

## Homework 6: Spark

If you did not attend the lab session, please do the Lab Session exercises in the file "Spark Lab Session.docx" before you start with this homework. If you attended the lab session, please review the "Spark Lab Session.docx" document and repeat any steps that you will unsure about.

The first part of the homework will go over the basic steps we did in the Lab Session, using the movielens data set. The second part of the homework will introduce you to some advanced topics, i.e., Pair RDDs. The deliverables are explained on the last page.

### Basic Spark Operations with the MovieLens Dataset

In the following, I will explain the steps we go through without showing the code. If you want, you can first try coming up with the code yourself. I will provide the code immediately after the steps. So, it is up to you how much help you want.

1. If you haven't already done so, open your Virtual Machine, open a terminal window, and start the Spark Shell. The first thing we need to do is to create a RDD based on the MovieLens data. I create a RDD called *ratingsCSV* and link it to the textFile in /user/cloudera/movielens/input/ratings.csv with the help of the SparkContext object (sc). Make sure you use the correct path on HDFS that links to the ratings.csv. You may want to verify that your and mine path names are the same and adjust accordingly if it is not. Remember, you will not find out whether you have the correct path name until you do an action with the RDDs.
2. Let's show the first five elements with take. This is an action. If you get an error, make sure your path to the ratings.csv is correctly set. Btw. the u in front of each element means 'unicode'. Strings are enclosed in single quotes. The following shows how your RDD will look like:

u'userId,movieId,rating,timestamp'
u'1,169,2.5,1204927694'
u'1,2471,3.0,1204927438'
u'2,2571,3.5,1436165433'

3. The csv file still has the header, so, we need to filter out the header. Since we have not done this, I will show you how you can do this. (There may be other ways, but this way builds on what we have done so far.) I will refer to the new RDD with the name ratings. .first() will return the first element in the RDD.

```
>>> header = ratingsCSV.first()
>>> ratings = ratingsCSV.filter(lambda line: line != header)
>>> ratings.take(5)
```

As you can see, the header is not in the new RDD anymore.

u'1,169,2.5,1204927694'
u'1,2471,3.0,1204927438'
u'2,2571,3.5,1436165433'

4. Let's count the elements (one element in the RDD = one line = one rating) in the RDD ratings.
5. Let's filter out ratings for one movie and count the number of ratings for this movie. The first, less efficient way, is to create a regular expression searching for the movieID 2309. The problem is that the number 2309 could appear in the timestamp as well as in the userid. Use collect to print all ratings for movie 2309. We have quite a lot. How many ratings for this movie do we have?
6. Creating regular expression can become quite complicated very fast. A better solution is to split each element in ratings in each components userid, movieid, rating, and timestamp. We can do this with split.line(',') and map.

This will produce an array containing the four columns for each element. Use `take(5)` to verify that you did it correctly. I used `ratingsArray` to refer to this new RDD.

[u'1',u'169',u'2.5',u'1204927694']
[u'1',u'2471',u'3.0',u'1204927438']
[u'2',u'2571',u'3.5',u'1436165433']

The advantage is that we can know address each column explicitly, e.g., `[0]` is the `userid`, `[1]` the `movieid`, etc.

7. Print out only the `userID` and `movieID` for the first 10 elements only. Use *for* and *print* and use labels for `userID` and `movieID`.
8. Using the `ratingsArray` RDD and filter for movie with `movieID` 2309 and count the number of ratings. Hint: use `==` to find the exact value.
9. Save the result of 8 to a text file in your local Downloads folder.
10. Remember, RDD execution is lazy. And nothing is done until an action is called. The RDD that are created are not stored in any way even once an action is called. What is stored is the lineage, e.g., what transformations have been done. You can use `toDebugString()` to see the lineage. With the lineage, each RDD can be recreated.

Operations for questions 1 to 10:

1. `ratingsCSV=sc.textFile ("/user/cloudera/movielens/input/ratings.csv")`
2. `ratingsCSV.take(5)`
3. see above
4. `ratings.count()`
5. `movie2309 = ratings.filter (lambda line: ', 2309,' in line)`  
`movie2309.collect()`  
`movie2309.count()`  
45
6. `ratingsArray = ratings.map(lambda line : line.split(','))`
7. for rating in ratingsArray: print "userID : " + rating[0] + " movieID : " + rating[1]
8. `ratingsArray.filter(lambda line: line [1] == '2309').count()`  
45
9. `ratingsArray.filter(lambda line: line[1] == '2309').saveAsTextFile('file:/home/cloudera/Downloads/movie2309')`
10. `print ratingsArray.toDebugString()`

## Aggregating Data with Pair RDDs

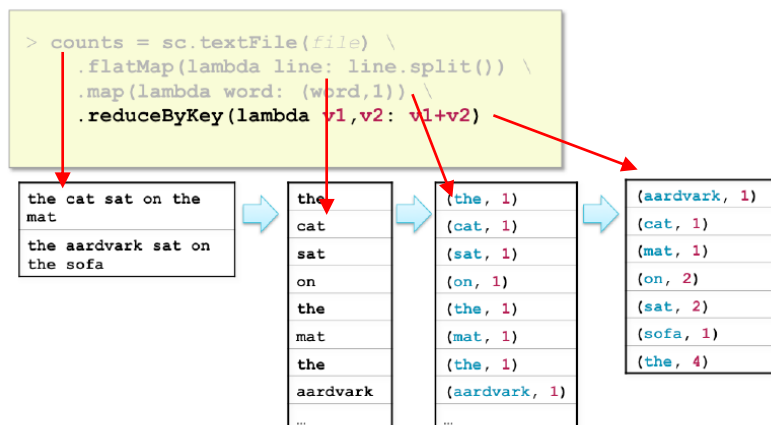
We continue working the web log files, as in the previous exercise. In this exercise you will learn to work with Pair RDDs. Pair RDDs are a special form of RDD. Each element must be a key-value pair (a two-element tuple). Keys and values can be any type. There are additional functions such as sorting, joining, grouping, counting that you can do with Pair RDDs as well as using map-reduce algorithms.

Pair RDD

(key1,value1)
(key2,value2)
(key3,value3)
...

With MapReduce from Hadoop you write jobs that each has a Map and a Reduce task. Outputs are saved to files and can then be read by another MapReduce jobs. That is limiting. Spark implements map-reduce with much greater flexibility than MapReduce. Map and reduce function can be interspersed. Results can be stored to memory which makes it easier to chain operations.

Map-reduce in Spark works on Pair RDDs. The map phase operates on one record at a time and maps each record to one or more new records (e.g., map, flatMap, filter, keyBy). The reduce phase works on the map output and consolidates multiple records (e.g., reduceByKey, sortByKey, mean). The word count example from the lab session is the classical example of a map-reduce problem.



There are some difference how MapReduce (Hadoop) and Spark ReduceByKey works. In Hadoop, a reducer is passed a key and a list of values and can aggregate those values any way it wants. In Spark's reduceByKey, two values for the same key are passed repeatedly and aggregated just those two values. This work is already done on each node and not centrally by a reducer like in MapReduce (Hadoop). That's why you cannot calculate a mean with reduceByKey. The reduceByKey requires that values can be processed in any order.

Why do we care about counting words? Because it typifies the characteristics of a MapReduce-able big data problem. Word counts are challenging over massive amounts of data and using a single computer node would be too time-consuming and number of unique words could exceed available memory. Map-reduce breaks complex tasks down into smaller elements which can be executed in parallel. Many common tasks are very similar to word count, e.g., log file analysis.

## Use Pair RDDs: Explore Web Log Files

1. The first step is to get the data into key-value form. Because reducing and joining large datasets can take time, we first reduce the data set.

```
>>> logs=sc.textFile("weblogs/2013-12*")
```

2. Next, we bring the data in a key-value form. Since our first goal is to count the number of requests a user made, we need the userID as a key and the value as one (similar to the word count example). We use the transformation map() to first split the logs and then use map() to create a Pair RDD with the user ID as the key, and the integer 1 as the value. (The user ID is the third field in each line.) Your RDD will look something like this:

(userid,1)
(userid,1)
(userid,1)
...

```
>>> users=logs.map(lambda line: line.split()).map(lambda words: (words[2],1))
```

```
>>> users.take(5)
```

3. Next, we will calculate the number of requests a user made by using the transformation reduce() to sum the values for each user ID. Your RDD will be similar to:

(userid,5)
(userid,7)
(userid,2)
...

```
>>> userreqs = users.reduceByKey(lambda count1,count2: count1 + count2)
```

```
>>> userreqs.take(5)
```

4. You can use count() to see how many unique users you have.
5. Next, we want to determine how many users visited the site for each frequency. That is, how many users visited once, twice, three times and so on. Use map to reverse the key and value, like this:

(5,userid)
(7,userid)
(2,userid)
...

```
>>> reqsuser = userreqs.map(lambda (userid,freq):(freq,userid))
```

```
>>> reqsuser.take(5)
```

```
>>> reqsuser.sortByKey(0).take(5)
```

SortByKey will sort by the key and 0 specifies descending order (1 would specify ascending order).

Next, use the countByKey action to return a Map of frequency:user-count pairs.

```
>>> freqcount = reqsuser.countByKey()
```

```
>>> print freqcount
```

6. Create an RDD where the user id is the key, and the value is the list of all the IP addresses that user has connected from. (IP address is the first field in each request line (words[0]).) Hint: Map to (userid, ipaddress) and then use groupByKey.

(userid,20.1.34.55)
(userid,245.33.1.1)
(userid,65.50.196.141)
...



(userid,[20.1.34.55, 74.125.239.98])
(userid,[75.175.32.10, 245.33.1.1, 66.79.233.99])
(userid,[65.50.196.141])
...

```
>>>userips = logs \
...     .map(lambda line: line.split()) \
...     .map(lambda words: (words[2],words[0])) \
...     .groupByKey()
```

7. Print out the first 10 user ids, and their IP list.

```
>>> print ([userip[0], [ip for ip in userip[1]]] \
... for userip in userips.take(10))
```

## Movie Ratings Count

1. We went over the word count example. Let us adjust it so that it counts the number of ratings for each movie. Since the ratingsArray already has each userID on one element, we only need to extract the userID from each element and pair it with one. This is what the .map() function does. Then, we will reduceByKey and sum up the one's. take() will get the first five elements of the new RDD.

```
>>>ratingsArray.map(lambda word: (word[1], 1)) \
...     .reduceByKey(lambda r1, r2: r1+r2).take(5)
```

2. We can get the movie with the most ratings by first switching the key-value pair and then sorting on the new key. Map(lambda word: (word[1], word[0])) switches the key-value pair around. Now, the ratings count is first, and then the movieid. SortByKey(0) will sort on the key (count of movie ratings) descending. And then we will print out the first five elements. This gives us the movieid with the most ratings.

```
>>>ratingsArray.map(lambda word: (word[1], 1)) \
...     .reduceByKey(lambda r1, r2: r1+r2) \
...     .map(lambda word: (word[1], word[0])) \
...     .sortByKey(0).take(5)
```

## Average Movie Rating

(Code adapted from here: <http://stackoverflow.com/questions/29930110/calculating-the-averages-for-each-key-in-a-pairwise-k-v-rdd-in-spark-with-pyth>)

1. Let's calculate the average movie rating for each movie. First, we need to create a key value pair with the movieID as the key, and the rating as the value. The rating needs to be changed into a float.

```
>>> movieIDratings = ratingsArray.map(lambda words: (words[1],  
float(words[2])))
```

2. Then, we use aggregateByKey to calculate the sum and count. The (0,0) is the starting number. The first lambda a, b will calculate a running sum (a[0] + b), and a running count (a[1] + 1) on each worker node. The second lambda a, b is central summing up of all datanode sums (a[0] + b[0]) and summing up of all datanode counts (a[1] + b[1]). At least, that is my understanding.

```
>>> rdd = movieIDratings.aggregateByKey ((0,0), \  
... lambda a, b: (a[0] + b, a[1] + 1, \  
... lambda a, b: (a[0] + b[0], a[1] + b[1]))
```

The result is a key-value pair with the final sum and final count for each movieID.

```
>>> rdd.take(5)
```

```
[(u'73399', (24.0, 8)), (u'110555', (4.0, 1)), (u'73462', (120.0, 54)), (u'145208', (2.0, 1)), (u'89373', (29.0, 10))]
```


3. We need to divide the sum by the count and have the average rating for each movieID.

```
>>> finalResult = rdd.mapValues (lambda v: v[0]/v[1])
```

```
>>> finalResult.take(5)
```

```
[(u'73399', 3.0), (u'110555', 4.0), (u'73462', 2.222222222222223), (u'145208', 2.0), (u'89373', 2.8999999999999999)]
```

## Deliverables

1. Use the ratingsArray RDD and perform some basic operations. Explain what kind of question you are trying to answer with your operations. I want you to come up with three different questions that you will answer with basic operations (i.e., operations we did during the lab sessions. Of course, you may include other operations as well. You do not have to limit yourself to what we did in the lab.) (30 points)
  - Your solution should show for each of the three questions:
    - Question you want to answer
    - List of operations
    - Snapshot (of a subset) of the results
  - Example:
    - How many ratings does the movie with the movieID 2309 have?
    - `ratingsArray.filter(lambda line: line [1] == '2309').count()`
    - A screenshot of a Jupyter Notebook cell. The cell contains the code `ratingsArray.filter(lambda line: line [1] == '2309').count()`. The output of the cell is the number 45. The notebook interface includes a code editor area and an output area showing the result.
  - 10 Bonus points for the most interesting questions and solutions.
2. Use the ratingsArray RDD and count how many ratings each user made. The result should show 20 user with the most ratings. Like above, give me the list of operations and a snapshot of the results. (30 points)
3. Continuing with 2: show me only the users that have as many ratings as your number in your Virtual Machine username (e.g., for my VM with user 26, that would be 26). Use `collect()` to print the result. Like above, give me the list of operations and a snapshot of the results. (30 points)
4. 10 Bonus points: Use the ratingsArray RDD and give me the average ratings for each user. Like above, give me the list of operations and a snapshot of the results.
5. Write your reflection (200 to 250 words) on the discussion board (10 points)