Cloud Computing Capstone: Task 2 Report by Panagiotis Salteris

Data extraction and cleaning

I performed the following steps in order to extract and clean the data:

- By taking a look at the queries and the dataset profile, it was apparent that the only
- relevant table is *airline_ontime*
- Fetched the table data and running the script *move-to-localfs.sh* from the snapshot, unzipped the csv files to local "input" folder keeping only the columns which were needed to answer the questions
- As a result only the following columns were taken in the new csv files in the input folder:
    - *Year, Month, Quarter, DayOfWeek, DayofMonth, FlightDate, UniqueCarrier, FlightNum, Origin, Dest, CRSDepTime, DepTime, DepDelay, DepDelayMinutes, CRSArrTime, ArrTime, ArrDelay, ArrDelayMinutes, Cancelled, Diverted.*

Systems

The following systems were used to complete the task:

- an ec2 t2.medium instance was used to store the data and as the Kafka server ingesting the files in the Spark cluster.
- an ec2 t2.medium instance was used as a master node for the Hadoop system and as resource manager for the Yarn system
- three ec2 t2.large instances were used for the Spark Streaming cluster which was deployed upon the Yarn system containing 1 driver and 2 worker nodes
- DynamoDB was used to store the results of part 2 and 3.2

Algorithms and approaches

From the "input" folder in kafka node all lines of all files are ingested to the spark cluster through a producer with the following command:

```
$ awk FNR-1 /home/ubuntu/input/On_Time_On_Time_Performance_* | bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test1
```

1.1 Airport from-to information is collected by using flatMap from input stream

```
rows.flatMap(lambda row: [row[0], row[1]])
```

We use the updateStateByKey function with Spark checkpoints to count all the occurrences for all airports. The updateFunction is a simple counter function.

```
airports.map(lambda airport: (airport, 1)).updateStateByKey(updateFunction)
```

To reduce the traffic, we cut off the amount of records to just the top 10 most popular in each partition.

```
sorted.transform(lambda rdd: rdd.mapPartitions(cutOffTopTen))
def cutOffTopTen(iterable):
        topTen = []
        for tupl in iterable:
                if len(topTen) < 10:
                        topTen.append(tupl)
                        topTen.sort(reverse=True)
                elif topTen[9][0] < tupl[0]:
```

```
                topTen[9] = tupl
                topTen.sort(reverse=True)
        return iter(topTen)
```

Then we sort RDDs by popularity.

```
sorted = sorted.transform(lambda rdd: rdd.sortByKey(False))
```

"ORD" 12449354
"ATL" 11540422
"DFW" 10799303
"LAX" 7723596
"PHX" 6585534
"DEN" 6273787
"DTW" 5636622
"IAH" 5480734
"MSP" 5199213
"SFO" 5171023

1.2 This is analogue to Question 1.1

"HA" -1.01
"AQ" 1.16
"PS" 1.45
"ML (1)" 4.75
"PA (1)" 5.32
"F9" 5.47
"NW" 5.56
"WN" 5.56
"OO" 5.74
"9E" 5.87

2.1 First we use the updateStateByKey function with Spark checkpoints to count average departure delays for all airport-carrier pairs. The updateFunction calculates three values for each: sum, count and sum/count.

```
airports_and_carriers.updateStateByKey(updateFunction)
def updateFunction(newValues, runningAvg):
        if runningAvg is None:
                runningAvg = (0.0, 0, 0.0)
        # calculate sum, count and average.
        prod = sum(newValues, runningAvg[0])
        count = runningAvg[1] + len(newValues)
        avg = prod / float(count)
        return (prod, count, avg)
```

Then we use the aggregateByKey to have an ordered list of top ten performing carrier for each airport. This is tricky at first, but keeps calculations and data traffic at minimum. Aggregate contains top ten carriers and departure delays. Sample aggregated value for an airport:

[('TZ',-0.0001), ('AQ',0.025), ('MS',0.3)]

```
airports = airports.transform(lambda rdd: rdd.aggregateByKey([],append,combine))
def append(aggr, newCarrierAvg):
        """
        Add new element to aggregate. Aggregate contains top ten carriers and departure delays.
        Sample: [('TZ',-0.0001), ('AQ',0.025), ('MS',0.3)]
```

```
        """
        aggr.append(newCarrierAvg)
        aggr.sort(key=lambda element: element[1])
        return aggr[0:10]
def combine(left, right):
        """
        Combine two aggregates. Aggregate contains top ten carriers and departure delays.
        Sample: [('TZ',-0.0001), ('AQ',0.025), ('MS',0.3)]
        """
        for newElement in right:
                left.append(newElement)
        left.sort(key=lambda element: element[1])
        return left[0:10]
```

Then the results are stored into DynamoDB table. Finally with a python script and the use of boto3 module the data for specific keys were retrieved and sorted.

```
--- SRQ ---
    (TZ: -0.38199697428139183055975794251134 64)
    (XE: 1.48976677772489290813898143741075 7)
    (YV: 3.40402193784277879341864716636197 4)
    (AA: 3.63349851337958374628344895936570 9)
    (UA: 3.95212206243423360224482637670992 6)
    (US: 3.96839828967415343785324617879785 7)
    (TW: 4.30467606501810338186580386719050 9)
    (NW: 4.85635924135366307177389364075864 6)
    (DL: 4.86917943415734561518939749976501 6)
    (MQ: 5.35058823529411764705882352941176 5)

--- CMH ---
    (DH: 3.49111470113085621970920840064620 4)
    (AA: 3.51392625990837030034179332412188 2)
    (NW: 4.04155500525798025713219580040028 5)
    (ML (1): 4.36645962732919254658385093167701 9)
    (DL: 4.71344133974094436826901316577696 7)
    (PI: 5.20129487934078869923484402589758 7)
    (EA: 5.93738938053097345132743362831858 4)
    (US: 5.99329635352025868983392855635200 1)
    (TW: 6.15909742531108435116692470898560 7)
    (YV: 7.96119133574007220216606498194945 8)

--- JFK ---
    (UA: 5.96832536487166582788122798188223 5)
    (XE: 8.11373626373626373626373626373626 4)
    (CO: 8.20120808164965632159966673609664 7)
    (DH: 8.74298090806995026472003850473287 3)
    (AA: 10.08073558389328018887826598261135)
    (B6: 11.12709622272875284923477694562032)
    (PA (1): 11.52347865576748410535876475930972)
    (NW: 11.63781771651253417088349909847031)
    (DL: 11.98667318414748128014782505935563)
    (TW: 12.63907141408771804834740862931644)

--- SEA ---
    (OO: 2.70581965465785546791728842464293 3)
    (PS: 4.72063933287004864489228630993745 7)
    (YV: 5.12226277372262773722627737226277 4)
    (TZ: 6.34500393391030684500393391030684 5)
    (US: 6.41238418225777662388238921252048 6)
```

```
    (NW: 6.498762407390315732948490976062022)
    (DL: 6.535621328706140038970828770143225)
    (HA: 6.855452674897119341563786008230453)
    (AA: 6.939152588687251860543616285985244)
    (CO: 7.096458868617853503093286634992177)

--- BOS ---
    TZ: 3.063792085056113408151210868281158)
    (PA (1): 4.447164795047815562874489107575096)
    (ML (1): 5.734775641025641025641025641025641)
    (EV: 7.208137715179968701095461658841941)
    (NW: 7.245188786506978788111297894156791)
    (DL: 7.445339060304794998015936853168602)
    (XE: 8.102922490470139771283354510800508)
    (US: 8.687922116917663676693859771544086)
    (AA: 8.733506415381083926324038088604289)
    (EA: 8.891433950423595858173831189206150)
```

2.2 This is analogue to Question 2.1 but instead of Carriers the destination airports are used.

```
--- JFK ---
(SWF: -10.5)
(MYR: 0.0)
(ABQ: 0.0)
(ISP: 0.0)
(ANC: 0.0)
(UCA: 1.91701244813)
(BGR: 3.21028037383)
(BQN: 3.60622761091)
(CHS: 4.40271055179)
(STT: 4.50210682155)

--- SEA ---
(EUG: 0.0)
(PIH: 1.0)
(PSC: 2.65051903114)
(CVG: 3.8787445578)
(MEM: 4.26022369801)
(CLE: 5.17016949153)
(BLI: 5.19824913369)
(YKM: 5.37964774951)
(SNA: 5.40625079405)
(LIH: 5.48108108108)

--- BOS ---
(SWF: -5.0)
(ONT: -3.0)
(GGG: 1.0)
(AUS: 1.20870767104)
(LGA: 3.05401785714)
(MSY: 3.2464678179)
(LGB: 5.13617677287)
(OAK: 5.78321003538)
(MDW: 5.89563753682)
(BDL: 5.98270484831)

--- CMH ---
(SYR: -5.0)
(AUS: -5.0)
(OMA: -5.0)
```

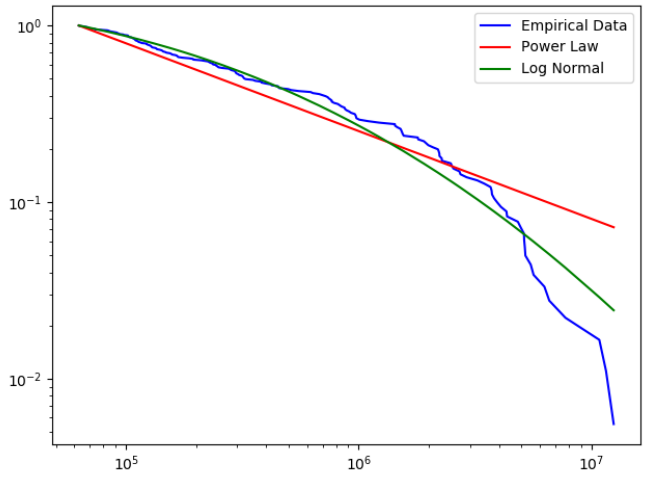
```
(MSN: 1.0)
(CLE: 1.10498687664)
(SDF: 1.35294117647)
(CAK: 3.70039421813)
(SLC: 3.93928571429)
(MEM: 4.15202156334)
(IAD: 4.15810344828)

--- SRQ ---
(EYW: 0.0)
(TPA: 1.32885132539)
(IAH: 1.44455747711)
(MEM: 1.70295983087)
(FLL: 2.0)
(BNA: 2.06231454006)
(MCO: 2.36453769887)
(RDU: 2.53540070988)
(MDW: 2.83812355467)
(CLT: 3.35836354221)
```

2.4 For each origin destination pair, compute mean arrival delay: We calculate the mean arrival delay for all the airport from-to pairs. The average calculation method is the same as in Question 2.1.
`airports_fromto = airports_fromto.updateStateByKey(updateFunction)`
The output was stored in dynamodb and with another python script the following results were retrieved:

The query for LGA returned the following items:
LGA -> BOS: 1.48386483871
The query for BOS returned the following items:
BOS -> LGA: 3.78411814784
The query for OKC returned the following items:
OKC -> DFW: 5.07023374012
The query for MSP returned the following items:
MSP -> ATL: 6.73700797367

3.1 Total frequency (to and from) of flights from each airport was calculated and ranked in descending order. This gave the frequency count as well as the rank needed to plot the curve. The CCDF of the popularity for airports looks like the following:



The CCDF of power-law distributions should be a straight line. Also the lognormal distribution fits better to the empirical data. So, the popularity of the airports definitely doesn't follow Zipf distribution.
Log likelihood ration tests gives us the following results, when comparing the fitted power-law and lognormal distributions:
R -3.485522, p 0.000491
As R is negative, the empirical data more likely follows a log normal distribution.

3.2 At first step we map every item from input_2008 topic and form a key that holds the airport from-to, flight date, and AM or PM according to what is the SCHEDULED_DEPARTURE_TIME of the flight.

```
airports_fromto = rows.map(lambda row: ((row[8], row[9], row[5], AMOrPM(row[10])), (row[6], row[7], departureTimePretty(row[10]), float(row[16]))))
```

Next, we filter out all unnecessary data that is not relevant for answering question 3.2 and do a minimum search for each key. Minimum search is based on the arrival performance of the given flight. This way for each key-value pair we just keep tracking of the best flights.

```
# Filtering just necessary flights
airports_fromto = airports_fromto.filter(lambda row: row[0] == ('BOS', 'ATL', '2008-04-03', 'AM')) \
.union(airports_fromto.filter(lambda row: row[0] == ('ATL', 'LAX', '2008-04-05', 'PM'))) \
.union(airports_fromto.filter(lambda row: row[0] == ('PHX', 'JFK', '2008-09-07', 'AM'))) \
.union(airports_fromto.filter(lambda row: row[0] == ('JFK', 'MSP', '2008-09-09', 'PM'))) \
.union(airports_fromto.filter(lambda row: row[0] == ('DFW', 'STL', '2008-01-24', 'AM'))) \
.union(airports_fromto.filter(lambda row: row[0] == ('STL', 'ORD', '2008-01-26', 'PM'))) \
.union(airports_fromto.filter(lambda row: row[0] == ('LAX', 'MIA', '2008-05-16', 'AM'))) \
.union(airports_fromto.filter(lambda row: row[0] == ('MIA', 'LAX', '2008-05-18', 'PM')))
# Minimum search
airports_fromto = airports_fromto.updateStateByKey(getMinimum)
```

Results are saved to Kafka topic and then to Cassandra by a different Spark Streaming job. The output was stored in dynamodb and with another python script the following results were retrieved:

Tom wants to fly from " BOS " to " ATL " at 2008-04-03 and from " ATL " to "LAX" at 2008-04-05:
"BOS" -> "ATL" on 2008-04-03: Flight: "FL" "270" at 06:00. Arrival Delay: 7.0
"ATL" -> "LAX" on 2008-04-05: Flight: "FL" "40" at 18:52. Arrival Delay: -2.0
Total arrival delay: 5.0

Tom wants to fly from " PHX" to " JFK " at 2008-09-07 and from " JFK" to " MSP" at 2008-09-09:
"PHX" -> "JFK" on 2008-09-07: Flight: "US" "12" at 09:04. Arrival Delay: -25.0
"JFK" -> "MSP" on 2008-09-09: Flight: "NW" "609" at 17:50. Arrival Delay: -17.0
Total arrival delay: -42.0

Tom wants to fly from " DFW" to " STL" at 2008-01-24 and from " STL" to "ORD" at 2008-01-26:
"DFW" -> "STL" on 2008-01-24: Flight: "AA" "314" at 08:40. Arrival Delay: -14.0
"STL" -> "ORD" on 2008-01-26: Flight: "AA" "640" at 14:55. Arrival Delay: -5.0
Total arrival delay: -19.0

Tom wants to fly from "LAX" to "MIA" at 2008-05-16 and from "MIA" to " LAX" at 2008-05-18:
"LAX" -> "MIA" on 2008-05-16: Flight: "AA" "202" at 07:10. Arrival Delay: 10.0
"MIA" -> "LAX" on 2008-05-18: Flight: "AA" "203" at 15:35. Arrival Delay: -19.0
Total arrival delay: -9.0