Cloud Computing Application Capstone Task1

https://github.com/elnin09/CloudComputingCapstone/tree/master/Task02

All the spark code and cleaning code is present in this directory. Description of all the files present in Task2Readme

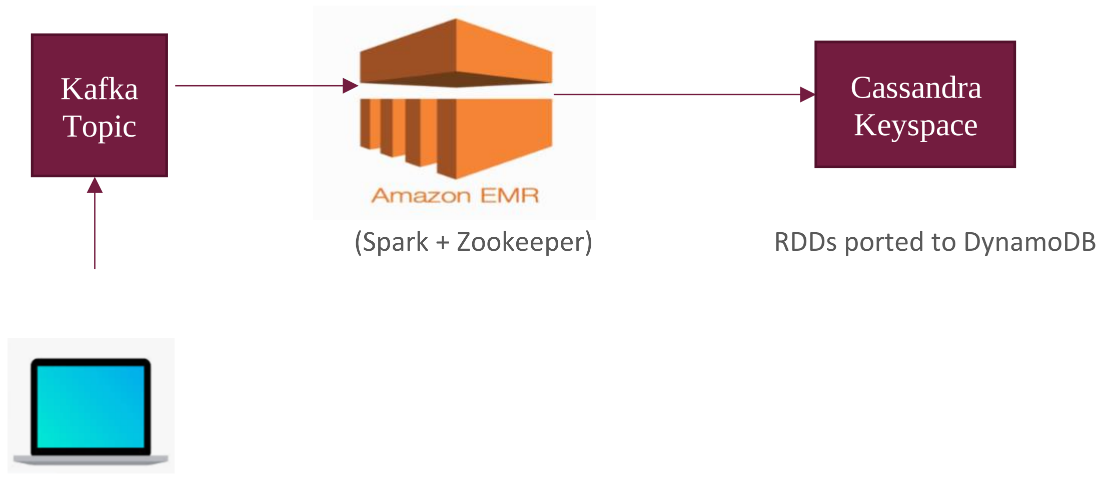
- ## Extracting and cleaning data

  The first task was to get the data from the EBS snapshot. It was achieved by attaching an EBS volume created using the EBS snapshot to the EC2 instance and then mounting it to get the required data files for this task.
  The dataset had a large number of files and for batch processing we had to combine them. In order to extract the data, I used a recursive python script to extract the data from the files. Once I had all the files in csv format, I then used pandas to clean the data on my local machine.

  1. The first task of cleaning process was to change the encoding of the files to UTF-8 so that it is easier to use frameworks that rely on UTF-8 encoding.
  2. Once encoding of all the files were changed, I used pandas to load the individual csv files into data frames. Once all the csv files were loaded into respective data frames, all the irrelevant columns were dropped from the data frames and only required columns were kept in the cleaned file.
  3. Once I had all the required data in different data frames, I concatenated the individual data frames into a single data frame.
  4. The data frame was then converted into a csv file with gzip compression and uploaded to Amazon S3.

- ## Integration of Systems

  Once I had all the required data in S3 bucket, the next task was to setup HDFS and this was done first locally using multiple docker containers and then executing the queries required for various tasks using Apache Spark. Post unit testing and getting the results locally, the setup was then migrated to AWS. On AWS an EMR cluster was setup with Spark and Zookeeper. The Kafka topic was also on EMR and I manually got the Kafka package on the EMR node. Post that a set of spark jobs were run to get the desired output by reading the stream from Kafka topic and was stored in Cassandra table hosted on another EC2 node.



Creating a kafka topic:

```
kafka-server-start /usr/local/etc/kafka/server.properties
kafka-topics --create --zookeeper localhost:2181 --replication-factor 3 --partitions 3 --topic testing12345
```

Feeding data from console:

```
 (your producer that outputs to console)|kafka-console-producer --bootstrap-server IP:9092 --topic testing123
```

Submitting spark job:

```
/usr/local/spark/bin/spark-submit --jars /usr/lib/spark/lib/datanucleus-api-jdo-3.2.6.jar,/usr/lib/spark/lib/datanucleus-core-3.2.10.jar,/usr/lib/spark/lib/datanucleus-rdbms-2.9.jar  --packages org.apache.spark:spark-streaming-kafka-0-8_2.11:2.2.0,com.datastax.spark:spark-cassandra-connector_2.11:2.0.0-M3 --conf spark.cassandra.connection.host=IP  sample1.py IP:2181 testing12345
```

• Solution approach

Group 1

1.1 Rank the top 10 most popular airports by numbers of flights to/from the airport.

The solution is to use standard word count approach where (source,1) and (destination,1) is emitted in map phase for every row and then we do a count using reduce by key using key(airport code).

1.2 Rank the top 10 airlines by on-time arrival performance.

1.3 Rank the days of the week by on-time arrival performance.

1.2 and 1.3 have identical solution approach where we need to calculate the average delay for airlines and day of week. So in this case for map reduce key is airline and day of week respectively and we emit (airline/dayofweek,[delay,1]). To calculate average we firstly add delay(summation of delay in value) and counts(which is done by summation of 1 in value) and then we compute the average using these two. Also since this is streaming rather than batch processing, we need to store values other than top 10 because if we receive more data than our top 10 values will change according to the existing data as well as the new data in the stream.

Results are as follows

Output 1.1	Output 1.2	Output 1.3
ORD,12449354	HA,-1.01180434575	6,4.30166992608
ATL,11540422	AQ,1.15692344248	2,5.99045998094
DFW,10799303	PS,1.45063851278	7,6.61328029244
LAX,7723596	ML (1),4.74760919573	1,6.71610280259
PHX,6585534	PA (1),5.32243099993	3,7.20365639467
DEN,6273787	F9,5.46588114882	4,9.09444100834
DTW,5636622	NW,5.55778339267	5,9.72103233759
IAH,5480734	WN,5.56077425988	
MSP,5199213	OO,5.73631246366	
SFO,5171023	9E,5.8671846617	

Group 2

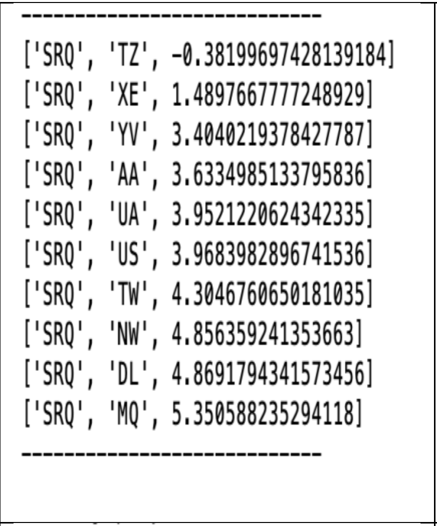
- 2.1 For each airport X, rank the top-10 carriers in decreasing order of on-time departure performance from X.
- 2.2 For each source airport X, rank the top-10 destination airports in decreasing order of on-time departure performance from X.

The approach is similar to question1. Here the key is combination of (airport,carrier) in 2.1 and (source,destination) in 2.2

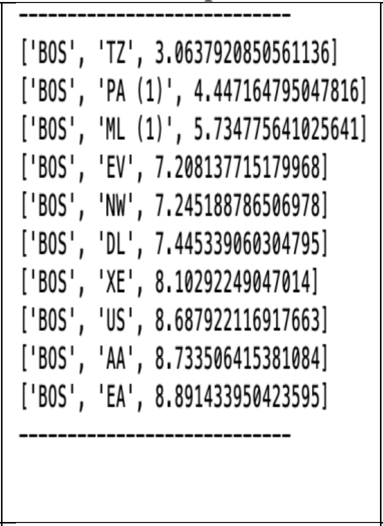
For 2.1 during the map phase, our key is ("airport,carrier",[departure delay,1]) for 2.1
As mentioned previously, we do reduce on airport,carrier key and get the average departure delay for every combination of airport,carrier. Once we get this, we can split the key from RDD and then save the data as Airport,Carrier and departure delay.

The same can be done for 2.2 where key is the only difference in our map reduce phase for Spark RDD

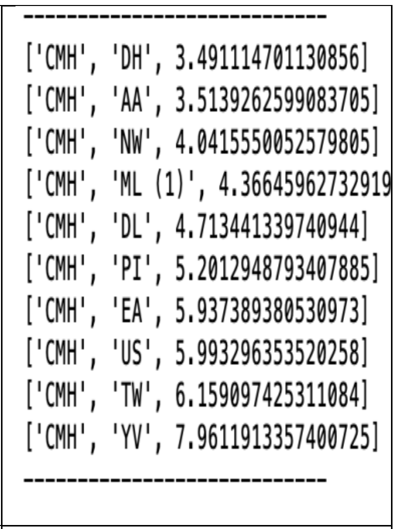
Output 2.1



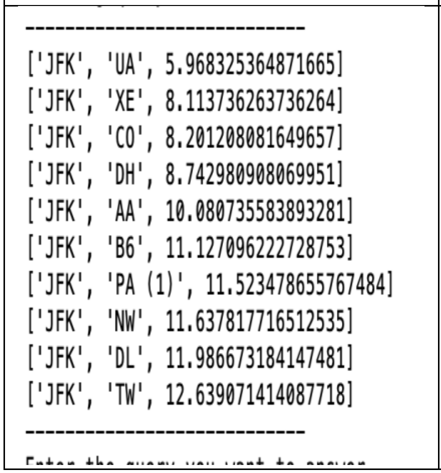
```
-----------------------------
['SRQ', 'TZ', -0.38199697428139184]
['SRQ', 'XE', 1.48976677777248929]
['SRQ', 'YV', 3.4040219378427787]
['SRQ', 'AA', 3.6334985133795836]
['SRQ', 'UA', 3.9521220624342335]
['SRQ', 'US', 3.9683982896741536]
['SRQ', 'TW', 4.3046760650181035]
['SRQ', 'NW', 4.856359241353663]
['SRQ', 'DL', 4.8691794341573456]
['SRQ', 'MQ', 5.350588235294118]
-----------------------------
```



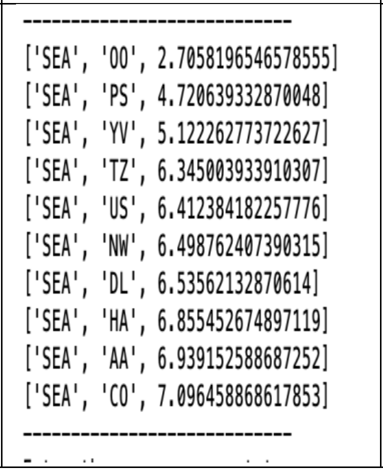
```
----------------------------
['BOS', 'TZ', 3.0637920850561136]
['BOS', 'PA (1)', 4.447164795047816]
['BOS', 'ML (1)', 5.734775641025641]
['BOS', 'EV', 7.208137715179968]
['BOS', 'NW', 7.245188786506978]
['BOS', 'DL', 7.445339060304795]
['BOS', 'XE', 8.10292249047014]
['BOS', 'US', 8.687922116917663]
['BOS', 'AA', 8.733506415381084]
['BOS', 'EA', 8.891433950423595]
----------------------------
```



```
----------------------------
['CMH', 'DH', 3.491114701130856]
['CMH', 'AA', 3.5139262599083705]
['CMH', 'NW', 4.0415550052579805]
['CMH', 'ML (1)', 4.36645962732919
['CMH', 'DL', 4.713441339740944]
['CMH', 'PI', 5.2012948793407885]
['CMH', 'EA', 5.937389380530973]
['CMH', 'US', 5.993296353520258]
['CMH', 'TW', 6.159097425311084]
['CMH', 'YV', 7.9611913357400725]
----------------------------
```

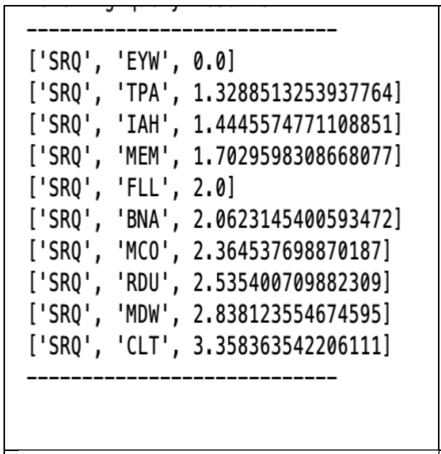


```
----------------------------
['JFK', 'UA', 5.968325364871665]
['JFK', 'XE', 8.113736263736264]
['JFK', 'CO', 8.201208081649657]
['JFK', 'DH', 8.742980908069951]
['JFK', 'AA', 10.080735583893281]
['JFK', 'B6', 11.127096222728753]
['JFK', 'PA (1)', 11.523478655767484]
['JFK', 'NW', 11.637817716512535]
['JFK', 'DL', 11.986673184147481]
['JFK', 'TW', 12.639071414087718]
----------------------------
```

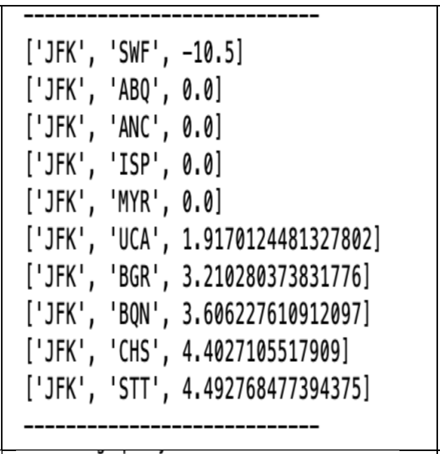


```
----------------------------
['SEA', 'OO', 2.7058196546578555]
['SEA', 'PS', 4.720639332870048]
['SEA', 'YV', 5.122262773722627]
['SEA', 'TZ', 6.345003933910307]
['SEA', 'US', 6.412384182257776]
['SEA', 'NW', 6.498762407390315]
['SEA', 'DL', 6.53562132870614]
['SEA', 'HA', 6.855452674897119]
['SEA', 'AA', 6.939152588687252]
['SEA', 'CO', 7.096458868617853]
----------------------------
```

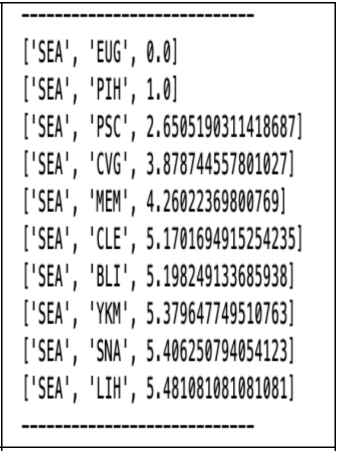
Output2.2



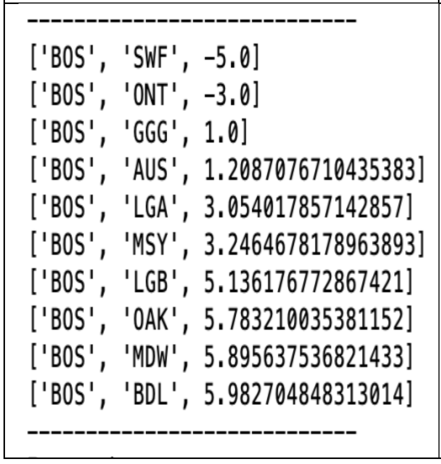
```
------------------------------
['SRQ', 'EYW', 0.0]
['SRQ', 'TPA', 1.3288513253937764]
['SRQ', 'IAH', 1.4445574771108851]
['SRQ', 'MEM', 1.7029598308668077]
['SRQ', 'FLL', 2.0]
['SRQ', 'BNA', 2.0623145400593472]
['SRQ', 'MCO', 2.364537698870187]
['SRQ', 'RDU', 2.535400709882309]
['SRQ', 'MDW', 2.838123554674595]
['SRQ', 'CLT', 3.358363542206111]
------------------------------
```



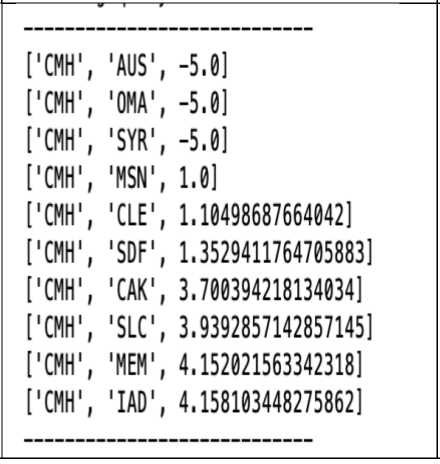
```
------------------------------
['JFK', 'SWF', -10.5]
['JFK', 'ABQ', 0.0]
['JFK', 'ANC', 0.0]
['JFK', 'ISP', 0.0]
['JFK', 'MYR', 0.0]
['JFK', 'UCA', 1.9170124481327802]
['JFK', 'BGR', 3.210280373831776]
['JFK', 'BQN', 3.606227610912097]
['JFK', 'CHS', 4.4027105517909]
['JFK', 'STT', 4.492768477394375]
------------------------------
```



```
------------------------------
['SEA', 'EUG', 0.0]
['SEA', 'PIH', 1.0]
['SEA', 'PSC', 2.6505190311418687]
['SEA', 'CVG', 3.878744557801027]
['SEA', 'MEM', 4.26022369800769]
['SEA', 'CLE', 5.1701694915254235]
['SEA', 'BLI', 5.198249133685938]
['SEA', 'YKM', 5.379647749510763]
['SEA', 'SNA', 5.406250794054123]
['SEA', 'LIH', 5.481081081081081]
------------------------------
```



```
------------------------------
['BOS', 'SWF', -5.0]
['BOS', 'ONT', -3.0]
['BOS', 'GGG', 1.0]
['BOS', 'AUS', 1.2087076710435383]
['BOS', 'LGA', 3.054017857142857]
['BOS', 'MSY', 3.2464678178963893]
['BOS', 'LGB', 5.136176772867421]
['BOS', 'OAK', 5.783210035381152]
['BOS', 'MDW', 5.895637536821433]
['BOS', 'BDL', 5.982704848313014]
------------------------------
```

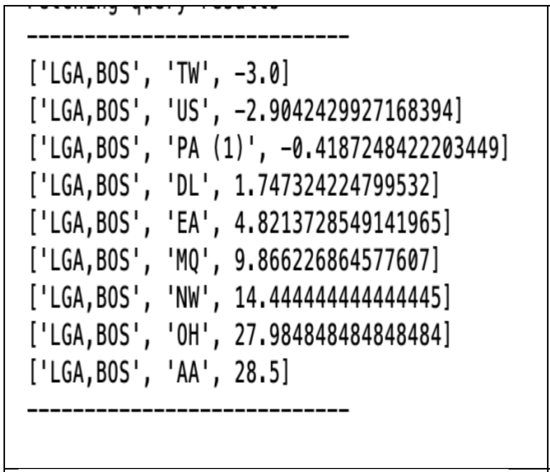


```
------------------------------
['CMH', 'AUS', -5.0]
['CMH', 'OMA', -5.0]
['CMH', 'SYR', -5.0]
['CMH', 'MSN', 1.0]
['CMH', 'CLE', 1.10498687664042]
['CMH', 'SDF', 1.3529411764705883]
['CMH', 'CAK', 3.700394218134034]
['CMH', 'SLC', 3.9392857142857145]
['CMH', 'MEM', 4.152021563342318]
['CMH', 'IAD', 4.158103448275862]
------------------------------
```

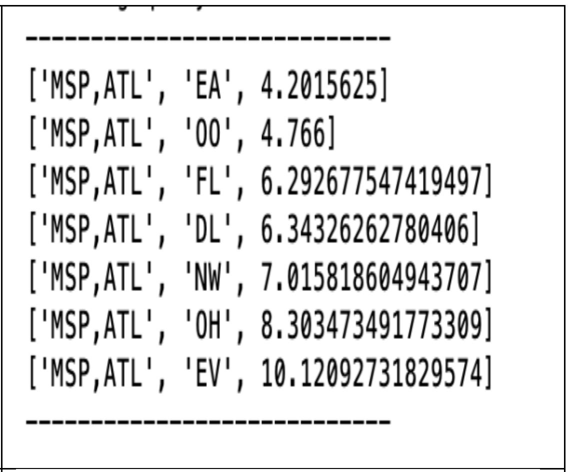
- 2.3For each source-destination pair X-Y, rank the top-10 carriers in decreasing order of on-time arrival performance at Y from X.

  The idea is similar to 2.1 and 2.2. In this case our key for initial map phase is "source,destination,carrier". So during initial map, we emit ("source,destination,carrier",[arrivaldelay,1])
  Now during reduce phase we get the average delay for each combination of source,destination,carrier. Now once the reduce phase is complete we can split the source,destination and carrier from the RDD and store the data into DB/Cassandra table.

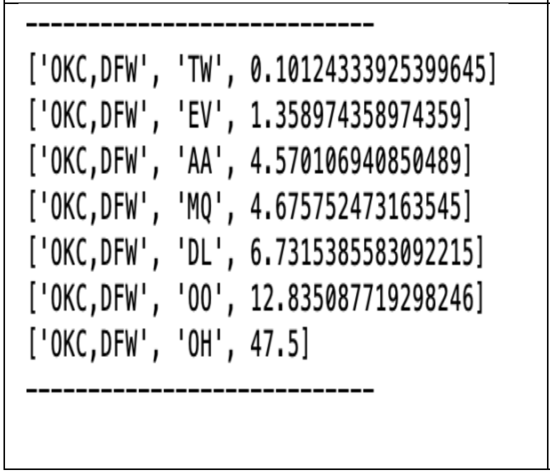
Results 2.3



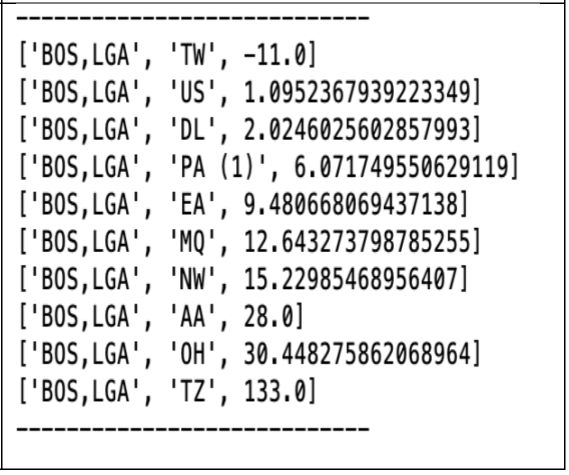
```
-----------------------------
['LGA,BOS', 'TW', -3.0]
['LGA,BOS', 'US', -2.9042429927168394]
['LGA,BOS', 'PA (1)', -0.4187248422203449]
['LGA,BOS', 'DL', 1.747324224799532]
['LGA,BOS', 'EA', 4.8213728549141965]
['LGA,BOS', 'MQ', 9.866226864577607]
['LGA,BOS', 'NW', 14.444444444444445]
['LGA,BOS', 'OH', 27.984848484848484]
['LGA,BOS', 'AA', 28.5]
-----------------------------
```



```
-----------------------------
['MSP,ATL', 'EA', 4.2015625]
['MSP,ATL', 'OO', 4.766]
['MSP,ATL', 'FL', 6.292677547419497]
['MSP,ATL', 'DL', 6.34326262780406]
['MSP,ATL', 'NW', 7.015818604943707]
['MSP,ATL', 'OH', 8.303473491773309]
['MSP,ATL', 'EV', 10.12092731829574]
-----------------------------
```



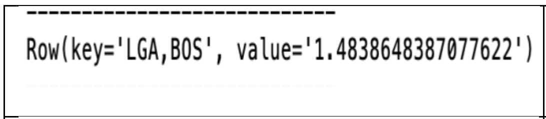
```
-----------------------------
['OKC,DFW', 'TW', 0.10124333925399645]
['OKC,DFW', 'EV', 1.358974358974359]
['OKC,DFW', 'AA', 4.570106940850489]
['OKC,DFW', 'MQ', 4.675752473163545]
['OKC,DFW', 'DL', 6.7315385583092215]
['OKC,DFW', 'OO', 12.835087719298246]
['OKC,DFW', 'OH', 47.5]
-----------------------------
```



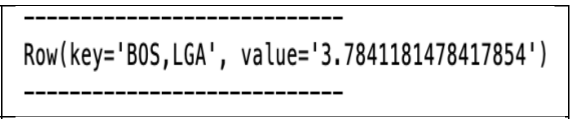
```
-----------------------------
['BOS,LGA', 'TW', -11.0]
['BOS,LGA', 'US', 1.0952367939223349]
['BOS,LGA', 'DL', 2.0246025602857993]
['BOS,LGA', 'PA (1)', 6.071749550629119]
['BOS,LGA', 'EA', 9.480668069437138]
['BOS,LGA', 'MQ', 12.643273798785255]
['BOS,LGA', 'NW', 15.22985468956407]
['BOS,LGA', 'AA', 28.0]
['BOS,LGA', 'OH', 30.448275862068964]
['BOS,LGA', 'TZ', 133.0]
-----------------------------
```

- 2.4 For each source-destination pair X-Y, determine the mean arrival delay (in minutes) for a flight from X to Y.

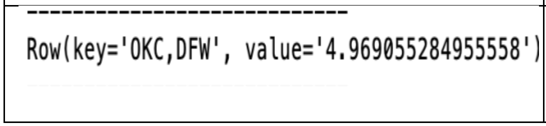
The approach for this is very straightforward and similar to previous queries.
Here the key is combination of “source,destination”.
So during our initial map phase, we emit (“source,destination,[arrivaldelay,1])
In our reduce phase we do summation of delays and counts and compute the average during reduce phase. Once that is done we can split the key and store the data in DB/cassandra



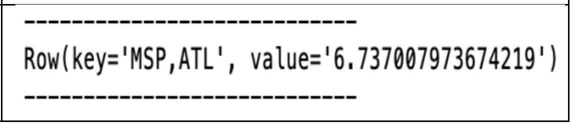
```
-----------------------------
Row(key='LGA,BOS', value='1.4838648387077622')
```



```
-----------------------------
Row(key='BOS,LGA', value='3.7841181478417854')
-----------------------------
```



```
-----------------------------
Row(key='OKC,DFW', value='4.969055284955558')
```



```
-----------------------------
Row(key='MSP,ATL', value='6.737007973674219')
-----------------------------
```

Group 3

- 3.1 Does the popularity distribution of airports follow a Zipf distribution? If not, what distribution does it follow?
  (Not discussed in Task2 as it is similar to Task1(batch processing)

- 3.2

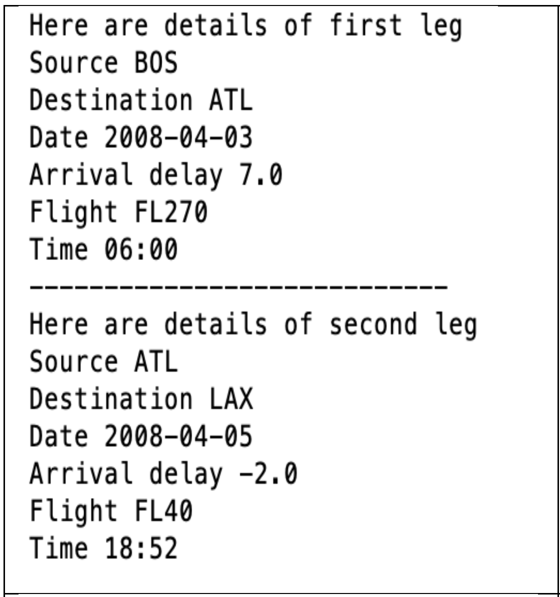
  For this task the approach was to first get all the flights with minimum delay for a unique combination of source, destination and date and with departure time before 12:00 and store the results in a table FirstLeg.
  In map phase we will emit (“source,stop,date”,[data]) data represents all the other details that is required for 3.2 for a particular flight. We will only be emitting where time is < 1200
  In reduce phase, we will get the best data for a particular key and will store it in a Cassandra table.

  Similarly get all the flights with minimum delay for a unique combination of source, destination and date and with departure time after12:00 and store the results in a temp table SecondLeg.
  In map phase we will emit (“stop,destination,date”,[data]). We will only be emitting where time is > 1200.
  In reduce phase, we will get the best data for a particular key and will store it in a Cassandra Table

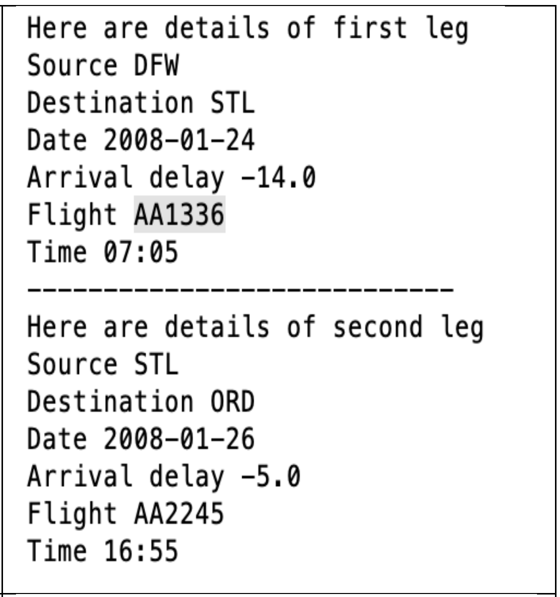
  Now instead of using a join on FirstLeg and SecondLeg(like Task 1), which will be a M*N operation the best approach would be to store the data for first leg and second leg in separate tables and upon a user query lookup data in Cassandra Table.
  Eg (source ,stop,destination,date) we can query source,stop,date in FirstLeg table and can query (stop,destination,date+2) we can query second table. Since the lookup will be in Cassandra key value pair the lookup time will be constant i.e. similar to lookup in one single table.

  The other reason to have two tables is because it’s not batch processing anymore, so any stream that changes any of the data in FirstLeg and SecondLeg would result in the change of data in the joined table which will be M*N with every stream and that would defeat the purpose of streaming and would be rather a batch processing. Also the amount of storage used for storing the data will be same in both the cases so we would prefer an approach that optimizes our pipeline.

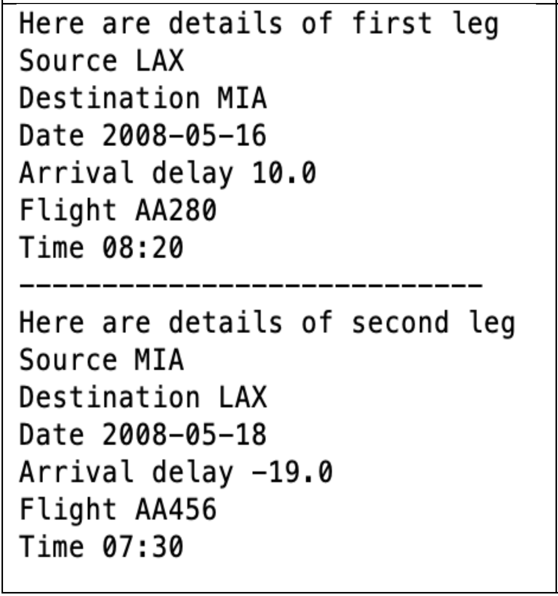
  Following is the output generated for the values in the query for Task 3.



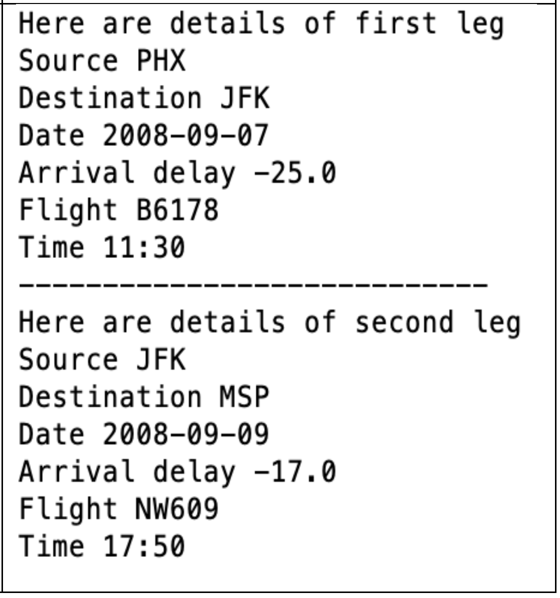
```
Here are details of first leg
Source BOS
Destination ATL
Date 2008-04-03
Arrival delay 7.0
Flight FL270
Time 06:00
----------------------------
Here are details of second leg
Source ATL
Destination LAX
Date 2008-04-05
Arrival delay -2.0
Flight FL40
Time 18:52
```



```
Here are details of first leg
Source DFW
Destination STL
Date 2008-01-24
Arrival delay -14.0
Flight AA1336
Time 07:05
----------------------------
Here are details of second leg
Source STL
Destination ORD
Date 2008-01-26
Arrival delay -5.0
Flight AA2245
Time 16:55
```



```
Here are details of first leg
Source LAX
Destination MIA
Date 2008-05-16
Arrival delay 10.0
Flight AA280
Time 08:20
----------------------------
Here are details of second leg
Source MIA
Destination LAX
Date 2008-05-18
Arrival delay -19.0
Flight AA456
Time 07:30
```



```
Here are details of first leg
Source PHX
Destination JFK
Date 2008-09-07
Arrival delay -25.0
Flight B6178
Time 11:30
----------------------------
Here are details of second leg
Source JFK
Destination MSP
Date 2008-09-09
Arrival delay -17.0
Flight NW609
Time 17:50
```

- Optimizations

1.1 The first optimization was to scale the number of EC2 nodes (worker nodes) on the basis of the spark job running on the server. This helped in bringing the execution time of all the queries by almost 50% in most of the cases and around 30% in case of query 3.2.

1.2 The data for output3_2 was huge as such this was a bottleneck in terms of migrating data to CassandraDB. So instead of having quorum, the consistency level can be changed to single node for the current Cassandra session so that writes are faster because only one replica needs to be updated.

1.3 The data fed to Kafka topic was pre cleaned so the message sent to topic consumes lesser amount of bandwidth and storage.

- Conclusion

Airlines rely on slots for arrival and departure. A potential delay is not feasible as it may further cause a delay and it may include in financial loss. So the data is useful for airlines as they can analyze the delays and financial losses that they have incurred because of these delays.
From passengers perspective, the results are useful as customers can choose flights conveniently according to their needs. They can also compare various airlines of different metrics which are more crucial to them.