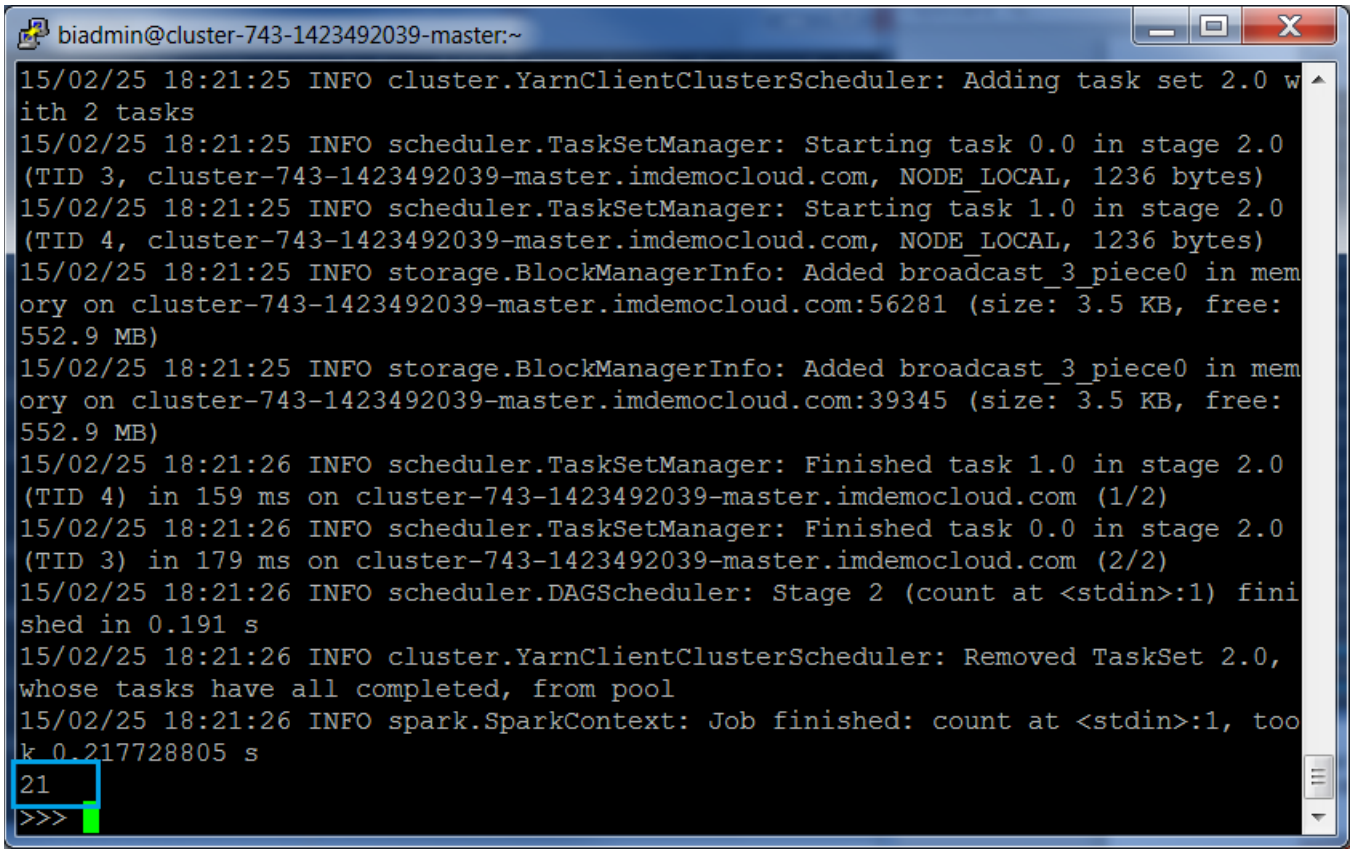


__6. Now let's try a transformation. Use the *filter* transformation to return a new RDD with a subset of the items in the file. Type in this command:

```
linesWithSpark = readme.filter(lambda line: "Spark" in line)
```

__7. You can even chain together transformations and actions. To find out how many lines contains the word "Spark", type in:

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


biadmin@cluster-743-1423492039-master:~
15/02/25 18:21:25 INFO cluster.YarnClientClusterScheduler: Adding task set 2.0 with 2 tasks
15/02/25 18:21:25 INFO scheduler.TaskSetManager: Starting task 0.0 in stage 2.0 (TID 3, cluster-743-1423492039-master.imdemocloud.com, NODE_LOCAL, 1236 bytes)
15/02/25 18:21:25 INFO scheduler.TaskSetManager: Starting task 1.0 in stage 2.0 (TID 4, cluster-743-1423492039-master.imdemocloud.com, NODE_LOCAL, 1236 bytes)
15/02/25 18:21:25 INFO storage.BlockManagerInfo: Added broadcast_3_piece0 in memory on cluster-743-1423492039-master.imdemocloud.com:56281 (size: 3.5 KB, free: 552.9 MB)
15/02/25 18:21:25 INFO storage.BlockManagerInfo: Added broadcast_3_piece0 in memory on cluster-743-1423492039-master.imdemocloud.com:39345 (size: 3.5 KB, free: 552.9 MB)
15/02/25 18:21:26 INFO scheduler.TaskSetManager: Finished task 1.0 in stage 2.0 (TID 4) in 159 ms on cluster-743-1423492039-master.imdemocloud.com (1/2)
15/02/25 18:21:26 INFO scheduler.TaskSetManager: Finished task 0.0 in stage 2.0 (TID 3) in 179 ms on cluster-743-1423492039-master.imdemocloud.com (2/2)
15/02/25 18:21:26 INFO scheduler.DAGScheduler: Stage 2 (count at <stdin>:1) finished in 0.191 s
15/02/25 18:21:26 INFO cluster.YarnClientClusterScheduler: Removed TaskSet 2.0, whose tasks have all completed, from pool
15/02/25 18:21:26 INFO spark.SparkContext: Job finished: count at <stdin>:1, took 0.217728805 s
21
>>>

1.3 More on RDD Operations

This section builds upon the previous section. You will have to use the same shell from the previous section (either Scala or Python). In this section, you will see that RDD can be used for more complex computations. You will find the line from that readme file with the most words in it.

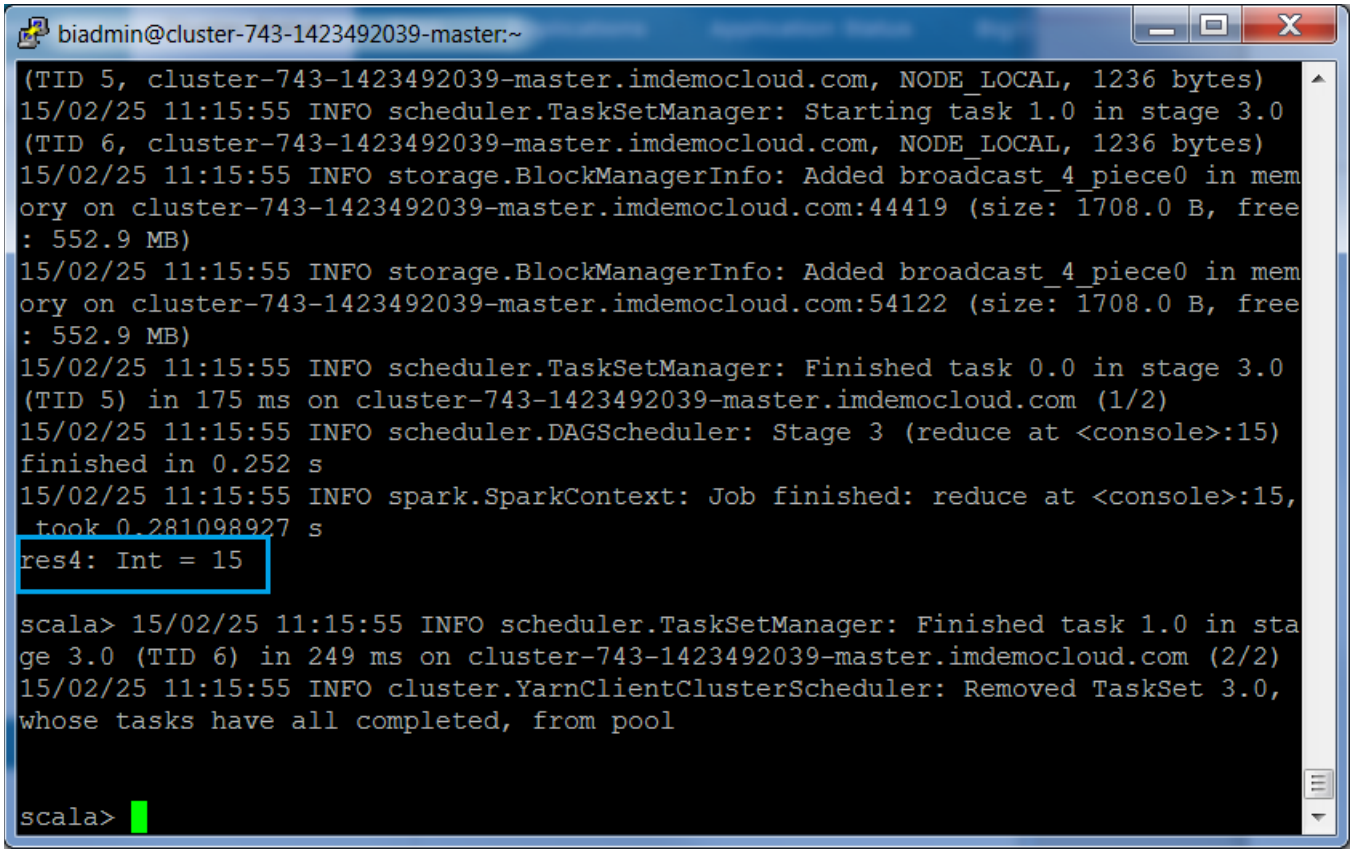
Similar to how the last section was structured, Scala will be shown first, followed by the same steps shown for Python.

1.3.1 With Scala

__1. Using the scala shell, copy and paste:

```
readme.map(line => line.split(" ").size).reduce((a, b) => if (a > b) a else b)
```

There are two parts to this. The first maps a line to an integer value, the number of words in that line. In the second part reduce is called to find the line with the most words in it. The arguments to *map* and *reduce* are Scala function literals (closures), but you can use any language feature or Scala/Java library.

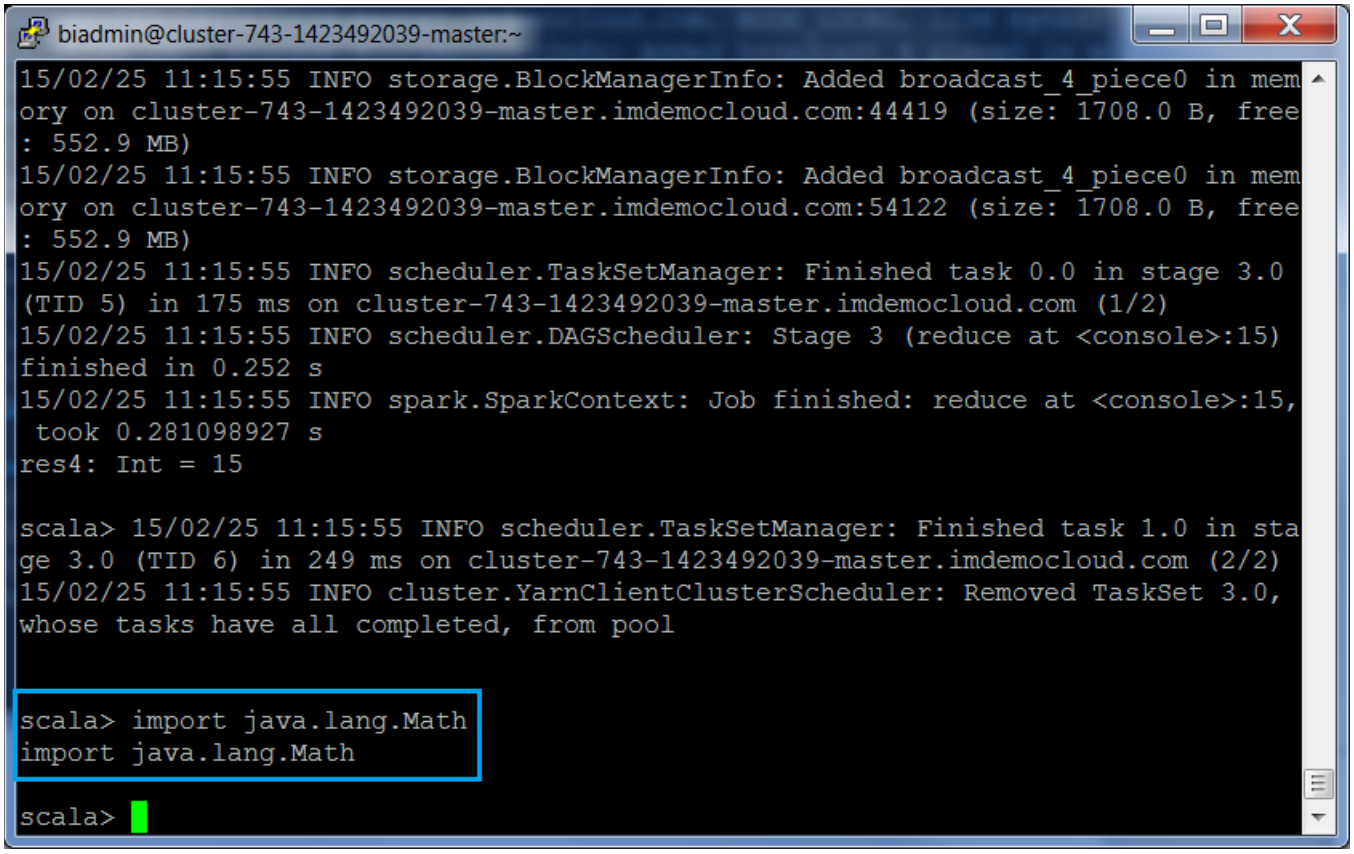


Line 15 contains the most words in it.

In the next step, you use the Math.max() function to show that you can indeed use a Java library instead.

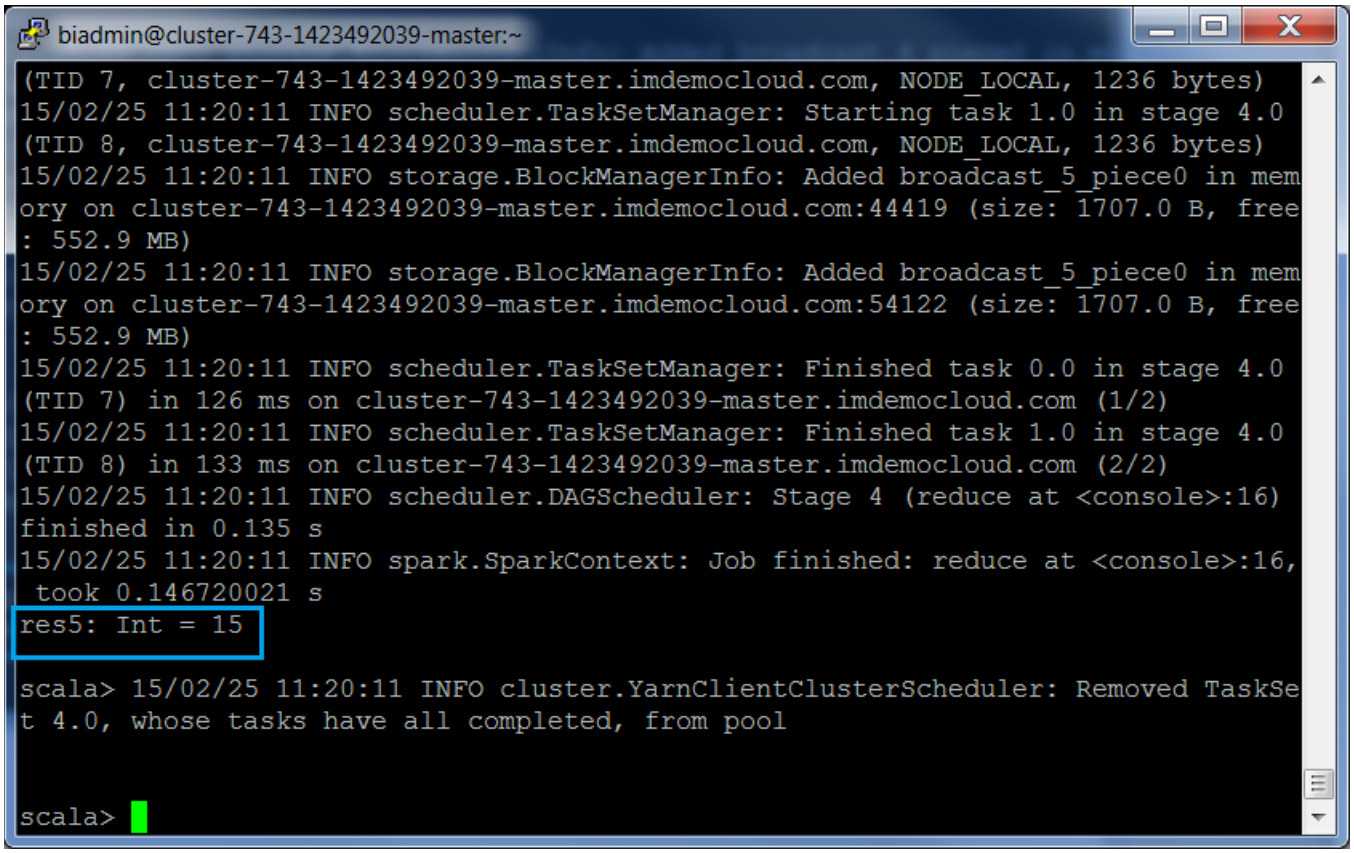
__2. Import in the java.lang.Math library. Copy/paste:

```
import java.lang.Math
```



__3. Now copy and paste in the following to run with the max function:

```
readme.map(line => line.split(" ").size).reduce((a, b) => Math.max(a, b))
```



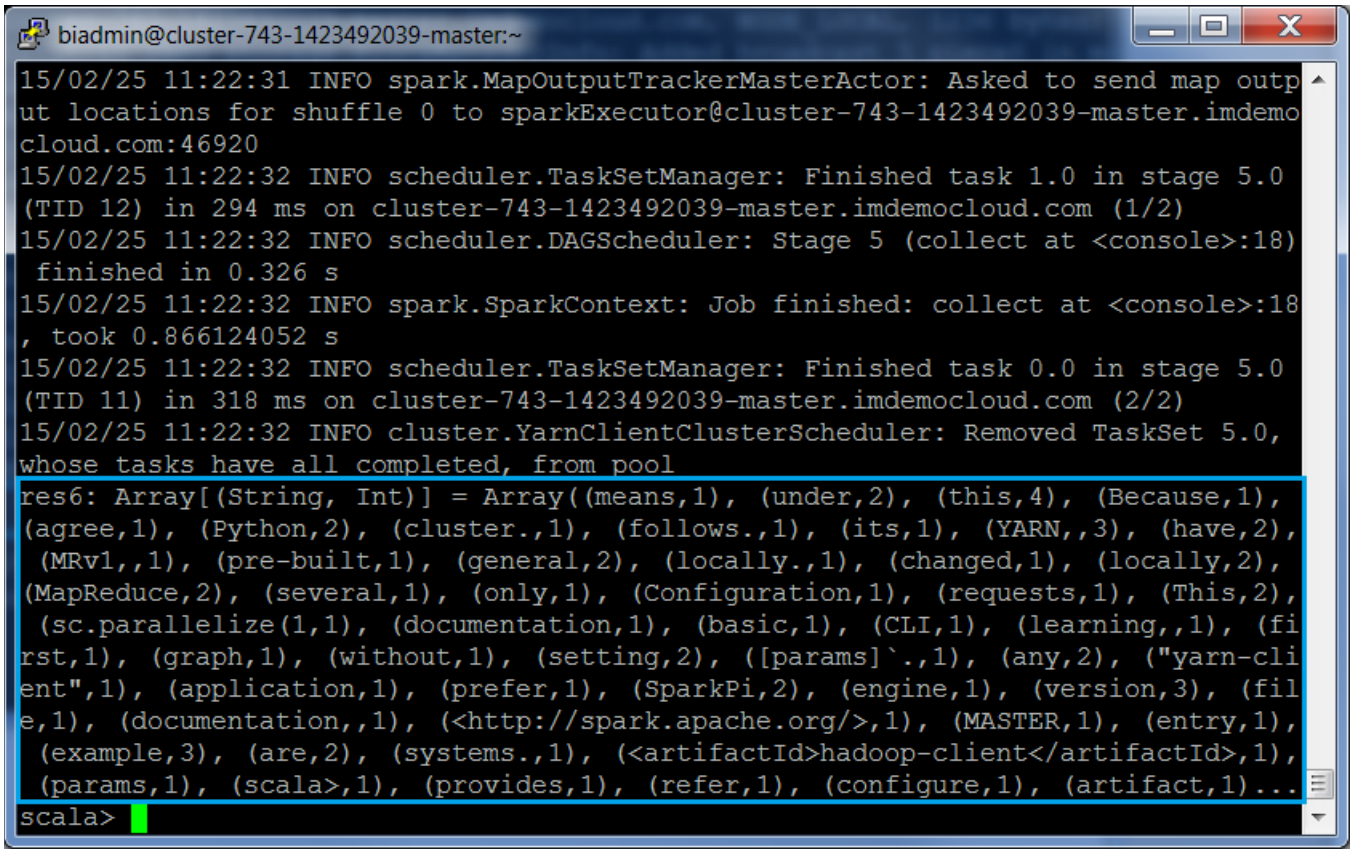
__4. Spark has a MapReduce data flow pattern. We can use this to do a word count on the readme file. Copy and paste in:

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

Here we combined the *flatMap*, *map*, and the *reduceByKey* functions to do a word count of each word in the readme file.

__5. To collect the word counts, use the *collect* action.

```
wordCounts.collect()
```



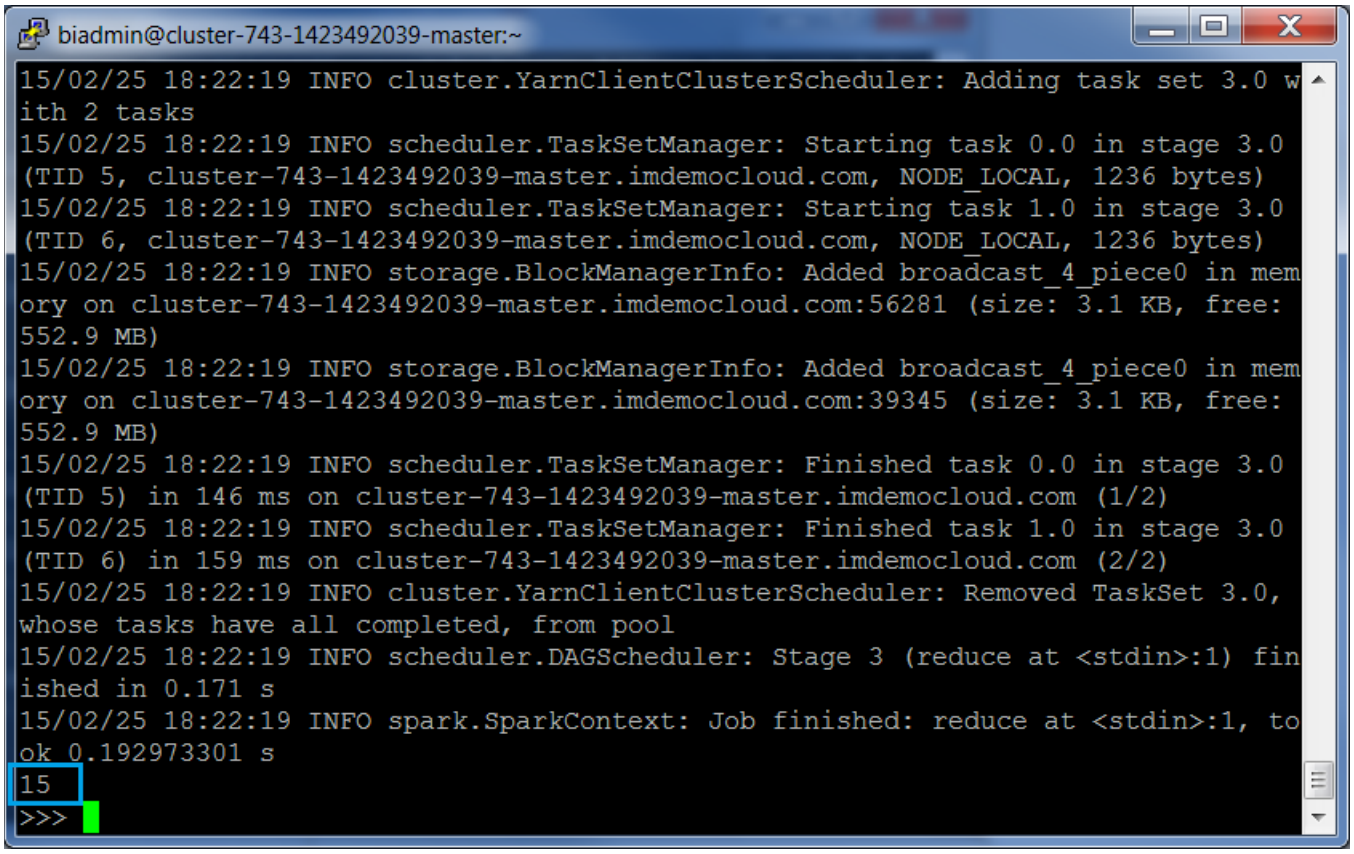
You can see the partial results using the collect action.

1.3.2 With Python

__6. Using the python shell, copy in:

```
readme.map(lambda line: len(line.split())).reduce(lambda a, b: a if (a > b)
else b)
```

There are two parts to this. The first maps a line to an integer value, the number of words in that line. In the second part reduce is called to find the line with the most words in it. The arguments to *map* and *reduce* are Python anonymous functions (lambdas), but you can use any top level Python functions. In the next step, you'll define a max function to illustrate this feature.

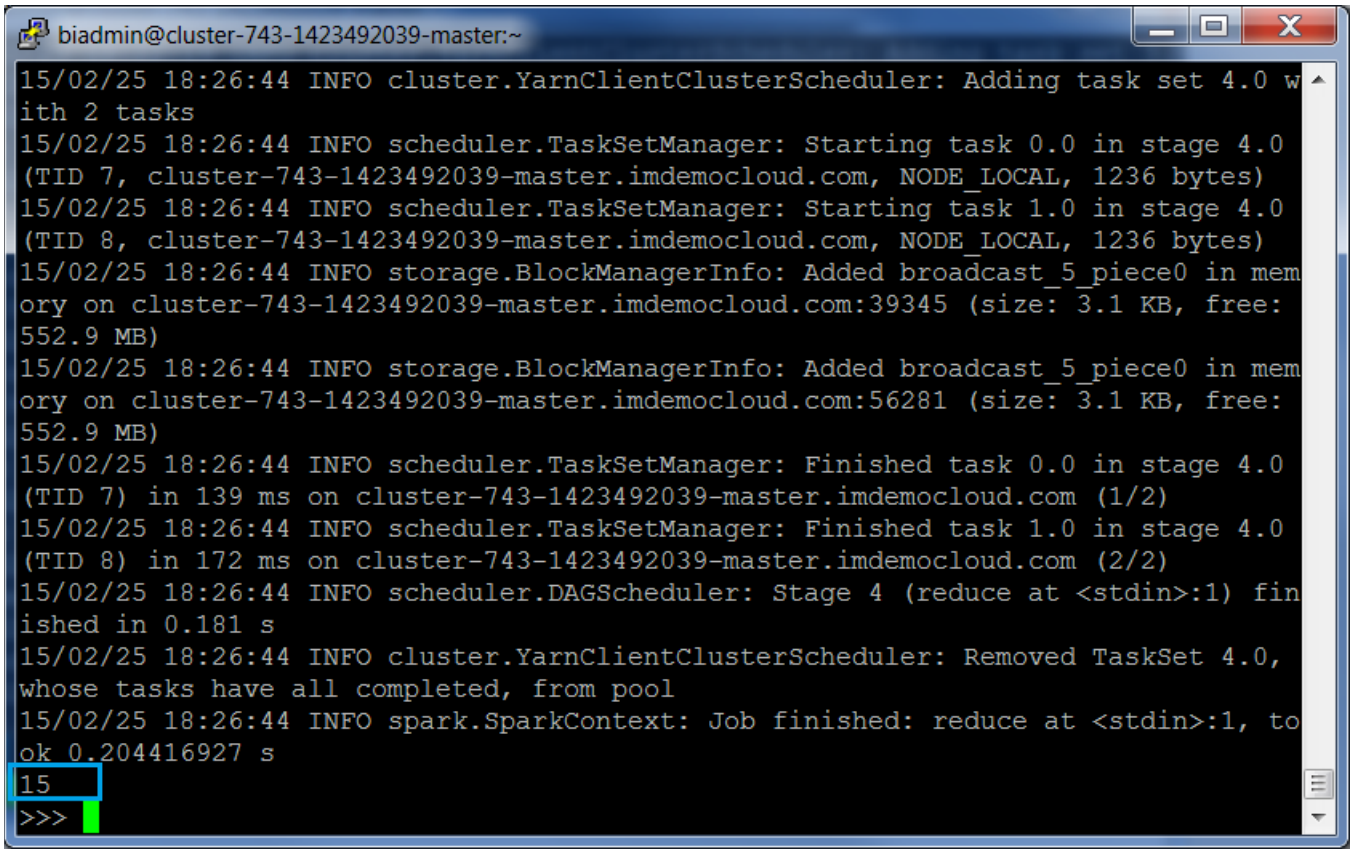


__7. Define the max function. You will need to type this in:

```
def max(a, b):
...     if a > b:
...         return a
...     else:
...         return b
...
```

__8. Now copy in the following to run with the max function:

```
readme.map(lambda line: len(line.split())).reduce(max)
```



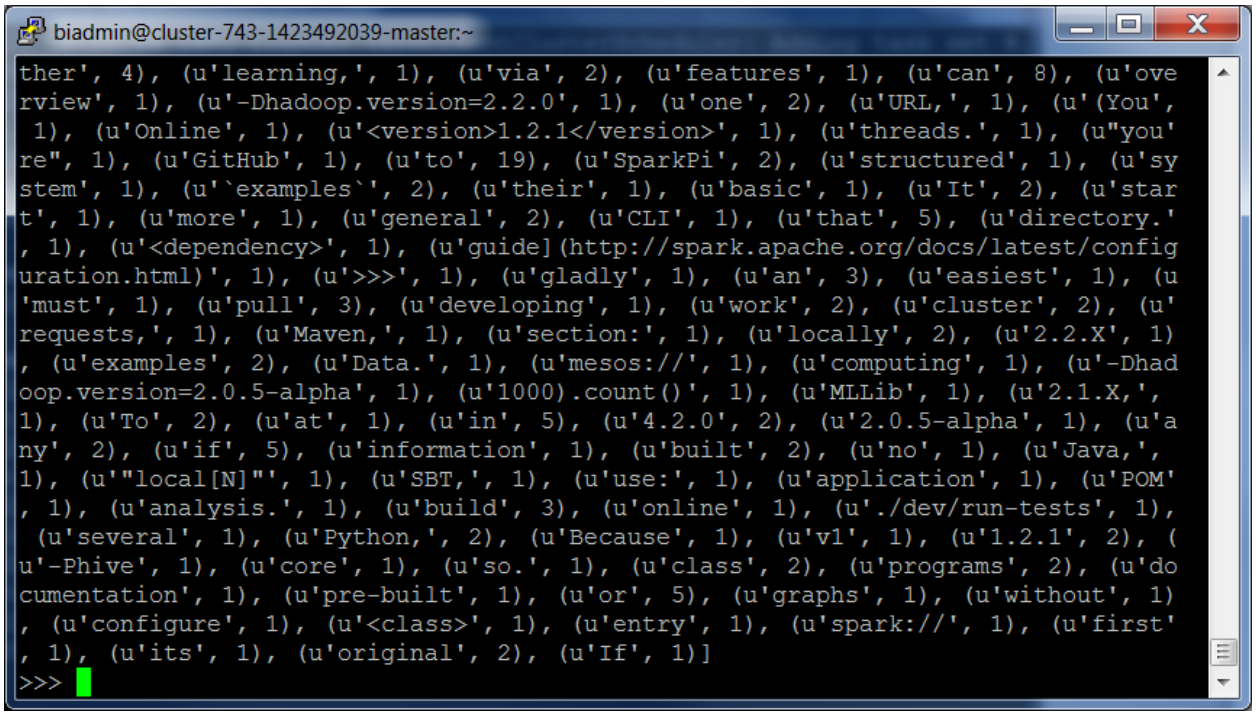
__9. Spark has a MapReduce data flow pattern. We can use this to do a word count on the readme file. Copy in:

```
wordCounts = readme.flatMap(lambda line: line.split()).map(lambda word: (word, 1)).reduceByKey(lambda a, b: a+b)
```

Here we combined the *flatMap*, *map*, and the *reduceByKey* functions to do a word count of each word in the readme file.

__10. To collect the word counts, use the *collect* action.

```
wordCounts.collect()
```


biadmin@cluster-743-1423492039-master:~
ther', 4), (u'learning,', 1), (u'via', 2), (u'features', 1), (u'can', 8), (u'ove
rview', 1), (u'-Dhadoop.version=2.2.0', 1), (u'one', 2), (u'URL,', 1), (u'(You',
1), (u'Online', 1), (u'<version>1.2.1</version>', 1), (u'threads.', 1), (u"you'
re", 1), (u'GitHub', 1), (u'to', 19), (u'SparkPi', 2), (u'structured', 1), (u'sy
stem', 1), (u'`examples`', 2), (u'their', 1), (u'basic', 1), (u'It', 2), (u'star
t', 1), (u'more', 1), (u'general', 2), (u'CLI', 1), (u'that', 5), (u'directory.'
, 1), (u'<dependency>', 1), (u'guide](http://spark.apache.org/docs/latest/config
uration.html)', 1), (u'>>>', 1), (u'gladly', 1), (u'an', 3), (u'easiest', 1), (u
'must', 1), (u'pull', 3), (u'developing', 1), (u'work', 2), (u'cluster', 2), (u'
requests,', 1), (u'Maven,', 1), (u'section:', 1), (u'locally', 2), (u'2.2.X', 1)
, (u'examples', 2), (u'Data.', 1), (u'mesos://', 1), (u'computing', 1), (u'-Dhad
oop.version=2.0.5-alpha', 1), (u'1000).count()', 1), (u'MLLib', 1), (u'2.1.X,',
1), (u'To', 2), (u'at', 1), (u'in', 5), (u'4.2.0', 2), (u'2.0.5-alpha', 1), (u'a
ny', 2), (u'if', 5), (u'information', 1), (u'built', 2), (u'no', 1), (u'Java,',
1), (u'"local[N]"', 1), (u'SBT,', 1), (u'use:', 1), (u'application', 1), (u'POM'
, 1), (u'analysis.', 1), (u'build', 3), (u'online', 1), (u'./dev/run-tests', 1),
(u'several', 1), (u'Python,', 2), (u'Because', 1), (u'v1', 1), (u'1.2.1', 2), (
u'-Phive', 1), (u'core', 1), (u'so.', 1), (u'class', 2), (u'programs', 2), (u'do
cumentation', 1), (u'pre-built', 1), (u'or', 5), (u'graphs', 1), (u'without', 1)
, (u'configure', 1), (u'<class>', 1), (u'entry', 1), (u'spark://', 1), (u'first'
, 1), (u'its', 1), (u'original', 2), (u'If', 1)]
>>>

1.4 Using Spark caching

In this short section, you'll see how Spark caching can be used to pull data sets into a cluster-wide in-memory cache. This is very useful for accessing repeated data, such as querying a small "hot" dataset or when running an iterative algorithm. Both Python and Scala use the same commands, so you can input these into any of the two shells.

__1. As a simple example, let's mark our *linesWithSpark* dataset to be cached and then invoke the first count operation to tell Spark to cache it. Remember that transformation operations such as cache does not get processed until some action like count() is called. Once you run the second count() operation, you should notice a small increase in speed.

```
linesWithSpark.cache()

linesWithSpark.count()

linesWithSpark.count()
```

It may seem silly to cache such a small file, but for larger data sets across tens or hundreds of nodes, this would still work. The second *linesWithSpark.count()* action runs against the cache and would perform significantly better for large datasets.

__2. At this time, you can terminate the shells. Use the key combination: CTRL + D

Summary

Having completed this exercise, you should now be able to log in to your environment and use the Spark shell to run simple actions and transformations for Scala and/or Python. You understand that Spark caching can be used to cache large datasets and subsequent operations on it will utilize the data in the cache rather than re-fetching it from HDFS.

NOTES

NOTES

IBM Software