Cloud Computing Project

1 Extraction and cleaning of the data

To extract the input data for queries, EC2 micro instance in us-east-1 region was created. Then EBS volume, created from snapshot snap-0d1e6d3ba6931e7c1 was mounted with command

```
mkdir ~/source && sudo mount /dev/xdvf ~/source
```

Folder ~/source has been investigated for usable data for chosen tasks. Data from folders airline_ontime/ was considered as useful, so folder ~/airline_ontime/ has been created and data from snapshot is copied with next command:

```
cd ~/airline_ontime/; find ~/source/aviation/airline_ontime  -name *.zip -exec unzip -o {} \;
```

After executing the command, folder ~/airline_ontime contains csv files with airlines statistics. To answer questions from group 1 and group 2 next fields in airline_ontime dataset are useful ArrDelayMinutes, DepDelayMinutes, DayOfWeek, UniqueCarrier, Origin, Dest. To retrieve only needed fields from csv files, csvcut command can be used, but some of the csv files contain wrongly encoded symbols, so csvcut failed. To fix this issue, first iconv command was used to fix encoding.

```
mkdir encoded
for f in ‘ls *.csv‘; do echo $f; iconv -f LATIN1 -t UTF8 $f >> encoded/$f; done
```

After files encoding was fixed, csv cut is executed in a next way.

```
cd encoded; for f in ‘ls *.csv‘; do echo $f; csvcut -c Origin,Dest,UniqueCarrier,\
DayOfWeek,DepDelayMinutes,ArrDelayMinutes $f >> ontime.dataset; done
```

After that all data is stored in ontime.dtaset csv file. To copy it to one of the cluster box, next command is used:

```
scp -i ../../virginia.pem ./ontime.dataset \
cloudbreak@ec2-34-232-27-247.compute-1.amazonaws.com:/home/cloudbreak/
```

To put data to HDFS, first RW access to corresponding folder should be granted, then data is trivially put to HDFS.

```
sudo runuser -l hdfs -c ’hadoop fs -chmod -R 666 /user/’
hadoop fs -mkdir /user/hadoop
hadoop fs -put /home/cloudbreak/ontime.dataset /user/hadoop/
```

2 Initial setup and integration

2.1 Strategy for cluster deployment, high-level design choices

Manual hadoop installation on AWS is quite time-consuming, also manually deployed cluster is hard to rescale and maintain. Among all possible ways to deploy Hadoop on AWS, Hortonworks Data Platform (HDP) stands out, it provides simple mechanism of Hadoop deployment, cluster can be rescaled easily. With HDP Hadoop clusters can be created on-demand. Instructions on how to deploy HDP via cloud formation is available in [1]. HDP deploys Apache Ambari as a tool for managing Hadoop cluster, it also supports Spark and provides access to hdfs via web interface. To make it easier to implement queries, Python programming language was selected. Hadoop job can be implemented in python, if hadoop streaming jar is used.

```
export HSTREAM=/usr/hdp/2.6.0.1-152/hadoop-mapreduce/hadoop-streaming.jar
```

The results are stored in DynamoDB, it’s easier to use DynamoDB service, than deploying own Cassandra instances. To be able to access Dynamo DB from Python, boto3 package has to be installed on the worker nodes of a cluster.

```
sudo /usr/local/bin/pip install boto3
```

To access DynamoDB from a terminal, DQL [2] can be used. DQL is CLI tool for accessing tables in DynamoDB. DQL syntax is similar to SQL. To make it possible to access Dynamo, aws_access_key_id and aws_secret_access_key have to be generated in AWS web console and then used to initialize the Boto3 session. On the box which is used to query results from Dynamo with DQL, set the environment variables AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY.

[1] https://hortonworks.github.io/hdp-aws/launch/

[2] https://github.com/stevearc/dql

2.2 Common code

To be able to use common code in python mappers or reducers, common code has to be archived to zip file and passed to -files CLI argument of hadoop command. Then modules can be imported by Python's zipimporter module functions. To make common modules available for mapper and reducer, they have to be zipped.

```
zip -r base.zip base_reducer.py base_mapper.py base_reducer2.py
```

2.3 Ingesting data to hadoop jobs

To feed input dataset to hadoop job, hadoop queries of the next form have been used:

```
hadoop jar $HSTREAM -files base.zip,mapper.py,reducer.py - input /user/hadoop/ontime.dataset \
 -output ./unique_name -mapper mapper.py -reducer reducer.py -numReduceTasks 1
```

3 Approaches and algorithms to answer each question

The selected queries are: rank 10 airlines and the days of week by on-time arrival performance, For each airport X rank the top-10 carriers and airports in decreasing order of on-time departure performance from X, For each source-destination pair X-Y, determine the mean arrival delay (in minutes) for a flight from X to Y.

3.1 Common approaches

First 5 questions are very similar: we are asked to provide best N results, for different objects (keys). In task 1 key is an airline, task 2 - a day of week, task 3 - a pair of origin airport and carrier, task 4 - origin airport and destination airport, task 5 - origin airport and destination airport. While solving queries, the average delay in minutes was considered as on-time performance. Negative delays were ignored. It is possible to extract common mapper and reducer parts for these tasks.

3.1.1 Common mapper

The idea of common mapper is to group input by the key and aggregaate sum of delays and number of records for the unique key. Technically it means next: we create dictionary, indexed by the tasks key, the values are pairs of sum of delays and counter of records for the key. Mapper takes CSV lines as input, parses it, using built-in python csv module, then delays are grouped by result of index_fn. The output of mapper is next: index, total number of delay minutes, number of delays processed per index. The tricky part is that for queries from group 2 we use keys, consisting of 2 elements, that can be done by concatenating elements forming one composite key. To achive this level of robustness, one of the base mapper entry's point argument is a function which input is a dictionary of current csv record and output is a key, so for tasks from group 1 these functions return just one element, for group 2 they return concatenation of corresponding two fields.

1. Common mapper code

   ```
   def process_row(d, m, make_index, delay_column):
       id = make_index(d)
       if id not in m:
           m[id] = [0.0, 0]
       rec = m[id]
       if d[delay_column]:
           rec[0] += float(d[delay_column])
           rec[1] += 1
   def process_input(delay_column, index_fn):
       with sys.stdin as csvfile:
           m = {}
           reader = csv.DictReader(csvfile)
           for row in reader:
               # skip header row
               if row['Origin'] == 'Origin':
                   continue
               process_row(row, m, index_fn, delay_column)
           for k in m.keys():
               print k, m[k][0], m[k][1]
   ```

2. Concrete mapper example

   First, base_mappr module should be imported, then function process_input is called: its first argument sets the name of the field of delay, second is a function which maps csv record to a key of the given mapper.

   ```
   importer = zipimport.zipimporter('base.zip')
   base_mappr = importer.load_module('base_mappr')
   base_mappr.process_input('ArrDelayMinutes', lambda d : d['UniqueCarrier'])
   ```

3.1.2 Common reducer

Reducer's job is trivial: to aggregate triplets of the form (key, sum, count) by the key and then calculate average value for each key, then sort and store in DynamoDB. Aggregation part is done, again by using a dictionary, which is indexed by the key and value is an average (sum/count). There is small difference between reducer used for group 2 and group 1: in group 2 queries keys are composite, so effectively they consist of two elements, so before writing a result to the database, we need to divide composite key back into parts and store in Dynamo as separate columns. Also, a small modification is done in reducer for the last query in group 2: the list of pairs of origin-destination is hardcoded to filter out airports which are not needed by the task description.

3.2 Group 3

Group 3 tasks differ from previous ones drastically. For the third group of tasks, it is more convenient to use spark, because spark programs are more compact and easier to run and debug. Spark program can use additional libraries in a more straightforward way. Also spark is much faster.

1. Does the popularity distribution of airports follow a Zipf distribution? If not, what distribution does it follow?

   For this task input data, used for group 1 and 2 can be reused. Airport popularity is calculated as number of departures from that airport. To generate sample distributions powerlaw python package has been used.

   ```
   df = spark.read.option("header", "true").csv('/user/hadoop/ontime.dataset')
   rows = df.groupBy('Origin').count().select('count').collect()
   powerlaw.plot_ccdf(numpy.random.uniform(low=1.0, high=100000.0, size=1000), color='red')
   powerlaw.plot_ccdf(numpy.random.lognormal(mean=10.50, sigma=2.0, size=1000), color='yellow')
   powerlaw.plot_ccdf(numpy.random.zipf(a=1.2, size=1000), color='brown')
   powerlaw.plot_ccdf(numpy.random.poisson(size=1000), color='blue')
   powerlaw.plot_ccdf(numpy.array([row['count'] for row in rows]), color='black')
   ```

   After plotting the data, its clear that the overall shape of the CCDF does not appear to follow a straight line, and thus we can reasonably reject the hypothesis that the data was drawn from a power-law (Zipfian) distribution simply using visual inspection. Rather, we can see that out of the three fitted distributions, log-normal with mean=10.50 and sigma=2.0 seems the best at modelling our data.

2. Tom wants to travel from airport X to airport Z. However, Tom also wants to stop at airport Y for some sightseeing on the way.

   For this task negative arrival delays were considered. To implement this query, additional fields, which are not in ontime.dataset are needed. There is also the requirement to use data only for the year 2008. Thus it was decided to use not cleaned csv files, all files corresponding to wildcard On_Time_On_Time_Performance_2008_* are deployed to HDFS the same way, ontime.dataset was. The basic algorithm for solving this task:

   (a) Create dataset (RDD) xs which contains all flights from X to Y, for which depart time less than 12:00 PM, year is 2008, the date equals to input date.

   (b) Create dataset (RDD) ys which contains all flights from Y to Z, on date equal input date plus 2 days, which depart time is greater than 12:00 PM and year is 2008.

   (c) Create cross product of two datasets, sort by the sum of delays.

4 Results for each query

To store results of each query, DynamoDB tables have been created, the indexes in that tables correspond to keys, used in mapper/reducer.

4.1 Results for group 2

4.1.1 Group 2 task 1

Airport	Carrier	AvgDepDelay	Airport	Carrier	AvgDepDelay
'BWI'	'F9'	4.91608391608	'CMI'	'US'	2.88274547188
'BWI'	'PA(1)'	5.94285714286	'CMI'	'TW'	4.15815384615
'BWI'	'CO'	7.1413334153	'CMI'	'PI'	4.52293031566
'BWI'	'AA'	7.65705490924	'CMI'	'OH'	5.36425479283
'BWI'	'YV'	7.67599067599	'CMI'	'DH'	9.64940239044
'BWI'	'NW'	8.30940419738	'CMI'	'EV'	9.69266055046
'BWI'	'US'	8.51417236303	'CMI'	'MQ'	11.7544899206
'BWI'	'DL'	8.81506807646			
'BWI'	'TW'	9.08485621193			
'BWI'	'EA'	9.17198697068			

Airport	Carrier	AvgDepDelay	Airport	Carrier	AvgDepDelay
'MIA'	'9E'	0.5	'LAX'	'PS'	4.9738958035
'MIA'	'PA(1)'	4.84346374454	'LAX'	'MQ'	5.0697457834
'MIA'	'EV'	5.66960352423	'LAX'	'OO'	6.09525787073
'MIA'	'XE'	6.10337698139	'LAX'	'ML(1)'	7.10127531883
'MIA'	'TZ'	6.82303539292	'LAX'	'NW'	7.25247915215
'MIA'	'NW'	6.99023545933	'LAX'	'TZ'	7.45686421605
'MIA'	'US'	7.42727823168	'LAX'	'US'	7.80373759019
'MIA'	'ML(1)'	7.63195146613	'LAX'	'FL'	8.0823277018
'MIA'	'UA'	8.27346848289	'LAX'	'F9'	8.36213813293
'MIA'	'PI'	9.06390283899	'LAX'	'AA'	8.4199274087
'IAH'	'PI'	4.64330450343	'SFO'	'PA(1)'	6.13963328632
'IAH'	'PA(1)'	5.7343030303	'SFO'	'PS'	6.43860728346
'IAH'	'NW'	6.15975939518	'SFO'	'TZ'	7.67880553532
'IAH'	'WN'	6.23224892212	'SFO'	'DL'	7.75330700463
'IAH'	'US'	7.05572327444	'SFO'	'NW'	7.94578766563
'IAH'	'AA'	7.26966230424	'SFO'	'MQ'	8.11433330068
'IAH'	'TW'	7.45336526342	'SFO'	'US'	8.57694968145
'IAH'	'OO'	7.94314970305	'SFO'	'TW'	8.67769505034
'IAH'	'HP'	8.04062547907	'SFO'	'CO'	8.8759604218
'IAH'	'DL'	8.27705795922	'SFO'	'F9'	8.91121495327

4.1.2 Group 2 task 2

Airport	AirportDest	AvgDepDelay	Airport	AirportDest	AvgDepDelay
'BWI'	'SAV'	0	'CMI'	'ABI'	0
'BWI'	'MLB'	2.38418079096	'CMI'	'PIT'	2.17013888889
'BWI'	'IAD'	3.08710801394	'CMI'	'DAY'	3.62729411765
'BWI'	'DAB'	3.83783783784	'CMI'	'STL'	4.01832669323
'BWI'	'SRQ'	4.26888536714	'CMI'	'PIA'	4.63243243243
'BWI'	'CHO'	4.82608695652	'CMI'	'CVG'	6.37942425672
'BWI'	'MDT'	4.90143084261	'CMI'	'DFW'	9.55624515128
'BWI'	'UCA'	4.93993879112	'CMI'	'ATL'	9.69266055046
'BWI'	'OAJ'	5.32	'CMI'	'ORD'	11.9431697613
'BWI'	'GSP'	5.43112513144			
'MIA'	'SHV'	0	'LAX'	'DRO'	0
'MIA'	'BUF'	1	'LAX'	'IDA'	0
'MIA'	'SAN'	2.51366120219	'LAX'	'LAX'	0
'MIA'	'HOU'	3.64113785558	'LAX'	'MAF'	0
'MIA'	'SLC'	4.07024793388	'LAX'	'PIH'	0
'MIA'	'ISP'	4.45664739884	'LAX'	'RSW'	0
'MIA'	'PSE'	4.94685990338	'LAX'	'SDF'	0
'MIA'	'MCI'	5.36054421769	'LAX'	'BZN'	1
'MIA'	'TLH'	5.44289663973	'LAX'	'VIS'	2.48051948052
'MIA'	'GNV'	6.00803212851	'LAX'	'PMD'	3
'IAH'	'MLI'	0	'SFO'	'FAR'	0
'IAH'	'MSN'	0	'SFO'	'PIH'	0
'IAH'	'HOU'	2.3019052956	'SFO'	'SDF'	0
'IAH'	'AGS'	2.83153347732	'SFO'	'MSO'	0.58333333333
'IAH'	'EFD'	3.91987363584	'SFO'	'LGA'	1.21212121212
'IAH'	'PIH'	4	'SFO'	'OAK'	2.54856787049
'IAH'	'VCT'	5.31756756757	'SFO'	'PIE'	2.72832369942
'IAH'	'RNO'	5.50723306544	'SFO'	'BNA'	3.06491611962
'IAH'	'MTJ'	5.63500784929	'SFO'	'SCK'	4
'IAH'	'MDW'	5.91583710407	'SFO'	'MEM'	5.43964855412

4.1.3 Group 2 task 3

```
us-east-1> SCAN * FROM MeanArrival;
```

MeanArrDelay	SrcDst	MeanArrDelay	SrcDst
13.6951942951	'LAX->SFO'	13.6202859298	'ATL->PHX'
6.61348747592	'IND->CMH'	15.7391506304	'CMI->ORD'
14.6323576467	'JFK->LAX'	11.0604620251	'DFW->IAH'

4.2 Results for group 3 task 2

```
us-east-1> SCAN * FROM Tom WHERE X='CMI' and Y='ORD' ORDER BY Delay;
```

```
|   Z   | YDelay | Delay |   Y   |   Flights    |       YDepart       |   X   | XDelay |       XDepart       |
| 'LAX' | -24    | -38   | 'ORD' | '4278->607'  | '2008-03-06 1950' | 'CMI' | -14    | '2008-03-04 0710' |
us-east-1> SCAN * FROM Tom WHERE X='JAX' and Y='DFW' ORDER BY Delay;
| 'CRP' | -7     | -6    | 'DFW' | '845->3627'  | '2008-09-11 1645' | 'JAX' | 1      | '2008-09-09 0725' |
us-east-1> SCAN * FROM Tom WHERE X='SLC'  ORDER BY Delay;
| 'LAX' | 6      | 18    | 'BFL' | '3755->5429' | '2008-04-03 1455' | 'SLC' | 12     | '2008-04-01 1100' |
us-east-1> SCAN * FROM Tom WHERE X='LAX' AND Y='SFO' ORDER BY Delay;
| 'PHX' | -19    | -32   | 'SFO' | '3534->412'  | '2008-07-14 1925' | 'LAX' | -13    | '2008-07-12 0650' |
us-east-1> SCAN * FROM Tom WHERE X='DFW' AND Y='ORD' ORDER BY Delay;
| 'DFW' | -10    | -31   | 'ORD' | '1104->2341' | '2008-06-12 1645' | 'DFW' | -21    | '2008-06-10 0700' |
us-east-1> SCAN * FROM Tom WHERE X='LAX' AND Y='ORD' ORDER BY Delay;
| 'JFK' | -7     | -7    | 'ORD' | '618->918'   | '2008-01-03 1900' | 'LAX' | 0      | '2008-01-01 0600' |
```

5 Optimizations

5.1 Optimization in group 2

In group 2 queries we are asked to provide results only for certain airports, filtering out non-required airports canimprove performance of the system. For example: it is known by requirement that mean arrival delay has to be calculated only for certain origin - destination pairs, so additional filtering can be applied to input data. If mapper filters out records, which contain non-required entries, then amount of network traffic and reducer input will be decreased.

5.2 Optimization in task 2 group 3

According to algorithm of solution for this task, described above, cross product is employed to generate all possible pairs of flights which correspond to criteria. Cross product is quadratic time algorithm, so its running time decreases redically in response to decreasing input size. To decrease input of cross-product, aggressive filtering of the input has been done, leaving only suitable entries in subsets for X-Y and Y-Z legs.

5.3 System-wide optimization

The first optimization that has been done is shrinking the size of input data by using csvcut command. Only useful fields are left in input, so network trafiic will be much less if full input files are used. Hortonworks Data Platform allows quick on-demand creation of Hadoop/Spark cluster, so there is no need to keep cluster running for the long period of time, cluster was summoned only for short period of time for show-casing work of algorithms and shooting video, thus amount of money spent was small, also this approach is more environment-friendly.

6 Conclusion

Obtained results are useful for analyzing which Airlines are underperforming in punctuality, this type of information can be used by policy-makers or compliance in order to optimize and improve service. The results can also be used by Airports to improve their services, by scaling their infrastructure according to the day of week. We've also found that popularity distribution of airports follows lognormal distribution, this means that airport popularity is the multiplicative product of many independent random variables. The distribution of an airport popularity is natural phenomena-like, so we can conclude that there are no major nonmarket force contributions to it.