

# SPARK LAB 1 – ASSIGNMENT (SUBMISSION)

**By:**  
**Alvaro Rueda**  
**Kamal Nandan**  
**Rahul Singh**

# Lab 1

## 1. Setup and Requirements

The necessary setup was done to set up clusters and launch Python in Spark.

The screenshot displays the Google Cloud Platform interface. The top navigation bar shows 'Google Cloud Platform' and 'spark-mbd-term2'. The left sidebar has 'Cloud Dataproc' selected. The main content area shows the 'Clusters' page with a table of clusters. One cluster, 'cluster-2349', is listed with details: Region (europe-west1), Zone (europe-west1-b), Total worker nodes (2), Cloud Storage staging bucket (dataproc-3539553e7-0f4e-4379-879e-b3a48276b444-europe-west1), Created (14 Feb 2018, 12:15:04), and Status (Running).

Below the cluster list, a terminal window shows the output of a command. The output includes the Debian GNU/Linux system's free software notice, the Spark version (2.2.0), and the Python version (2.7.9). It also shows a warning message about the Hive configuration file not being found.

```
Connected, host fdgspgpc1t1-88n-88n-2048-dataproc-3539553e7-0f4e-4379-879e-b3a48276b444-europe-west1
The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
rahu1@cluster-2349-m1:~$ pyspark
Python 2.7.9 (default, Jun 29 2016, 13:08:31)
[[GCC 4.9.2] on linux2
Type "help", "copyright", "credits" or "license()" for more information.
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
18/02/14 11:22:03 WARN org.apache.hadoop.hdfs.DataStreamer: Caught exception
java.lang.InterruptedException
    at java.lang.Object.wait(Native Method)
    at java.lang.Thread.join(Thread.java:1252)
    at java.lang.Thread.join(Thread.java:1326)
    at org.apache.hadoop.hdfs.DataStreamer.closeResponder(DataStreamer.java:973)
    at org.apache.hadoop.hdfs.DataStreamer.endBlock(DataStreamer.java:624)
    at org.apache.hadoop.hdfs.DataStreamer.run(DataStreamer.java:801)
18/02/14 11:22:03 WARN org.apache.hadoop.hdfs.DataStreamer: Caught exception
java.lang.InterruptedException
    at java.lang.Object.wait(Native Method)
    at java.lang.Thread.join(Thread.java:1252)
    at java.lang.Thread.join(Thread.java:1326)
    at org.apache.hadoop.hdfs.DataStreamer.closeResponder(DataStreamer.java:973)
    at org.apache.hadoop.hdfs.DataStreamer.endBlock(DataStreamer.java:624)
    at org.apache.hadoop.hdfs.DataStreamer.run(DataStreamer.java:801)
ivysettings.xml file not found in HIVE_HOME or HIVE_CONF_DIR, etc/hive/conf/dist/ivysettings.xml will be used
Welcome to
Spark version 2.2.0
Using Python version 2.7.9 (default, Jun 29 2016 13:08:31)
SparkSession available as 'spark'.
>>>
```

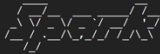
## 2. Execute different commands on RDDs

```

Connected, host fingerprint: ssh-rsa 2048 9A:AD:FA:CF:BD:BB:6A:76:55:5C:EA:39:5B:1E:3C:D4:E9:40:2B:9A

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/*copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
rahu1@cluster-2349-m1-8:~$ pyspark
Python 2.7.9 (default, Jun 29 2016, 13:08:31)
[GCC 4.9.2] on linux2
Type "help", "copyright", "credits" or "license()" for more information.
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
18/02/14 11:22:03 WARN org.apache.hadoop.hdfs.DataStreamer: Caught exception
java.lang.InterruptedException
    at java.lang.Object.wait(Native Method)
    at java.lang.Thread.join(Thread.java:1252)
    at java.lang.Thread.join(Thread.java:1326)
    at org.apache.hadoop.hdfs.DataStreamer.closeResponder(DataStreamer.java:973)
    at org.apache.hadoop.hdfs.DataStreamer.endBlock(DataStreamer.java:624)
    at org.apache.hadoop.hdfs.DataStreamer.run(DataStreamer.java:801)
18/02/14 11:22:03 WARN org.apache.hadoop.hdfs.DataStreamer: Caught exception
java.lang.InterruptedException
    at java.lang.Object.wait(Native Method)
    at java.lang.Thread.join(Thread.java:1252)
    at java.lang.Thread.join(Thread.java:1326)
    at org.apache.hadoop.hdfs.DataStreamer.closeResponder(DataStreamer.java:973)
    at org.apache.hadoop.hdfs.DataStreamer.endBlock(DataStreamer.java:624)
    at org.apache.hadoop.hdfs.DataStreamer.run(DataStreamer.java:801)
ivysettings.xml file not found in HIVE_HOME or HIVE_CONF_DIR, etc/hive/conf/dist/ivysettings.xml will be used
Welcome to

 version 2.2.0

Using Python version 2.7.9 (default, Jun 29 2016 13:08:31)
SparkSession available as 'spark'.
>>> data = [1,3,5,7,9]
>>> data
[1, 3, 5, 7, 9]
>>> myFirstRdd = sc.parallelize(data, 4)
>>> myFirstRdd.take(2)
[1, 3]
>>> myFirstRdd.collect()
[1, 3, 5, 7, 9]
>>> y=myFirstRdd.filter(lambda myFirstRdd: myFirstRdd % 3 == 0)
>>> y.collect()
[3, 9]
>>> mySecondRdd= sc.parallelize(range(7,12))
>>> myFirstRdd.union(mySecondRdd).distinct().collect()
[1, 7, 8, 9, 3, 10, 11, 5]
>>>

```

We create some data, in this case an array with 5 numbers:

```

>>> data = [1, 3, 5, 7, 9]
>>> data

```

What happens when we type “data”?

When we type “data”, the entire set of values inputted in “data” array by the user is displayed in python spark shell.

In order to get that array into an RDD we need to parallelize the data structure as an RDD:

```

>>> myFirstRdd = sc.parallelize(data, 4)
>>> myFirstRdd.take(2)

```

Explain what happens when use take(2)?

take(2) returns the first two values from the myFirstRdd if two elements are present in the rdd. In our case it was [1,3].

Notice that when we run sc.parrallelize, we create an RDD and we direct the driver node to break up the work into parallel tasks and each task would contain information about the split of the data that it will operate on; after this the Tasks are assigned to worker nodes.

What is the 4 doing when calling the parallelize method?

4 is the number of partitions for each CPU in our cluster.

And we can also get all the elements by using collect, although we will see during the course that this could cause some problems in real life apps.

```
>>> myFirstRdd.collect()  
[1, 3, 5, 7, 9]
```

We observed that the entire set of elements in myFirstRdd is displayed. In our case it was [1,3,5,7,9]

Why do you think collect might mean problems in real life applications?

In real life problems, the whole point of using big data is to divide the workload among clusters. If we call collect function and the size of the data is bigger than our cluster/computer/node, then the system may crash as the huge amount of information cannot be collected in a smaller object. Also, collect is an expensive operation because all the processed data is sent to the driver code.

Let's play with the different operations we have learnt in this lesson and explained in the slides and then attach the results to this document. These are the functions to use:

<b>filter</b> ( <i>func</i> )	Return a new dataset formed by selecting those elements of the source on which <i>func</i> returns true.
<b>distinct</b> ([ <i>numTasks</i> ])	Return a new dataset that contains the distinct elements of the source dataset.
<b>sample</b> ( <i>withReplacement</i> , <i>fraction</i> , <i>seed</i> )	Sample a fraction <i>fraction</i> of the data, with or without replacement, using a given random number generator seed.

Create an example which each function above and use this format to write the results

-----START OF RESULTS-----

(A)

**Operation:** filter(func)

**Example used:**

```
y=myFirstRdd.filter(lambda myFirstRdd: myFirstRdd % 3 ==0)
y.collect()
```

#filter function helps us select values in a dataframe, array,etc. which meet the specified condition. In our case we wanted the numbers which are divisible by 3 as output.

**Result:** [3,9]

(B)

**Operation:** distinct([num Tasks])

**Example used:**

```
mySecondRdd= sc.parallelize(range(7,12))
myFirstRdd.union(mySecondRdd).distinct().collect()
```

#distinct function along with the union function helps us select distinct values in both the joint array in our case (note that depending on the input this function can also be applied to other types of data such as a dataframe). First we created mySecondRdd which has a range of numbers from 7 to 11 (12 is not included). After performing a joint between myFirstRdd and mySecondRdd we can see the distinct values in the output given below.

**Result:** [1,7,8,9,3,10,11,5]

(C)

**Operation:** sample(withReplacement, fraction,seed)

**Example used:**

```
myFirstRdd.sample(True, .4).count()
```

#sample function returns a subset RDD of the input RDD. In our case the input RDD was myFirstRdd. We put the withReplacement as FALSE. withReplacement as TRUE allows the sample to be able to draw it multiple times. fraction represents the expected size of the sample as a fraction of RDD's size without replacement. This value must be between 0 and 1 (with 0 and 1 included) with replacement. Setting a seed helps to be able to generate the same random draws provided the input seed remains the same.

**Result:** 3

```

SyntaxError: Invalid syntax
>>> myFirstRdd.sample(True, .3).count()
2
>>> myFirstRdd.sample(True, .2).count()
0
>>> myFirstRdd.sample(True, .1).count()
2
>>> myFirstRdd.sample(True, .0).count()
0
>>> myFirstRdd.sample(True, .4).count()
3
>>>

```

-----END OF RESULTS-----

Now let's focus on map, execute the following lines in your REPL:

```

>>> data = [1, 3, 5, 7, 9]

>>> rdd = sc.parallelize(data, 4)

>>> rdd2 = rdd.map(lambda x : x + 1)

>>> rdd3 = rdd2.filter(lambda x : x % 2 == 0)

>>> rdd3.count()

```

Execute the operations above and attach results here. Explain what is happening.

-----START OF RESULTS-----

```

>>> data = [1, 3, 5, 7, 9]
>>> rdd = sc.parallelize(data, 4)
>>> rdd2 = rdd.map(lambda x : x + 1)
>>> rdd3 = rdd2.filter(lambda x : x % 2 == 0)
>>> rdd3.count()
5

```

-----END OF RESULTS-----

- We create a collection object that contains 5 elements
- Then we partition the data into 4 parts and assign it into "rdd"
- Then we write a transformation lambda function that will "map" each value in "rdd" by 1. The resultant rdd will be called rdd2
- Then, the contents of rdd2 will be passed through a filter lambda function. This function will filter out all the values that are not divisible by 2 and the resultant data would be stored in rdd3
- All of the above steps are transformation steps, but nothing will happen yet, unless some action is performed by calling count() or collect().
- Finally we see the count () function – which is an action. As soon as any action is invoked, all the transformation steps would be executed and the resultant data would be stored somewhere or dumped on to the screen. Please note that any of

the transformation(s) won't take place until some action is invoked - In our case it is the count() function which is the action

Let's work now with text instead of numbers and we are going to use some of the "set type" operations we can do on RDDs:

```
>>> rdd1 = sc.parallelize(['this', 'is', 'is', 'course',  
'course'])  
  
>>> rdd2 = sc.parallelize(['this', 'is', 'spark', 'master',  
'course'])  
  
>>> rdd1.distinct().collect()  
  
>>> rdd1.union(rdd2).collect()  
  
>>> rdd1.intersection(rdd2).collect()  
  
>>> rdd1.subtract(rdd2).collect()  
  
>>> rdd2.cartesian(rdd2).collect()
```

Execute the operations above and attach results here.

-----START OF RESULTS-----

```
>>> rdd1 = sc.parallelize(['this', 'is', 'is', 'course', 'course'])  
>>> rdd2 = sc.parallelize(['this', 'is', 'spark', 'master', 'course'])  
>>> rdd1.distinct().collect()  
['this', 'is', 'course']  
>>> rdd1.union(rdd2).collect()  
['this', 'is', 'is', 'course', 'course', 'this', 'is', 'spark', 'master', 'course']  
>>> rdd1.intersection(rdd2).collect()  
['this', 'course', 'is']  
>>>  
>>> rdd1.subtract(rdd2).collect()  
[]  
>>> rdd2.cartesian(rdd2).collect()  
[('this', 'this'), ('this', 'is'), ('is', 'this'), ('is', 'is'), ('this', 'spark'), ('this', 'master'), ('is', 'spark'),  
( 'is', 'master'), ('this', 'course'), ('is', 'course'), ('spark', 'this'), ('spark', 'is'), ('master', 'this'),  
( 'master', 'is'), ('course', 'this'), ('course', 'is'), ('spark', 'spark'), ('spark', 'master'),  
( 'master', 'spark'), ('master', 'master'), ('spark', 'course'), ('master', 'course'), ('course',  
'spark'), ('course', 'master'), ('course', 'course')]
```

-----END OF RESULTS-----

Let's work now with pair RDDs

```
>>> rdd = sc.parallelize([('a', 1), ('a', 1), ('b', 0), ('a', 1), ('b', 1), ('b', 1), ('a', 0), ('b', 0)], 3)

>>> rdd2 = rdd.reduceByKey(lambda accum, n : accum + n)

>>> rdd2.collect()

[('b', 2), ('a', 3)]
```

What happens when we execute this? Can you explain what is happening?

- First we create a pair RDD called "rdd"
- Then we call the `reduceByKey()`, a transformation operation, on "rdd".  
`reduceByKey()` works only on pair RDDs. `reduceByKey()` runs several parallel reduce operations, one for each key in the RDD, where each reduce operation combines values that have the same key. In this example above we are "reducing" the values for each key to one single value by adding up the values of all the pairs that have the same key. The summing up operation is performed by the lambda function which is simply adding the values of the pairs. The result of `reduceByKey()` is stored in another RDD called "rdd2" (Please note that If we try to run `reduceByKey()` on an RDD that is not paired, we will get an exception. Also, note that if the reduce operation needs to be performed on a no. of machines, in case the RDD is spread across more than one machine, in such a case, the reduce operation will be performed locally on each machine and then aggregated globally)
- Finally, we call `collect()` action which actually starts the execution of the



transformation and the result is printed onto the screen.

-----START OF RESULTS-----

```
>>> rdd = sc.parallelize([('a', 1), ('a', 1), ('b', 0), ('a', 1), ('b', 1), ('b', 1), ('a', 0), ('b', 0)], 3)
>>> rdd2 = rdd.reduceByKey(lambda accum, n : accum + n)
>>> rdd2.collect()
18/02/15 08:09:54 WARN org.apache.spark.ExecutorAllocationManager: No stages are
running, but numRunningTasks != 0
[('b', 2), ('a', 3)]
-----END OF RESULTS-----
```

### Let's code

You need to create two RDDs composed of integers. First RDD must contain the odd numbers from 1 to 30000 and the second RDD the rest of the numbers from 1 to 30000.

Then create a pairRDD where the key is the number and the value for the odd numbers is an "a" multiple times (exactly as specified in the number) and for even numbers a "b". For example:

```
[(1, 'a'), (3, 'aaa'), (5, 'aaaaa'), (7, 'aaaaaaa'), (9, 'aaaaaaaaa')]
```

Then get both rdds together and create another RDD only with the key, discarding the value. Add all the keys and show the result. Calculate statistics from the previous RDD, it should show something like this:

```
(count: 29999, mean: 15000.0, stdev: 8659.9653579, max: 29999.0, min: 1.0)
```

Then take only the first 100 numbers and calculate the statistics again.

-----START OF RESULTS-----

```
kamal_nandan@cluster-643f-m:~$ spark-submit OddEven.py
18/02/15 10:54:46 INFO org.spark_project.jetty.util.log: Logging initialized @4074ms
18/02/15 10:54:46 INFO org.spark_project.jetty.server.Server: jetty-9.3.z-SNAPSHOT
```

```
18/02/15 10:54:46 INFO org.spark_project.jetty.server.Server: Started @4198ms
18/02/15 10:54:46 INFO org.spark_project.jetty.server.AbstractConnector: Started
ServerConnector@33c72dbb{HTTP/1.1,[http/1.1]}{0.0.0.0:4040}
18/02/15 10:54:47 INFO
com.google.cloud.hadoop.fs.gcs.GoogleHadoopFileSystemBase: GHFS version: 1.6.3-
hadoop2
```

```
****Printing summary stats of all the keys****
```

```
(count: 29999, mean: 15000.0, stdev: 8659.9653579, max: 29999.0, min: 1.0)
```

```
****Printing summary stats of first 100 items****
```

```
(count: 100, mean: 100.0, stdev: 57.7321400954, max: 199.0, min: 1.0)
```

```
18/02/15 10:54:52 INFO org.spark_project.jetty.server.AbstractConnector: Stopped
Spark@33c72dbb{HTTP/1.1,[http/1.1]}{0.0.0.0:4040}
```

-----END OF RESULTS-----

Paste your python code here:

-----START OF CODE-----

```
from pyspark import SparkConf, SparkContext
from pyspark.mllib.stat import Statistics
```

```
APP_NAME = "My Spark Application"
NTIMES = 30000
```

```
def oddeven(sc):
    odds = [x for x in range(1,NTIMES) if x % 2 != 0]
    evens = [x for x in range(1,NTIMES) if x % 2 == 0]
```

```
    rddodds = sc.parallelize(odds)
    rddevens = sc.parallelize(evens)
    #print(rddodds.collect())
    #print(rddevens.collect())
```

```
    rddoddspair = rddodds.map(lambda x: (x, "a" * x))
    rddevenspair = rddevens.map(lambda x: (x, "b" * x))
    #print(rddoddspair.collect())
    #print(rddevenspair.collect())
```

```
    allkeys = rddoddspair.keys().union(rddevenspair.keys())
    #print(allkeys.collect())
```

```
    #calculate the statistics now
    summary = allkeys.stats()
    print("\n****Printing summary stats of all the keys****")
    print("(count: " + str(summary.count()) + ", mean: " +
str(summary.mean()) + ", stdev: " + str(summary.stdev()) + ", max: " +
str(summary.max()) + ", min: " + str(summary.min()) + ")")
```

```

        # Now calculate the summary stats of the first 100 numbers and their
stats;
        summary = sc.parallelize(allkeys.take(100)).stats() # IMP: This
method works but this is an overkill; we don't really need to parallelize and
create another RDD, because now the datasize is very small and with the
driver program; it would be much more efficient to call some other common
function to get statistics (such as from pandas or scipy or statsmodels)

        print("\n***Printing summary stats of first 100 items***")
        print("(count: " + str(summary.count()) + ", mean: " +
str(summary.mean()) + ", stdev: " + str(summary.stdev()) + ", max: " +
str(summary.max()) + ", min: " + str(summary.min()) + ")")

#*****Main function*****
if __name__ == "__main__":
    # Configure Spark
    conf = SparkConf().setAppName(APP_NAME)
    conf = conf.setMaster("local[*]")
    sc = SparkContext(conf=conf)
    oddeven(sc)

```

-----END OF CODE-----