CS 598 Cloud Computing Capstone Task 1 Report

1. Data mounting, extraction and cleaning

The EBS snapshot ID for the transportation dataset is given (snap-e1608d88). To use the data, a new volume was created with size 100 GB (a default 8 GB was chosen, however it has not enough space to store the extracted dataset), availability zone us-east-1b, and the given snapshot id. Then, a new instance is launched and attached with the volume created.

To mount the attached volume, follow link and do:

```
sudo mkdir /data
sudo mount /dev/xvdf /data
```

As seen from the dataset and the project requirement, only the *airline_ontime* folder from aviation portion of the dataset is needed. Therefore, the files from *airline_ontime* folder are extracted and put into hdfs folder, using bash scripts:

```
for FILE in `ls $DATA/airline_ontime/*/*.zip`; do
    for CSV_FILE in `unzip -l $FILE | grep csv | tr -s ' ' | cut -d ' ' -f4`; do
        unzip -p $FILE $CSV_FILE | $HADOOP_HOME/bin/hdfs dfs -put - $HDFS_TARGET_FOLDER/$CSV_FILE
    done
done
```

2. Overview of the approaches to solve the questions

In Task 1, Java, Hadoop Yarn, Spark, Cassandra are installed and used as illustrated in below diagram.

- To install Hadoop Yarn cluster, the steps in Documentation and link are followed.
- To install Spark, the steps in link are followed. Notice that based on compatibility of the Spark Cassandra Connector, the version of Spark has been installed as 2.1 rather than the latest version.
- To install Cassandra, the steps in link are followed.
- To setup Spark Cassandra Connection, the steps in link are followed.

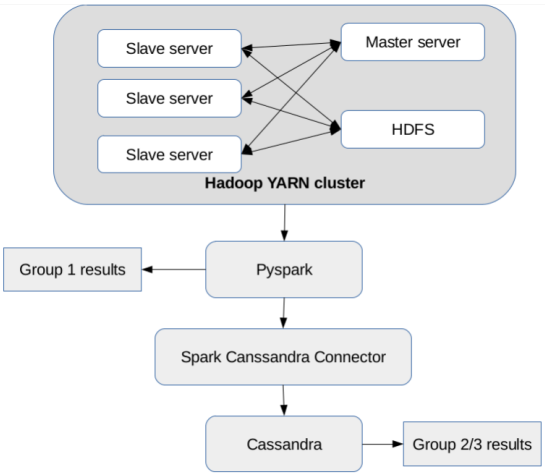


Figure 1. System structure of Task 1

As seen from above diagram, the main work, including data analysis and simplification, has been done using map-reduce in Hadoop Yarn. The simplified data are then further processed using Spark.

- For Group 1 questions, results will be directly generated by PySpark.
- For Group 2 and 3 questions, Cassandra database has been used for data query. The Spark Cassandra Connector is used for setting up Spark application to read/write data from/to Cassandra.

3. Steps and results of each question

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

A MapReduce framework is used to firstly process and simplify the data:

- In the Mapper method, each line is processed at a time, get splitted into tokens by “,”. The value at the column ‘Origin’ and ‘Dest’ is extracted and emitted as key-value pair of <<airport name>, 1>.
- The Reducer method sums up the values under the same key, and output the results into file.

The MapReduce implementation class is compiled and jar file has been created. The MapReduce execution is then done using:

```
$HADOOP_HOME/bin/hadoop jar ../artifacts/test.jar test.src.TenMostPopularAirports /origin ten_most_popular_airports
```

To further process the data and get the result, PySpark is used:

```
file = sc.textFile('hdfs://localhost:9000/user/ec2-user/popular_airports/part-r-00000/part-r-00000')
rdd = file.cache()
rdd = rdd.map(lambda line: line.split()).cache()
rdd2 = rdd.map(lambda tuple: (float(tuple[1]), tuple[0])).cache()
rdd2.takeOrdered(10)
```

The result is:
[(12446097, u'ORD'), (11537401, u'ATL'), (10795494, u'DFW'), (7721141, u'LAX'), (6582467, u'PHX'), (6270420, u'DEN'), (5635421, u'DTW'), (5478257, u'IAH'), (5197649, u'MSP'), (5168898, u'SFO')]

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

Similar to Question 1.1, MapReduce framework is used to firstly process and simplify the data:

- In the Mapper method, each line is processed at a time, get splitted into tokens by “,”. The value at the column ‘Carrier’ and ‘ArrDel15’ is extracted and emitted as key-value pair of <<Carrier>, Arrival delay in double>.
- The Reducer method sums up the values under the same key, calculate the average, and output the results into file.

The MapReduce implementation class is compiled and jar file has been created. The MapReduce execution is then done using:

```
$HADOOP_HOME/bin/hadoop jar ../artifacts/test.jar test.src.TenOnTimeArriavalAirports /origin ten_on_time_arrival_airport
```

To further process the data and get the result, PySpark is used:

```
file = sc.textFile('hdfs://localhost:9000/user/ec2-user/popular_airports/part-r-00000/part-r-00000')
rdd = file.cache()
rdd = rdd.map(lambda line: line.split()).cache()
rdd2 = rdd.map(lambda tuple: (float(tuple[1]), tuple[0])).cache()
rdd2.takeOrdered(10)
```

The result is:
[(-1.01, u'HA'), (1.16, u'AQ'), (1.45, u'PS'), (4.75, u'ML'), (5.35, u'PA'), (5.47, u'F9'), (5.56, u'NW'), (5.56, u'WN'), (5.74, u'OO'), (5.87, u'9E')

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

For MapReduce implementation, since information including airport, carrier, departure delay needs to be used and analyzed, a custom key class that implement the WritableComparable interface is created to include both the airport and carrier information. The emitted key-value pair of <<airport, carrier>, departure delay> from the Mapper method is given to Reducer method, and the average departure delay is obtained and written into file for each key.

To further process and query the data, Spark and Cassandra is used. Spark is to further process the data, and write into Cassandra for easier query.

A table is created in Cassandra cqlsh:

```
apache-cassandra-3.11.2/bin/cassandra
apache-cassandra-3.11.2/bin/cqlsh $(hostname -i)

create keyspace result WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 1 };
```

```
create table result.airport_carrier_departuredelay (
  airport text,
  carrier text,
  departure_delay decimal,
  PRIMARY KEY(airport, departure_delay, carrier)
);
```

PySpark is then executed together with Spark-Cassandra_Connector:

```
$SPARK_HOME/bin/pyspark --conf spark.cassandra.connection.host=$(hostname -i) --packages datastax:spark-cassandra-connector:2.0.0-RC1-s_2.11
```

To process data in PySpark and write into Cassandra:

```
file = sc.textFile('hdfs://localhost:9000/user/ec2-user/popular_airports/part-r-00000/part-r-00000')
rdd = file.map(lambda line: line.split())
rdd2 = rdd.map(lambda tuple: (tuple[0], tuple[1], float(tuple[2])))
from pyspark.sql import Row
rdd3 = rdd2.map(lambda row: Row(airport=row[0], carrier=row[1], dep_delay=row[2]))
df = spark.createDataFrame(rdd3)
df.write\
    .format("org.apache.spark.sql.cassandra")\
    .mode('append')\
    .options(table="airport_carrier_departuredelay", keyspace="result")\
    .save()
```

To query the result in Cassandra cqlsh:

```
select * from result.airport_carrier_departuredelay where airport = 'CMI' order by departure_delay limit 10;
```

The given query result is:

airport	dep_delay	carrier
CMI	0.61	OH
CMI	2.03	US
CMI	4.12	TW
CMI	4.46	PI
CMI	6.03	DH
CMI	6.67	EV
CMI	8.02	MQ

airport	dep_delay	carrier
BWI	0.76	F9
BWI	4.76	PA
BWI	5.18	CO
BWI	5.5	YV
BWI	5.71	NW
BWI	5.75	AL
BWI	6.0	AA
BWI	7.24	9E
BWI	7.5	US
BWI	7.68	DL

airport	dep_delay	carrier
MIA	-3.0	9E
MIA	1.2	EV
MIA	1.3	RU
MIA	1.78	TZ
MIA	2.75	XE
MIA	4.2	PA
MIA	4.5	NW
MIA	6.06	US
MIA	6.87	UA
MIA	7.5	ML

airport	dep_delay	carrier
LAX	1.95	RU
LAX	2.41	MQ
LAX	4.22	OO
LAX	4.73	FL
LAX	4.76	TZ
LAX	4.86	PS
LAX	5.12	NW
LAX	5.73	F9
LAX	5.81	HA
LAX	6.02	YV

airport	dep_delay	carrier
IAH	3.56	NW
IAH	3.98	PA
IAH	3.99	PI
IAH	4.8	RU
IAH	5.06	US
IAH	5.1	AL
IAH	5.55	F9
IAH	5.71	AA
IAH	6.05	TW
IAH	6.23	W

airport	dep_delay	carrier
SFO	3.95	TZ
SFO	4.85	MQ
SFO	5.16	F9
SFO	5.29	PA
SFO	5.76	NW
SFO	6.3	PS
SFO	6.56	DL
SFO	7.08	CO
SFO	7.4	US
SFO	7.79	TW

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

Similar to Question 2.1, the MapReduce implementation is executed that emitted key-value pair of <<departure airport, arrival airport>, departure delay> from the Mapper method is given to Reducer method, and the average departure delay is obtained and written into file for each key.

A table is created in Cassandra for queries, and Spark is used for rearranging the data form the Reducer method output.

The given query result is:

airport	dep_delay	airport_to
CMI	-7.0	ABI
CMI	1.1	PIT
CMI	1.89	CVG
CMI	3.12	DAY
CMI	3.98	STL
CMI	4.59	PIA

airport	dep_delay	airport_to
BWI	-7.0	SAV
BWI	1.16	MLB
BWI	1.47	DAB
BWI	1.59	SRQ
BWI	1.79	IAD
BWI	3.65	UCA

airport	dep_delay	airport_to
MIA	0.0	SHV
MIA	1.0	BUF
MIA	1.71	SAN
MIA	2.54	SLC
MIA	2.91	HOU
MIA	3.65	ISP

CMI | 5.94 | DFW CMI | 6.67 | ATL CMI | 8.19 | ORD	BWI | 3.74 | CHO BWI | 4.2 | GSP BWI | 4.45 | SJU BWI | 4.47 | OAJ	MIA | 3.75 | MEM MIA | 3.98 | PSE MIA | 4.26 | TLH MIA | 4.61 | MCI
airport | dep_delay | airport_to ---------+-----------+----------- LAX | -16.0 | SDF LAX | -7.0 | IDA LAX | -6.0 | DRO LAX | -3.0 | RSW LAX | -2.0 | LAX LAX | -0.73 | BZN LAX | 0.0 | MAF LAX | 0.0 | PIH LAX | 1.27 | IYK LAX | 1.38 | MFE	airport | dep_delay | airport_to ---------+-----------+----------- IAH | -2.0 | MSN IAH | -0.62 | AGS IAH | -0.5 | MLI IAH | 1.89 | EFD IAH | 2.17 | HOU IAH | 2.57 | JAC IAH | 2.95 | MTJ IAH | 3.22 | RNO IAH | 3.6 | BPT IAH | 3.61 | VCT	airport | dep_delay | airport_to ---------+-----------+----------- SFO | -10.0 | SDF SFO | -4.0 | MSO SFO | -3.0 | PIH SFO | -1.76 | LGA SFO | -1.34 | PIE SFO | -0.81 | OAK SFO | 0.0 | FAR SFO | 2.43 | BNA SFO | 3.3 | MEM SFO | 4.0 | SCK

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

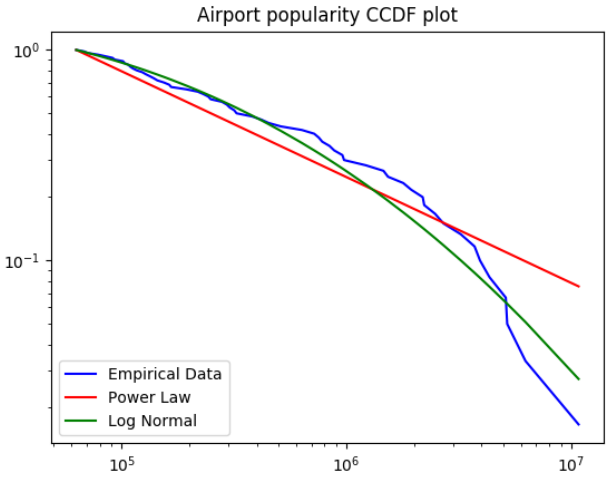
Similar to Question 2.2, the MapReduce implementation is executed that emitted key-value pair of <<departure airport, arrival airport>, arrival delay> from the Mapper method is given to Reducer method, and the average departure delay is obtained and written into file for each key.

A table is created in Cassandra for queries, and Spark is used for rearranging the data form the Reducer method output.

The given query result is:

airport | airport_to | arr_delay ---------+------------+----------- CMI | ORD | 10.14	airport | airport_to | arr_delay ---------+------------+----------- IND | CMH | 2.89	airport | airport_to | arr_delay ---------+------------+----------- DFW | IAH | 7.62
airport | airport_to | arr_delay ---------+------------+----------- LAX | SFO | 9.59	airport | airport_to | arr_delay ---------+------------+----------- JFK | LAX | 6.64	airport | airport_to | arr_delay ---------+------------+----------- ATL | PHX | 9.02

Question 3.1



The popularity of the airport has been retrieved from Question 1.1, and the CCDF of the data have been plotted in above diagram. As seen, the popularity of the airports does not follow Zipf distribution, but follow lognormal distribution.

Question 3.2 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...

This question could be spitted into 2 sub-questions:

1. Find the flight from X to Y on given date in the morning, with minimum arrival delay;
*2. Find the flight from Y to Z on 2 days after given date in the afternoon, with minimum arrival d*elay.
To solve each sub-question, the MapReduce implementation needs to prepare a custom key class that includes the information of departure airport, arrival airport, flight data, morning/afternoon; Also, the implementation needs a custom value class that includes the information of carrier Id, flight number, departure time and arrival delay. The Reducer method is mainly to find the key with minimum arrival time.
To further process and query the data, Spark and Cassandra is used.
A table is created in Cassandra cqlsh:

```
apache-cassandra-3.11.2/bin/cassandra
apache-cassandra-3.11.2/bin/cqlsh $(hostname -i)

create table aviation.optimum_flights (
  airport_from text,
  airport_to text,
  given_date date,
  am_or_pm text,
  carrier text,
  flight_num text,
  departure_time text,
  arr_delay decimal,
  PRIMARY KEY(airport_from, airport_to, given_date, am_or_pm)
);
```

PySpark is then executed together with Spark-Cassandra_Connector, and write the data into Cassandra table.
To query the result in Cassandra cqlsh:

```
select * from aviation.optimum_flights where airport_from = ‘CMI' and airport_to = 'ORD' and given_date = '2008-04-03' and am_or_pm = 'AM';
select * from aviation.optimum_flights where airport_from = ‘ORD' and airport_to = 'LAX' and given_date = '2008-04-05' and am_or_pm = 'PM';
```

The given query result is:

	Carrier	Flight number	Departure time	Flight date	Arrival delay
CMI → ORD → LAX, 04/03/2008	MQ	4278	07:10	2008-03-04	-14.0
	AA	607	19:50	2008-03-06	-24.0
JAX → DFW → CRP, 09/09/2008	AA	845	07:25	2008-09-09	1.0
	MQ	3627	16:45	2008-09-11	-7.0
SLC → BFL → LAX, 01/04/2008	00	3755	11:00	2008-04-01	12.0
	00	5429	14:54	2008-04-03	6.0
LAX → SFO → PHX, 12/07/2008	WN	3534	06:50	2008-07-12	-13.0
	WN	1619	16:05	2008-07-09	-17.0
DFW → ORD → DFW, 10/06/2008	UA	1104	07:00	2008-06-10	-21.0
	AA	2341	16:45	2008-06-12	-10.0
LAX → ORD → JFK, 01/01/2008	UA	944	07:05	2008-01-01	1.0
	B6	918	19:00	2008-01-03	-7.0