Lab 4: Create and Manipulate Pair RDDs (Python)

About This Lab

Objective:
Create pair RDD's and use various functions to transform these RDD's using Python in Zeppelin.

File Locations:
`/home/zeppelin/spark/data/`

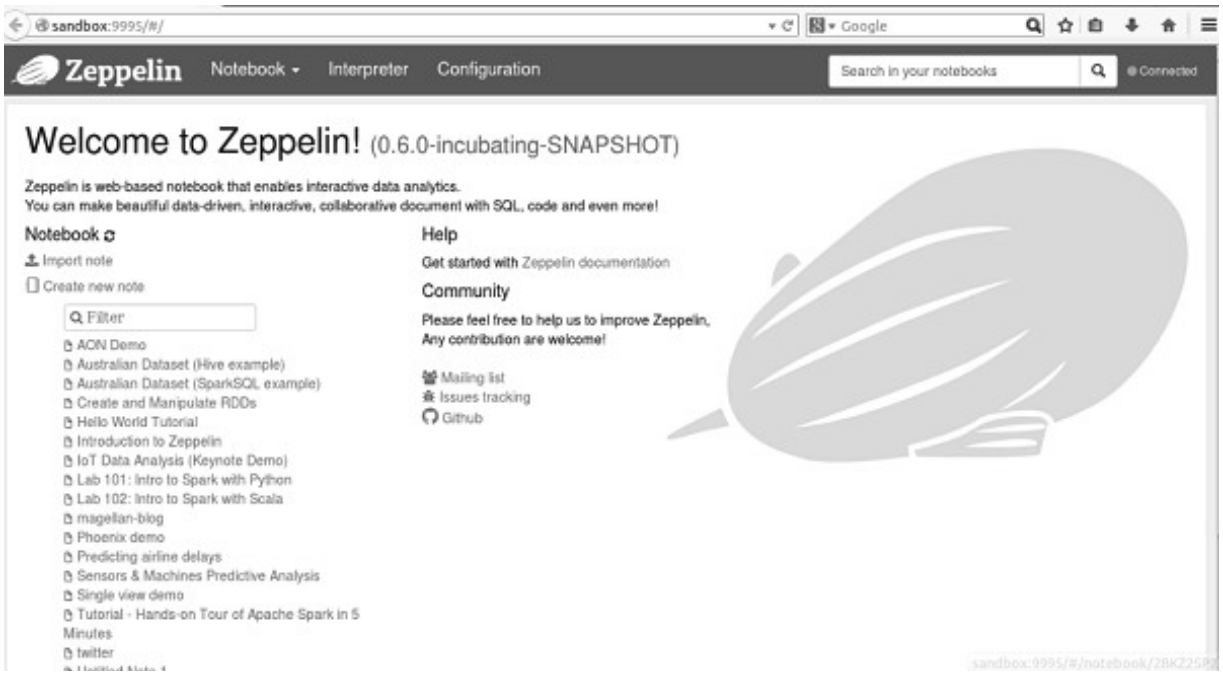
Successful Outcome:
REQUIRED: Create pair RDDs and perform various operations.
OPTIONAL: Complete challenge labs performing more complex operations.

Lab Steps

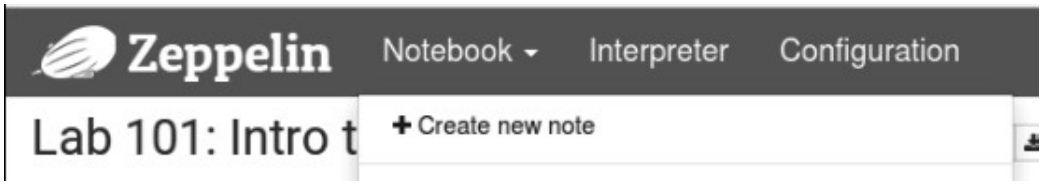
Perform the following steps:

1. **Create a Pair RDD note in Zeppelin.**

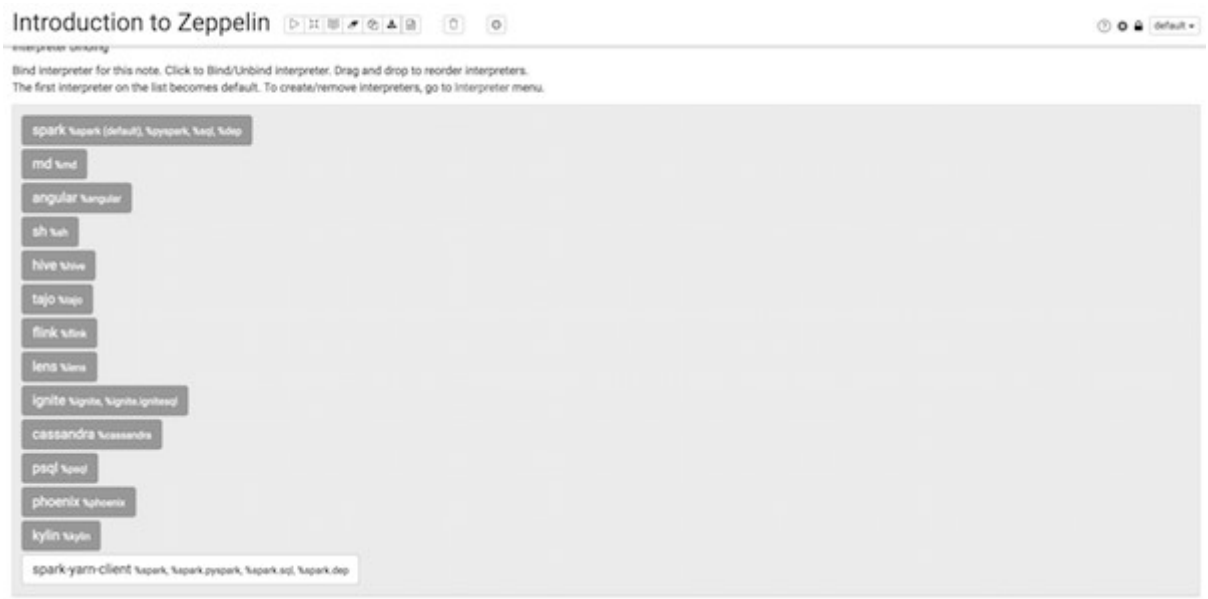
    a. Open the Firefox browser and enter the following URL to view the Zeppelin UI.
    http://sandbox:9995/



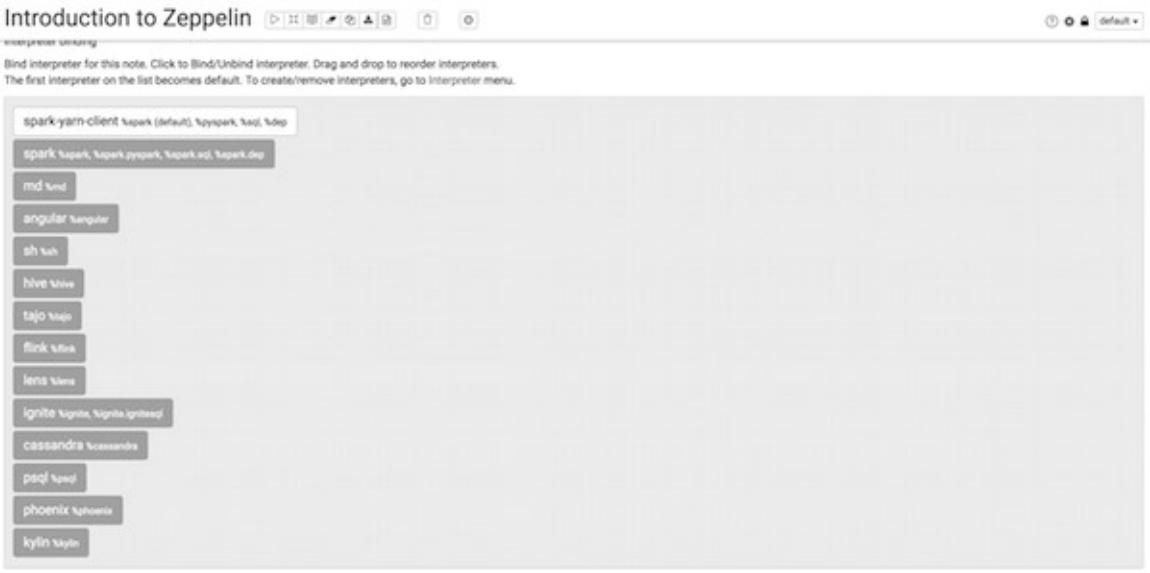
b. Click on Notebook and select Create new note on the drop down. Name this note Pair RDDs.



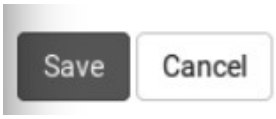
c. At the top right click on the gear icon to change interpreter binding.



Drag the spark-yarn-client to the top and click save.



The first interpreter on the list becomes default.



2. **Create a Pair RDD from a text file using map().**

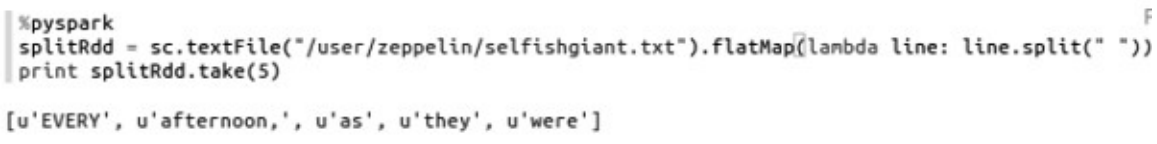
   a. Recreate the RDD splitRDD using the selfishgiant.txt file by importing it to an RDD as a text file and then flatting it into individual word elements. Then view the first 5 words to confirm the RDD exists and is correctly formatted.

      In the code below, there are no line breaks between splitRdd and (“ “)). Please refer to the screenshot.

```
%pyspark

splitRdd = sc.textFile(“/user/zeppelin/selfishgiant.txt”).flatMap(lambda line:
line.split(“ “))

print splitRdd.take(5)
```



NOTE:

In the previous lab, this RDD creation was performed over two steps, creating an intermediary RDD named baseRdd. The creation of the intermediary is not necessary unless it needs to be used in a future step.

b. Use `map()` to create an RDD named `mappedRdd` that converts each element into a key-value pair with a value of 1. View the first five elements to confirm successful operation.

```
%pyspark

mappedRdd = splitRdd.map(lambda word: (word, 1))

print mappedRdd.take(5)
```

```
%pyspark
mappedRdd = splitRdd.map(lambda word: (word, 1))
print mappedRdd.take(5)

[(u'EVERY', 1), (u'afternoon,', 1), (u'as', 1), (u'they', 1), (u'were', 1)]
```

3. **Create Pair RDDs using zip functions and perform simple transformations.**

a. Create a variable named `months` that contains the values `Jan`, `Feb`, `Mar`, `Apr`, `May`, `Jun`, and `Jul` as a list of string values. Convert this to an RDD named `monthsRdd`. Then create another RDD named `monthsIndexed0Rdd` using `zipWithIndex()` to create a Pair RDD that automatically assigns a value to each element based on its position in the list.

REMINDER:

The first element will be assigned a value of "0" using this function.

```
%pyspark

months =("Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul")

monthsRdd = sc.parallelize(months)

monthsIndexed0Rdd = monthsRdd.zipWithIndex()

print monthsIndexed0Rdd.collect()
```

```
%pyspark
months = ("Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul")
monthsRdd = sc.parallelize(months)
monthsIndexed0Rdd = monthsRdd.zipWithIndex()
print monthsIndexed0Rdd.collect()
```

```
[('Jan', 0), ('Feb', 1), ('Mar', 2), ('Apr', 3), ('May', 4), ('Jun', 5), ('Jul', 6)]
```

b. Use `map()` to convert the value for each month to the actual month number and store this in a new RDD named `monthsIndexed1Rdd`. For reference, `Jan` should have a value of 1, `Feb` should have a value of 2, and so on. View the new RDD to confirm success.

```
%pyspark

monthsIndexed1Rdd = monthsIndexed0Rdd.map(lambda (x,y): (x,y+1))

print monthsIndexed1Rdd.collect()
```

```
%pyspark
monthsIndexed1Rdd = monthsIndexed0Rdd.map(lambda (x,y): (x,y+1))
print monthsIndexed1Rdd.collect()
```

```
[('Jan', 1), ('Feb', 2), ('Mar', 3), ('Apr', 4), ('May', 5), ('Jun', 6), ('Jul', 7)]
```

c. Create a new RDD named `monthsIndexed2Rdd` that performs the same operation on `monthsIndexed0Rdd` as in the previous step but uses `mapValues()` instead of `map()` to perform the operation. View the new RDD and confirm it looks identical to the output of `monthsIndexed1Rdd`.

```
%pyspark

monthsIndexed2Rdd = monthsIndexed0Rdd.mapValues(lambda y: y+1)

print monthsIndexed2Rdd.collect()
```

```
%pyspark
monthsIndexed2Rdd = monthsIndexed0Rdd.mapValues(lambda y: y+1)
print monthsIndexed2Rdd.collect()
```

```
[('Jan', 1), ('Feb', 2), ('Mar', 3), ('Apr', 4), ('May', 5), ('Jun', 6), ('Jul', 7)]
```

NOTE:

No difference exists between the two previous lab steps from Spark's perspective. The `mapValues` function simply performs a `map()` and returns the key without modification, while performing the function you define on the value.

d. Create a variable named `quarters` that contains the following seven values: `1, 1, 1, 2, 2, 2,` and `3`. Convert the variable into an RDD named `quartersRdd`. Then create an RDD named `monthsZipQuarters` and use `zip()` to create a Pair RDD that assigns each value from `quartersRdd` to a month in `monthsRdd`. Finally, view the output and make sure that each month was assigned to the correct quarter in the final RDD.

```
%pyspark

quarters = (1, 1, 1, 2, 2, 2, 3)

quartersRdd = sc.parallelize(quarters)

monthsZipQuarters = monthsRdd.zip(quartersRdd)

print monthsZipQuarters.collect()
```

```
%pyspark
quarters = (1, 1, 1, 2, 2, 2, 3)
quartersRdd = sc.parallelize(quarters)
monthsZipQuarters = monthsRdd.zip(quartersRdd)
print monthsZipQuarters.collect()

[('Jan', 1), ('Feb', 1), ('Mar', 1), ('Apr', 2), ('May', 2), ('Jun', 2), ('Jul', 3)]
```

e. Perform the following operations on `monthsZipQuarters` without creating new RDDs: view the keys only, view the values only, and view the contents of the RDD sorted alphabetically by key.

```
%pyspark

print monthsZipQuarters.keys().collect()

print monthsZipQuarters.values().collect()

print monthsZipQuarters.sortByKey().collect()
```

```
%pyspark
print monthsZipQuarters.keys().collect()
print monthsZipQuarters.values().collect()
print monthsZipQuarters.sortByKey().collect()

['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul']
[1, 1, 1, 2, 2, 2, 3]
[('Apr', 2), ('Feb', 1), ('Jan', 1), ('Jul', 3), ('Jun', 2), ('Mar', 1), ('May', 2)]
```

4. **Count the number of times words appear in a Pair RDD and manipulate the output.**

   a. Use the `mappedRDD` created in a previous step and create a new RDD named `reducedByKeyRdd` that reduces the file so that each word appears only once but has a value equal to the number of times it appeared in the original RDD. View the first five elements of the new RDD to confirm successful operation.

```
%pyspark

reducedByKeyRdd = mappedRdd.reduceByKey(lambda x,y: x+y)

print reducedByKeyRdd.take(5)
```

```
%pyspark
reducedByKeyRdd = mappedRdd.reduceByKey(lambda x,y: x+y)
print reducedByKeyRdd.take(5)

[(u'', 33), (u'all', 11), (u'stole', 1), (u'Through', 1), (u'cried', 3)]
```

   b. Use `map()` to create a new RDD named `flippedRdd` that switches your keys and values so that the current keys become the values, and the values become the keys. View the first five elements of the new RDD to confirm successful operation.

```
%pyspark

flippedRdd = reducedByKeyRdd.map(lambda (x,y): (y,x))

print flippedRdd.take(5)
```

```
%pyspark
flippedRdd = reducedByKeyRdd.map(lambda (x,y): (y,x))
print flippedRdd.take(5)

[(33, u''), (11, u'all'), (1, u'stole'), (1, u'Through'), (3, u'cried')]
```

   c. Create a new RDD named `orderedRdd` that manipulates `flippedRDD` and arranges the words in descending order by number of times they appear. View the first five elements of the new RDD to confirm successful operation.

```
%pyspark

orderedRdd = flippedRdd.sortByKey(ascending=False)

print orderedRdd.take(5)
```

```
%pyspark
orderedRdd = flippedRdd.sortByKey(ascending=False)
print orderedRdd.take(5)

[(148, u'the'), (85, u'and'), (44, u'he'), (38, u'to'), (33, u'')]
```

Result

You have successfully created and manipulated Pair RDD's using various functions.

Challenge Labs

The labs below work with Pair RDDs to perform real-world operations. In some cases, the solutions to the lab utilize programming techniques not explicitly described in the course lecture. These techniques, however, should be clear and easy to understand by carefully following the instructions. If you have questions and are in an instructor-supported class, please ask for assistance as needed.

You may want to start by creating a new notebook named Pair RDD Challenge Labs, but this is up to you.

Perform the following steps:

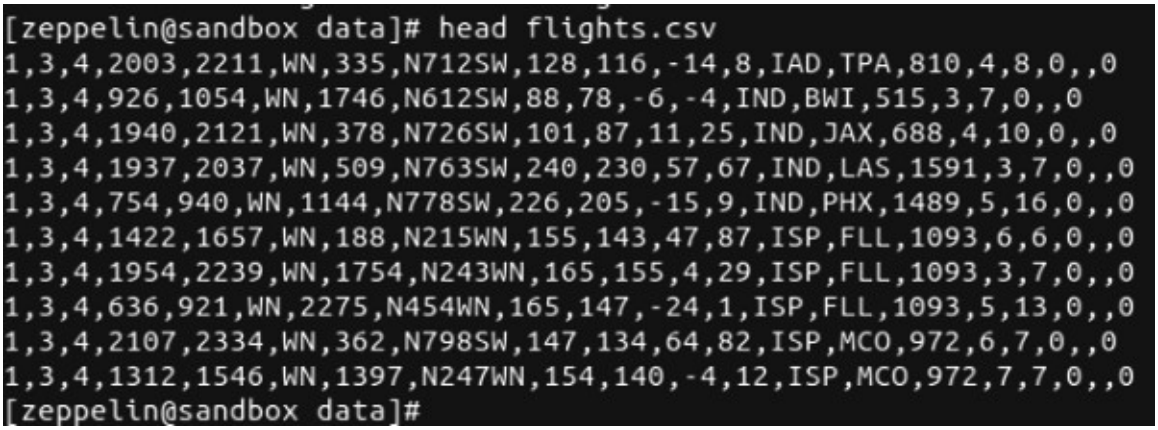
1. **Determine the airlines with the greatest number of flights.**
    a. Go back to a terminal window that has used SSH to connect to the sandbox Docker environment and change to the /home/zeppelin/spark/data directory if necessary. View the contents of this directory and confirm the existence of three files: airports.csv, plane-data.csv, and flights.csv.

```
# ls
```

```
[zeppelin@sandbox data]# pwd
/home/zeppelin/spark/data
[zeppelin@sandbox data]# ls
airports.csv  data.txt     plane-data.csv    small_blocks.txt
carriers.csv  flights.csv  selfishgiant.txt
[zeppelin@sandbox data]#
```

b. Use `head` to view the first few lines of the flights.csv file.

```
# head flights.csv
```



Each column in the file can be interpreted using the guide below. The first comma-separated value in each line (index number 0) represents the month, the second value represents the day of the month, and so on. Of note for our purposes: the sixth value (index number 5) represents the carrier for each flight.

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

c. Use Zeppelin to import this file into the `/user/zeppelin` folder in HDFS.

```
%sh

hdfs dfs -put /home/zeppelin/spark/data/flights.csv /user/zeppelin/flights.csv
```

QUESTION:

Why do this in Zeppelin instead of from the command line?

ANSWER:

When the tasks are performed in a Zeppelin notebook, the entire series of actions can be exported and then imported and replayed on another system. This will be discussed in more detail in a later lab exercise.

d. Create an RDD named `carrierRdd` by performing the following transformations:

1. Import the text file from HDFS using `sc.textFile()`.

2. Split the lines into an array of individual elements using `map()`
(Hint: The elements are comma-separated rather than space-separated as in previous examples.)

3. Use `map()` to create a key-value pair from only the elements in the sixth column (index number 5) - which can be specified by appending `[5]` to the anonymous function value – and assign each instance a value of 1.

4. View the first five elements to confirm successful operation.

```
%pyspark

carrierRdd = sc.textFile("/user/zeppelin/flights.csv").map(lambda val:
val.split(",")).map(lambda column: (column[5],1))

print carrierRdd.take(5)
```

```
%pyspark                                                                                    FINISHE
carrierRdd = sc.textFile("/user/zeppelin/flights.csv").map(lambda val: val.split(",")).map(lambda column: (column[5],1))
print carrierRdd.take(5)

[(u'WN', 1), (u'WN', 1), (u'WN', 1), (u'WN', 1), (u'WN', 1)]
```

NOTE:

As in a previous example, these operations to create carrierRdd could have been performed in stages, using intermediate RDDs at each transformation step. We do not need the data in these intermediate forms, however, so chaining together multiple transformations to get to the final output works fine.

e. Perform a reduce and sort the results, then display the top three carrier codes by number of flights based on this data.

```
%pyspark

carriersSorted = carrierRdd.reduceByKey(lambda x,y: x+y).map(lambda (a,b):
(b,a)).sortByKey(ascending=False)

print carriersSorted.take(3)
```

```
%pyspark
carriersSorted = carrierRdd.reduceByKey(lambda x,y: x+y).map(lambda (a,b): (b,a)).sortByKey(ascending=False)
print carriersSorted.take(3)

[(356167, u'WN'), (175969, u'AA'), (166445, u'OO')]
```

2. **Determine the most common routes between two cities.**

a. The next exercise uses the flights.csv file from the previous lab, as well as the airports.csv file. Go back to the terminal window and take a look at the first few lines of the airports.csv file.

```
# head airports.csv
```

```
[zeppelin@sandbox data]# pwd
/home/zeppelin/spark/data
[zeppelin@sandbox data]# head airports.csv
iata,airport,city,state,country,lat,long
00M,Thigpen,BaySprings,MS,USA,31.95376472,-89.23450472
00R,LivingstonMunicipal,Livingston,TX,USA,30.68586111,-95.01792778
00V,MeadowLake,ColoradoSprings,CO,USA,38.94574889,-104.5698933
01G,Perry-Warsaw,Perry,NY,USA,42.74134667,-78.05208056
01J,HilliardAirpark,Hilliard,FL,USA,30.6880125,-81.90594389
01M,TishomingoCounty,Belmont,MS,USA,34.49166667,-88.20111111
02A,Gragg-Wade,Clanton,AL,USA,32.85048667,-86.61145333
02C,Capitol,Brookfield,WI,USA,43.08751,-88.17786917
02G,ColumbianaCounty,EastLiverpool,OH,USA,40.67331278,-80.64140639
[zeppelin@sandbox data]#
```

Each column in the file can be interpreted using the guide below. The first comma-separated value in each line (index number 0) represents the airport code, the second value represents the airport name, and so on. Of note for our purposes: the airport code (index number 0) and the airport city (index number 2).

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

From the flights.csv file used earlier, columns 13 and 14 (index values 12 and 13) will be used in this exercise.

Field	Index	Example data
Origin	**12**	**JAX**
Dest	**13**	**FLL**

b. Use Zeppelin to import the airports.csv file into the `/user/zeppelin` folder in HDFS.

```
%sh

hdfs dfs -put /home/zeppelin/spark/data/airports.csv
/user/zeppelin/airports.csv
```

c. Create an RDD named `cityRdd` by performing the following transformations:

1. Import the text file from HDFS using `sc.textFile()`.

2. Split the lines into an array of individual elements using `map()`
(Hint: Once again, the elements are comma-separated rather than space-separated.)

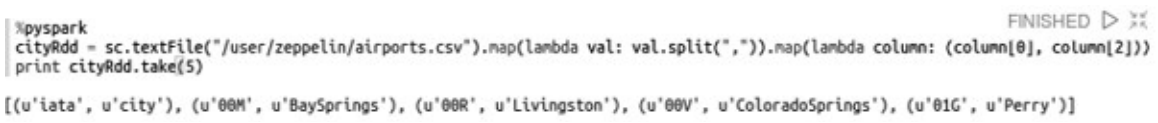
3. Use `map()` to pull out only the airport code and city elements in the first and third columns (index numbers 0 and 2).

4. View the first five elements to confirm successful operation.

```
%pyspark

cityRdd = sc.textFile(“/user/zeppelin/airports.csv”).map(lambda val:
val.split(“,”)).map(lambda column: (column[0], column[2]))


print cityRdd.take(5)
```



d. Create an RDD named `flightOrigDestRdd` by performing the following transformations:

   1. Import the text file from HDFS using `sc.textFile()`.

   2. Split the lines into an array of individual elements using `map()`.

   3. Use `map()` to pull out only the origin and destination elements in the 13th and 14th columns (index numbers 12 and 13).

   4. View the first five elements to confirm successful operation.

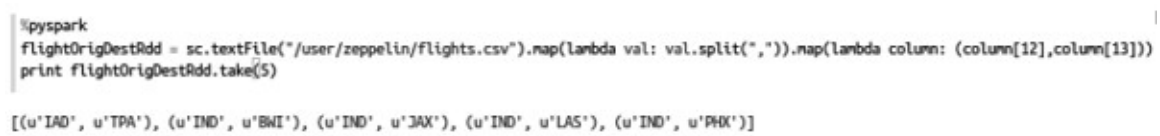
NOTE:

Some of this code can be copied and pasted from a previous paragraph in the Zeppelin notebook.

```
%pyspark

flightOrigDestRdd = sc.textFile(“/user/zeppelin/flights.csv”).map(lambda val:
val.split(“,”)).map(lambda column: (column[12],column[13]))

print flightOrigDestRdd.take(5)
```



e. Use `join()` to join `flightOrigDestRdd` and `cityRdd` into a third RDD named `origJoinRdd`.

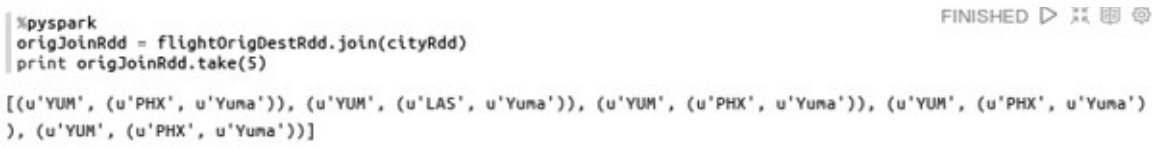
   This operation will result in an RDD that contains the origin code as the key, with a value of (destination code, origin city). This is half of the operation needed to get origin and destination cities.

View the first five elements to confirm successful operation.

```
%pyspark

origJoinRdd = flightOrigDestRdd.join(cityRdd)

print origJoinRdd.take(5)
```



f. Next use `join()` again to create an RDD named `destOrigJoinRdd` using `origJoinRdd` as a source and joining it `cityRdd` once again. Before performing the join operation, use `values()` to filter out the origin code (which is no longer needed) and pull out only the destination code and city name from the previous transformation.

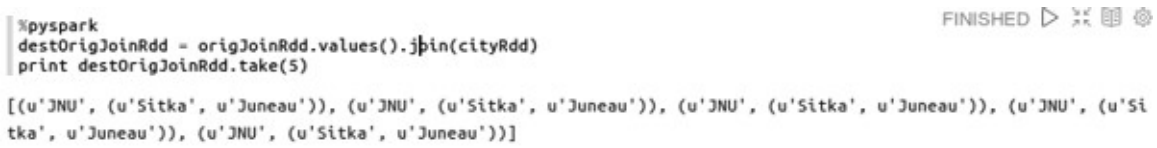
   This operation will result in an RDD that contains the destination code as the key, with a value of (origin city, destination city).

   View the first five elements to confirm successful operation.

```
%pyspark

destOrigJoinRdd = origJoinRdd.values().join(cityRdd)

print destOrigJoinRdd.take(5)
```

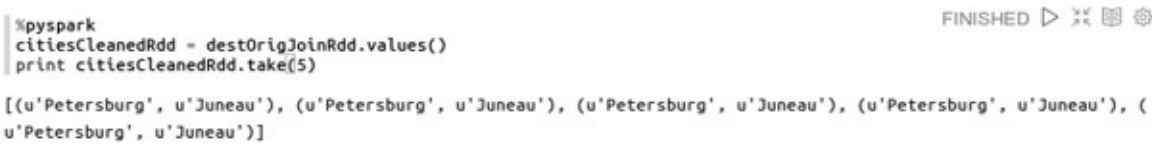


g. Create another RDD named `citiesCleanedRdd` that contains only the values of the `destOrigJoinRdd` (in other words, just the origin and destination city names). View the first five elements to confirm successful operation.

```
%pyspark

citiesCleanedRdd = destOrigJoinRdd.values()

print citiesCleanedRdd.take(5)
```

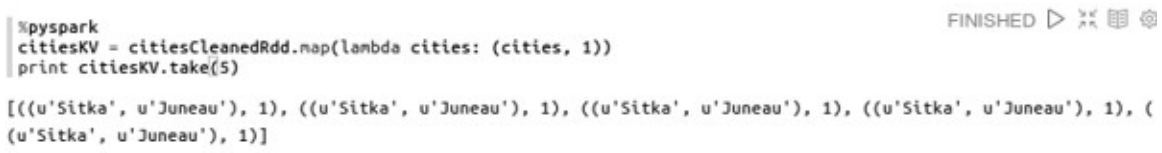


h. Use `map()` to convert the key-value pairs in `citiesCleanedRdd` into keys for a new RDD named `citiesKV`, and give each key a value of 1. View the first five elements to confirm successful operation.

```
%pyspark

citiesKV = citiesCleanedRdd.map(lambda cities: (cities, 1))

print citiesKV.take(5)
```

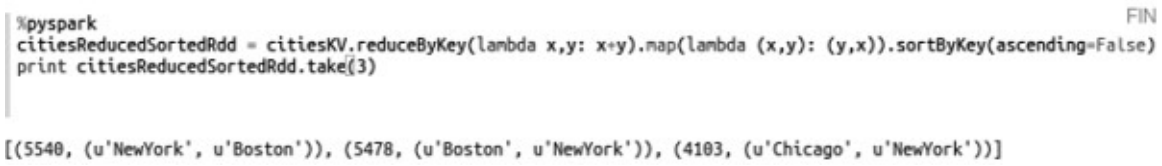


i. Create an RDD named `citiesReducedSortedRdd` that reduces by key, swaps the keys and values, and then sorts by key in descending order. View the first three elements to confirm successful operation.

```
%pyspark

citiesReducedSortedRdd = citiesKV.reduceByKey(lambda x,y: x+y).map(lambda
(x,y): (y,x)).sortByKey(ascending=False)

print citiesReducedSortedRdd.take(3)
```



NOTE:

The top three origin city / destination combinations are New York to Boston, Boston to New York, and Chicago to New York.

3. **Find the longest departure delays for any airline that experienced a delay of 15 minutes or more.**

This exercise once again uses the flights.csv file. This time we use the unique carrier code in column 6 (index value 5) and the departure delay value in minutes, which is in column 12 (index value 11).

Field	Index	Example data
UniqueCarrier	**5**	**WN**
DepDelay	**11**	**8**

b. Create an RDD named `delayRdd` by performing the following transformations:

1. Import the flights.csv file from HDFS using `sc.textFile()`.

2. Split the lines into an array of individual elements using `map()`.

3. Use `filter()` to remove any lines for which the value of column 12 (index value 11) is less than 15. Because the `sc.textFile()` operation reads in all values as strings, you will need to cast the values in column 12 as integers prior to performing the `filter()` evaluation.

4. Use `map()` to pull out only the carrier code and departure delay elements in the 6th and 12th columns (index numbers 5 and 11).

5. View the first five elements to confirm successful operation.

```
%pyspark
delayRdd = sc.textFile("/user/zeppelin/flights.csv").map(lambda val: val.split(",")).filter(lambda delay: int(delay[11]) > 15).map(lambda column: (column[5],column[11]))
print delayRdd.take(5)

[(u'WN', u'25'), (u'WN', u'67'), (u'WN', u'87'), (u'WN', u'29'), (u'WN', u'82')]
```

For sake or readability, here is another screenshot of the above code with lines wrapped so that the code can be viewed in a larger font.

```
%pyspark
delayRdd = sc.textFile("/user/zeppelin/flights.csv").map(lambda val: val.split(",")).filter(lambda delay:
int(delay[11]) > 15).map(lambda column: (column[5],column[11]))
print delayRdd.take(5)

[(u'WN', u'25'), (u'WN', u'67'), (u'WN', u'87'), (u'WN', u'29'), (u'WN', u'82')]
```

c. Create an RDD named `delayMaxRdd` that reduces the elements in `delayRdd` and returns only the longest delay per airline. For this exercise, it is not necessary to sort the values from largest to smallest.

   Display five values to confirm successful operation.

NOTE:

The reduce operation will need to compare all values for the same key and only keep the largest value in the final output.

The values in `delayRdd` are strings, so to compare the values they will first need to be cast as integers, similar to the `filter()` operation performed in the first step of this exercise.

```
%pyspark

delayMaxRdd = delayRdd.reduceByKey(lambda x,y: max(int(x), int(y)))

print delayMaxRdd.take(5)
```

```
%pyspark
delayMaxRdd = delayRdd.reduceByKey(lambda x,y: max(int(x), int(y)))
print delayMaxRdd.take(5)

[(u'OO', 767), (u'AA', 1521), (u'DL', 716), (u'CO', 1011), (u'UA', 1268)]
```

4. **Remove records than contain incomplete data from a file.**

   a. The next exercise uses the plane-data.csv. Go back to the terminal window and take a look at the first few lines of the plane-data.csv file.

```
# head plane-data.csv
```

```
[root@sandbox data]# pwd
/home/zeppelin/spark/data
[root@sandbox data]# head plane-data.csv
tailnum,type,manufacturer,issue_date,model,status,aircraft_type,engine_type,year
N050AA
N051AA
N052AA
N054AA
N055AA
N056AA
N057AA
N058AA
N059AA
[root@sandbox data]#
```

Note that in the screenshot above, this file contains the column header names, followed by the column values. In this case, the first few records only have values for the first column, and the rest of the values are blank.

To see what complete records should look like, take a look at the last few lines of the file.

```
# tail plane-data.csv
```

```
[root@sandbox data]# tail plane-data.csv
N995AT,Corporation,BOEING,11/08/2002,717-200,Valid,Fixed Wing Multi-Engine,Turbo
-Fan,2002
N995DL,Corporation,MCDONNELL DOUGLAS AIRCRAFT CO,03/06/1992,MD-88,Valid,Fixed Wi
ng Multi-Engine,Turbo-Fan,1991
N996AT,Corporation,BOEING,07/30/2002,717-200,Valid,Fixed Wing Multi-Engine,Turbo
-Fan,2002
N996DL,Corporation,MCDONNELL DOUGLAS AIRCRAFT CO,02/27/1992,MD-88,Valid,Fixed Wi
ng Multi-Engine,Turbo-Fan,1991
N997AT,Corporation,BOEING,01/02/2003,717-200,Valid,Fixed Wing Multi-Engine,Turbo
-Fan,2002
N997DL,Corporation,MCDONNELL DOUGLAS AIRCRAFT CO,03/11/1992,MD-88,Valid,Fixed Wi
ng Multi-Engine,Turbo-Fan,1992
N998AT,Corporation,BOEING,01/23/2003,717-200,Valid,Fixed Wing Multi-Engine,Turbo
-Fan,2002
N998DL,Corporation,MCDONNELL DOUGLAS CORPORATION,04/02/1992,MD-88,Valid,Fixed Wi
ng Multi-Engine,Turbo-Jet,1992
N999CA,Foreign Corporation,CANADAIR,07/09/2008,CL-600-2B19,Valid,Fixed Wing Mult
i-Engine,Turbo-Jet,1998
N999DN,Corporation,MCDONNELL DOUGLAS CORPORATION,04/02/1992,MD-88,Valid,Fixed Wi
ng Multi-Engine,Turbo-Jet,1992
[root@sandbox data]#
```

Each column in the file can be interpreted using the guide below. Note that there are nine possible column values for each record (index 0 through 8).

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

b. Use Zeppelin to import the plane-data.csv file into the `/user/zeppelin` folder in HDFS.

```
%sh

hdfs dfs -put /home/zeppelin/spark/data/plane-data.csv /user/zeppelin/plane-
data.csv
```

```
%sh
hdfs dfs -put /home/zeppelin/spark/data/plane-data.csv /user/zeppelin/plane-data.csv
```

c. Create an RDD named `planeDataRdd` from the plane-data.csv file. Before performing any transformations, use `count()` to display the number of lines in the RDD.

```
%pyspark

planeDataRdd = sc.textFile("/user/zeppelin/plane-data.csv")

print planeDataRdd.count()
```

```
%pyspark
planeDataRdd = sc.textFile("/user/zeppelin/plane-data.csv")
print planeDataRdd.count()

5030
```

d. Create an RDD named `cleanedPlaneDataRdd` by performing the following transformations:

1. Start with `planeDataRdd` from the previous step.

2. Split the lines into an array of individual elements using `map()`. (Hint: The elements are comma-separated.)

3. Use `filter()` to remove any lines that do not have a length of exactly 9 elements.

4. Use `count()` to display the number of lines in the new RDD and confirm that the data set contains fewer lines than before.

```
%pyspark

cleanedPlaneDataRdd = planeDataRdd.map(lambda val:
val.split(",")).filter(lambda elements: len(elements) == 9)

print cleanedPlaneDataRdd.count()
```

```
%pyspark
cleanedPlaneDataRdd = planeDataRdd.map(lambda val: val.split(",")).filter(lambda vals: len(vals) == 9)
print cleanedPlaneDataRdd.count()

4481
```