Data Cleaning And Exploration

Aviation Dataset Mount & Initialization

To be able to work with data, first we have to mount the transportation dataset to our EC2 instance. I used the `lsblk` command to see available block storages mounted to my EC2. Then created a separate folder and mounted the volume to the filesystem.

```
$ lsblk
$ sudo mkdir /data
$ mount /dev/xvdb /data
```

References

- AWS User Guide: Attaching EBS volumes
- AWS User Guide: Using EBS volumes

Data exploration using `bash`

At a certain level bash is perfectly fine to discover what's included in the transportation dataset. At first with directory navigation we can investigate each folder and see what's inside. Obviously we're just interested in the `aviation` subfolder.

```
$ ls
air_carrier_employees          air_carrier_statistics_summary  airline_origin_destination    aviation_safety_reporting
...
$ cd airline_ontime
$ ls
1988  1989  1990  1991  1992  1993  1994  1995  1996  1997  1998
...
$ cd 2008
$ ls
On_Time_On_Time_Performance_2008_10.zip  On_Time_On_Time_Performance_2008_2.zip  On_Time_On_Time_Performance_2008_6.zip
...
```

We can peek inside each zip file using `bash` as well, ff we pipe the output of each file to the `gunzip` command. Directing the output to an other file allows us to save samples and test our map-reduce jobs on small portion of data.

```
cat ./On_Time_On_Time_Performance_2008_10.zip | gunzip | head -255 > ~/airline_ontime_perf.csv
```

Moving relevant data to Hadoop HDFS

We'll just work with the `airline_ontime` data, which contains on-time performance for each flight. A special `bash` script is getting all the zip archives in all subfolders, searches inside each zip file for CSV extensions and unzips only those files from the zip archive. We'll pipe each CSV output to a `hdfs put` command.

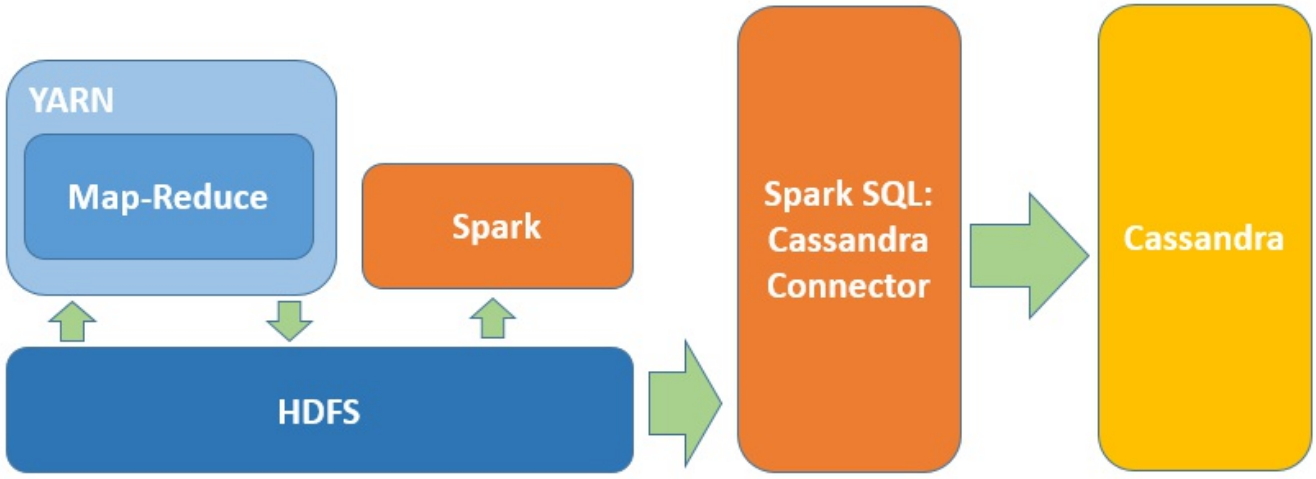
```
migration/move-ontime-perf-to-hadoop.sh /data/aviation /user/ec2-user/ontime_perf
```

References

- Migration scripts on GitHub

System Integration

All heavyweight data simplifying and transformation jobs are done by map-reduce over YARN. These jobs most of the time crawl all the CSV files under `airline_ontime` subfolder. The goal of these map-reduce jobs is to prepare dataset digestible for Spark. Spark provides the final results over this "clean" dataset.



Cassandra Migration

Migrating to Apache Cassandra is done by using Spark Cassandra Connector. This allows shifting loaded DataFrames from Spark to Apache Cassandra.

References

- Spark Cassandra Connector
- PySpark with Spark Cassandra Connector

Solution Approach

Question 1.1

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

A map-reduce job collects all the from-to airport field from `airline_ontime` data and counts each airport. This is very similar to the well-known Word Count example in Hadoop documentation.

<..., line> -> **map()** -> <airport_id, 1> -> **reduce()** -> <airport_id, occurrence>

Map-Reduce execution is done by the following command.

```
bin/hadoop jar ~/IdeaProjects/cloud-capstone/out/artifacts/cloud_capstone/cloud-capstone.jar com.cloudcomputing.PopularAirportsPlaintext
ontime_perf popular_airports
```

As result we get one file with airports in alphabetical order.

```
ABE     236094
ABI     39323
ABQ     1428081
...
```

This file is sorted and trimmed using `PySpark`

```
file = sc.textFile('hdfs://localhost:9000/user/ec2-user/popular_airports/part-r-00000')
rdd = file.cache()
rdd.map(lambda line: line.split()).filter(lambda tuple: len(tuple) == 2).filter(lambda tuple: len(tuple[0]) == 3).map(lambda tuple:
(int(tuple[1] tuple[0])).sortByKey(ascending=False).take(10)
```

References

Map-Reduce job

Question 1.2

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

The solution is similar to the previous. We get each carrier and it's arrival delay field. Then at the reduce phase we calculate the average arrival delay for each carrier.

<..., line> -> **map()** -> <carrier_id, arrival_delay> -> **reduce()** -> <carrier_id, average_arrival_delay>

Map-Reduce execution is done by the following command.

```
bin/hadoop jar ~/IdeaProjects/cloud-capstone/out/artifacts/cloud_capstone/cloud-capstone.jar com.cloudcomputing.AverageDelays ontime_perf
avg_delays
```

As result we get one file with carriers in alphabetical order.

```
9E      5.87
AA      7.11
AL      8.29
...
```

This file is then sorted and trimmed using `PySpark`

```
file = sc.textFile('hdfs://localhost:9000/user/ec2-user/avg_delays/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)
```

References

Map-Reduce job

Question 2.1

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

For this calculation, the map-reduce jobs will use a custom compound key, that's composed from airport ID and carrier ID. Map jobs will emit each **<airport,carrier>** key pair's on-time departure performance and reduce jobs will calculate average for each **<airport,carrier>** key pair.

<..., line> -> **map()** -> <(airport_id, carrier_id departure_delay> -> **reduce()** -> <(airport_id, carrier_id average_departure_delay>

Map-Reduce execution is done by the following command.

```
bin/hadoop jar ~/IdeaProjects/cloud-capstone/out/artifacts/cloud_capstone/cloud-capstone.jar com.cloudcomputing.OnTimeDepartureByCarriers
ontime_perf departure_by_carriers
```

As result we get one file with airports and carriers in alphabetical order. Each value represents the on-time departure average for that airport-carrier pair.

```
ABE 9E  6.75
ABE AA  4.76
ABE AL  3.63
ABE DH  5.34
ABE DL  23.28
...
```

This file is then sorted using `PySpark`

```
def top_ten_carriers(catalog, from_airport):
        rdd2 = catalog.filter(lambda tuple: tuple[0] == from_airport)
        rdd2 = rdd2.map(lambda tuple: (float(tuple[2]), tuple[1], tuple[0]))
```

```
        return rdd2.takeOrdered(10)

file = sc.textFile('hdfs://localhost:9000/user/ec2-user/departure_by_carriers/part-r-00000')
catalog = file.map(lambda line: line.split()).cache()
top_ten_carriers(catalog, 'CMI')
```

Saving to Cassandra is done by using the spark-cassandra-connector package. We have to mark this dependency, when starting up PySpark

```
./bin/pyspark --conf spark.cassandra.connection.host=<CASSANDRA_HOST> --packages datastax:spark-cassandra-connector:2.0.0-RC1-s_2.11
```

PySpark command, that loads results and then moves to Cassandra node.

```
file = sc.textFile('hdfs://localhost:9000/user/ec2-user/departure_by_carriers/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_departure", keyspace="aviation")\
    .save()
```

Cassandra table definition is the following

```
create keyspace aviation WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 1 };
create table aviation.airport_carrier_departure (
  airport text,
  carrier text,
  dep_delay decimal,
  PRIMARY KEY(airport, dep_delay, carrier)
);
```

Using the table to determine top performer can be done with this CQL command

```
select * from aviation.airport_carrier_departure where airport = 'MIA' order by dep_delay limit 3;

 airport | dep_delay | carrier
---------+-----------+---------
     MIA |      -3.0 |      9E
     MIA |       1.2 |      EV
     MIA |       1.3 |      RU

(3 rows)
```

References

Map-Reduce job

Question 2.2

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

This is very similar to solution to Question 2.1. Here the compound key consists origin and destination airport.

<..., line> -> **map()** -> <(airport_from, airport_to departure_delay> -> **reduce()** -> <(airport_from, airport_to average_departure_delay>

Map-Reduce execution is done by the following command.

```
bin/hadoop jar ~/IdeaProjects/cloud-capstone/out/artifacts/cloud_capstone/cloud-capstone.jar com.cloudcomputing.OnTimeDepartureByAirports
ontime_perf departure_by_airports
```

Sample from map-reduce result

```
  ABE ALB 10.00
  ABE ATL 10.03
  ABE AVP 3.44
  ABE AZO 241.00
  ABE BDL 0.00
```

```
...
```

This file is then sorted using `PySpark`

```
def top_ten_airports(catalog, from_airport):
        rdd2 = catalog.filter(lambda tuple: tuple[0] == from_airport)
        rdd2 = rdd2.map(lambda tuple: (float(tuple[2]), tuple[1], tuple[0]))
        return rdd2.takeOrdered(10)

file = sc.textFile('hdfs://localhost:9000/user/ec2-user/departure_by_airports/part-r-00000')
catalog = file.map(lambda line: line.split()).cache()
top_ten_airports(catalog, 'CMI')
```

PySpark command, that loads results and then moves to Cassandra node.

```
file = sc.textFile('hdfs://localhost:9000/user/ec2-user/departure_by_airports/part-r-00000')
rdd = file.map(lambda line: line.split())
rdd = rdd.filter(lambda tuple: len(tuple) == 3).filter(lambda tuple: len(tuple[0]) == 3).filter(lambda tuple: len(tuple[1]) == 3)
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], airport_to=row[1], dep_delay=row[2]))
df = spark.createDataFrame(rdd3)
df.write\
    .format("org.apache.spark.sql.cassandra")\
    .mode('append')\
    .options(table="airport_airport_departure", keyspace="aviation")\
    .save()
```

Cassandra table definition is the following

```
create table aviation.airport_airport_departure (
  airport text,
  airport_to text,
  dep_delay decimal,
  PRIMARY KEY(airport, dep_delay, airport_to)
);
```

CQL query example

```
select * from aviation.airport_airport_departure where airport = 'LAX' order by dep_delay limit 3;

 airport | dep_delay | airport_to
---------+-----------+------------
     LAX |     -16.0 |        SDF
     LAX |      -7.0 |        IDA
     LAX |      -6.0 |        DRO

(3 rows)
```

References

Map-Reduce job

Question 2.4

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

This is very similar to solution to Question 2.2. Here the compound value consists arrival delay instead of departure delay.

<..., line> -> **map()** -> <(airport_from, airport_to arrival_delay> -> **reduce()** -> <(airport_from, airport_to average_arrival_delay>

Map-Reduce execution is done by the following command.

```
bin/hadoop jar ~/IdeaProjects/cloud-capstone/out/artifacts/cloud_capstone/cloud-capstone.jar com.cloudcomputing.OnTimeArrivalByAirports
ontime_perf arrival_by_airports
```

Sample from map-reduce result

```
ABE ALB 23.00
```

```
  ABE ATL 7.65
  ABE AVP 2.35
  ABE BDL 1.00
  ABE BHM -3.00
  ...
```

This file is then sorted using `PySpark`

```
def mean_arrival_delay(catalog, from_airport, to_airport):
        rdd = catalog.filter(lambda tuple: tuple[0] == from_airport and tuple[1] == to_airport)
        return rdd.collect()

file = sc.textFile('hdfs://localhost:9000/user/ec2-user/arrival_by_airports/part-r-00000')
catalog = file.map(lambda line: line.split()).cache()
mean_arrival_delay(catalog, 'CMI', 'ORD')
```

PySpark command, that loads results and then moves to Cassandra node.

```
file = sc.textFile('hdfs://localhost:9000/user/ec2-user/arrival_by_airports/part-r-00000')
rdd = file.map(lambda line: line.split())
rdd = rdd.filter(lambda tuple: len(tuple) == 3).filter(lambda tuple: len(tuple[0]) == 3).filter(lambda tuple: len(tuple[1]) == 3)
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], airport_to=row[1], arr_delay=row[2]))
df = spark.createDataFrame(rdd3)
df.write\
    .format("org.apache.spark.sql.cassandra")\
    .mode('append')\
    .options(table="airport_airport_arrival", keyspace="aviation")\
    .save()
```

Cassandra table definition is the following

```
create table aviation.airport_airport_arrival (
  airport text,
  airport_to text,
  arr_delay decimal,
  PRIMARY KEY(airport, airport_to)
);
```

CQL query example

```
cqlsh> select * from aviation.airport_airport_arrival where airport = 'DFW' and airport_to = 'IAH';

 airport | airport_to | arr_delay
---------+------------+-----------
     DFW |        IAH |      7.62

(1 rows)
```

References

Map-Reduce job

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.

We use the same approach except, that key-value pairs will be both custom Writable extensions in this case. Keys will be constructed from (airport_from, airport_to, flight_date, am_or_pm). Values will be created from (carrier_id, flight_num, departure_time, arrival_delay). Reduce jobs will get all arrival delays from all the keys.

The main goal here is to be able to tell average delay for all the origin-destination pair at a given date. Distinguishing morning and afternoon flights. Because the problem can be split into two independent events (getting from X to Y and from Y to Z we can answer each questions by two queries.

<..., line> -> **map()** -> <(airport_from, airport_to, flight_date, am_or_pm (carrier_id, flight_num, departure_time, arrival_delay)> -> **reduce()** -> <(airport_from, airport_to, flight_date, am_or_pm (carrier_id, flight_num, departure_time, average_arrival_delay)>

Map-Reduce execution is done by the following command.

```
bin/hadoop jar ~/IdeaProjects/cloud-capstone/out/artifacts/cloud_capstone/cloud-capstone.jar com.cloudcomputing.BestFlightOnAGivenDate
ontime_perf/*2008*.csv best_flights_2008
```

Sample from map-reduce result

```
  ABE ATL 2008-01-01 PM   EV 4833 12:05 12.00
  ABE ATL 2008-01-02 AM   EV 4923 06:00 -19.00
  ABE ATL 2008-01-02 PM   EV 4206 17:17 -35.00
  ABE ATL 2008-01-03 AM   EV 4877 06:40 409.00
  ABE ATL 2008-01-03 PM   EV 4206 17:20 13.00
  ...
```

Searching for a given flight can be done by `PySpark` using two different queries. One from searching at travel date with AM and another one with two days after using PM as parameter.

```
def get_best_flight(catalog, from_airport, to_airport, date, am_or_pm):
    rdd = catalog.filter(lambda tuple: tuple[0] == from_airport and tuple[1] == to_airport)
    rdd = rdd.filter(lambda tuple: tuple[2] == date and tuple[3] == am_or_pm)
    values = rdd.first()
    print values
    return "%s %s at %s, delay: %s" % tuple(values[4:])

file = sc.textFile('hdfs://localhost:9000/user/ec2-user/best_flights_2008/part-r-00000')
catalog = file.map(lambda line: line.split()).cache()
get_best_flight(catalog, 'SLC', 'BFL', '2008-01-04', 'AM')
get_best_flight(catalog, 'BFL', 'JFK', '2008-01-06', 'PM')
```

PySpark command, that loads results and then moves to Cassandra node.

```
data_catalog = 'hdfs://localhost:9000/user/sniper/best_flights_2008/part-r-00000'
catalog = sc.textFile(data_catalog).cache()
rdd = catalog.map(lambda line: line.split())
rdd2 = rdd.map(lambda tuple: (tuple[0], tuple[1], tuple[2], tuple[3], tuple[4], tuple[5], tuple[6], float(tuple[7])))
from pyspark.sql import Row
rdd3 = rdd2.map(lambda row: Row(airport_from=row[0], airport_to=row[1], given_date=row[2], am_or_pm=row[3], carrier=row[4],
flight_num=row[5], departure_time=row[6], arr_delay=row[7])).cache()
df = spark.createDataFrame(rdd3)
df.write\
    .format("org.apache.spark.sql.cassandra")\
    .mode('append')\
    .options(table="best_flights_2008", keyspace="aviation")\
    .save()
```

Cassandra table definition is the following

```
create table aviation.best_flights_2008 (
  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)
);
```

CQL query example

```
cqlsh> select * from aviation.best_flights_2008 where airport_from = 'SLC' and airport_to = 'BFL' and given_date = '2008-01-06' and
am_or_pm = 'AM';

 airport_from | airport_to | given_date | am_or_pm | arr_delay | carrier | departure_time | flight_num
--------------+------------+------------+----------+-----------+---------+----------------+------------
          SLC |        BFL | 2008-01-06 |       AM |     100.0 |      OO |          11:40 |       3755

(1 rows)
```

References

Map-Reduce job

Results

Question 1.1

```
12446097, ORD
11537401, ATL
10795494, DFW
7721141, LAX
6582467, PHX
6270420, DEN
5635421, DTW
5478257, IAH
5197649, MSP
5168898, SFO
```

Question 1.2

```
-1.01, HA
1.16, AQ
1.45, PS
4.75, ML
5.35, PA
5.47, F9
5.56, NW
5.56, WN
5.74, OO
5.87, 9E
```

Question 2.1

CMI

```
[(0.61, u'OH', u'CMI'),
(2.03, u'US', u'CMI'),
(4.12, u'TW', u'CMI'),
(4.46, u'PI', u'CMI'),
(6.03, u'DH', u'CMI'),
(6.67, u'EV', u'CMI'),
(8.02, u'MQ', u'CMI')]
```

BWI

```
[(0.76, u'F9', u'BWI'),
(4.76, u'PA', u'BWI'),
(5.18, u'CO', u'BWI'),
(5.5, u'YV', u'BWI'),
(5.71, u'NW', u'BWI'),
(5.75, u'AL', u'BWI'),
(6.0, u'AA', u'BWI'),
(7.24, u'9E', u'BWI'),
(7.5, u'US', u'BWI'),
(7.68, u'DL', u'BWI')]
```

MIA

```
[(-3.0, u'9E', u'MIA'),
(1.2, u'EV', u'MIA'),
(1.3, u'RU', u'MIA'),
(1.78, u'TZ', u'MIA'),
(2.75, u'XE', u'MIA'),
```

```
 (4.2, u'PA', u'MIA'),
 (4.5, u'NW', u'MIA'),
 (6.06, u'US', u'MIA'),
 (6.87, u'UA', u'MIA'),
 (7.5, u'ML', u'MIA')]
```

LAX

```
[(1.95, u'RU', u'LAX'),
 (2.41, u'MQ', u'LAX'),
 (4.22, u'OO', u'LAX'),
 (4.73, u'FL', u'LAX'),
 (4.76, u'TZ', u'LAX'),
 (4.86, u'PS', u'LAX'),
 (5.12, u'NW', u'LAX'),
 (5.73, u'F9', u'LAX'),
 (5.81, u'HA', u'LAX'),
 (6.02, u'YV', u'LAX')]
```

IAH

```
[(3.56, u'NW', u'IAH'),
 (3.98, u'PA', u'IAH'),
 (3.99, u'PI', u'IAH'),
 (4.8, u'RU', u'IAH'),
 (5.06, u'US', u'IAH'),
 (5.1, u'AL', u'IAH'),
 (5.55, u'F9', u'IAH'),
 (5.71, u'AA', u'IAH'),
 (6.05, u'TW', u'IAH'),
 (6.23, u'WN', u'IAH')]
```

SFO

```
[(3.95, u'TZ', u'SFO'),
 (4.85, u'MQ', u'SFO'),
 (5.16, u'F9', u'SFO'),
 (5.29, u'PA', u'SFO'),
 (5.76, u'NW', u'SFO'),
 (6.3, u'PS', u'SFO'),
 (6.56, u'DL', u'SFO'),
 (7.08, u'CO', u'SFO'),
 (7.4, u'US', u'SFO'),
 (7.79, u'TW', u'SFO')]
```

Question 2.2

CMI

```
[(-7.0, u'ABI', u'CMI'),
 (1.1, u'PIT', u'CMI'),
 (1.89, u'CVG', u'CMI'),
 (3.12, u'DAY', u'CMI'),
 (3.98, u'STL', u'CMI'),
 (4.59, u'PIA', u'CMI'),
 (5.94, u'DFW', u'CMI'),
 (6.67, u'ATL', u'CMI'),
 (8.19, u'ORD', u'CMI')]
```

BWI

```
[(-7.0, u'SAV', u'BWI'),
 (1.16, u'MLB', u'BWI'),
 (1.47, u'DAB', u'BWI'),
```

```
 (1.59, u'SRQ', u'BWI'),
 (1.79, u'IAD', u'BWI'),
 (3.65, u'UCA', u'BWI'),
 (3.74, u'CHO', u'BWI'),
 (4.2, u'GSP', u'BWI'),
 (4.45, u'SJU', u'BWI'),
 (4.47, u'OAJ', u'BWI')]
```

MIA

```
[(0.0, u'SHV', u'MIA'),
 (1.0, u'BUF', u'MIA'),
 (1.71, u'SAN', u'MIA'),
 (2.54, u'SLC', u'MIA'),
 (2.91, u'HOU', u'MIA'),
 (3.65, u'ISP', u'MIA'),
 (3.75, u'MEM', u'MIA'),
 (3.98, u'PSE', u'MIA'),
 (4.26, u'TLH', u'MIA'),
 (4.61, u'MCI', u'MIA')]
```

LAX

```
[(-16.0, u'SDF', u'LAX'),
 (-7.0, u'IDA', u'LAX'),
 (-6.0, u'DRO', u'LAX'),
 (-3.0, u'RSW', u'LAX'),
 (-2.0, u'LAX', u'LAX'),
 (-0.73, u'BZN', u'LAX'),
 (0.0, u'MAF', u'LAX'),
 (0.0, u'PIH', u'LAX'),
 (1.27, u'IYK', u'LAX'),
 (1.38, u'MFE', u'LAX')]
```

IAH

```
[(-2.0, u'MSN', u'IAH'),
 (-0.62, u'AGS', u'IAH'),
 (-0.5, u'MLI', u'IAH'),
 (1.89, u'EFD', u'IAH'),
 (2.17, u'HOU', u'IAH'),
 (2.57, u'JAC', u'IAH'),
 (2.95, u'MTJ', u'IAH'),
 (3.22, u'RNO', u'IAH'),
 (3.6, u'BPT', u'IAH'),
 (3.61, u'VCT', u'IAH')]
```

SFO

```
[(-10.0, u'SDF', u'SFO'),
 (-4.0, u'MSO', u'SFO'),
 (-3.0, u'PIH', u'SFO'),
 (-1.76, u'LGA', u'SFO'),
 (-1.34, u'PIE', u'SFO'),
 (-0.81, u'OAK', u'SFO'),
 (0.0, u'FAR', u'SFO'),
 (2.43, u'BNA', u'SFO'),
 (3.3, u'MEM', u'SFO'),
 (4.0, u'SCK', u'SFO')]
```

Question 2.4

CMI - ORD

```
[[u'CMI', u'ORD', u'10.14']]
```

IND - CMH

```
[[u'IND', u'CMH', u'2.89']]
```

DFW - IAH

```
[[u'DFW', u'IAH', u'7.62']]
```

LAX - SFO

```
[[u'LAX', u'SFO', u'9.59']]
```

JFK - LAX

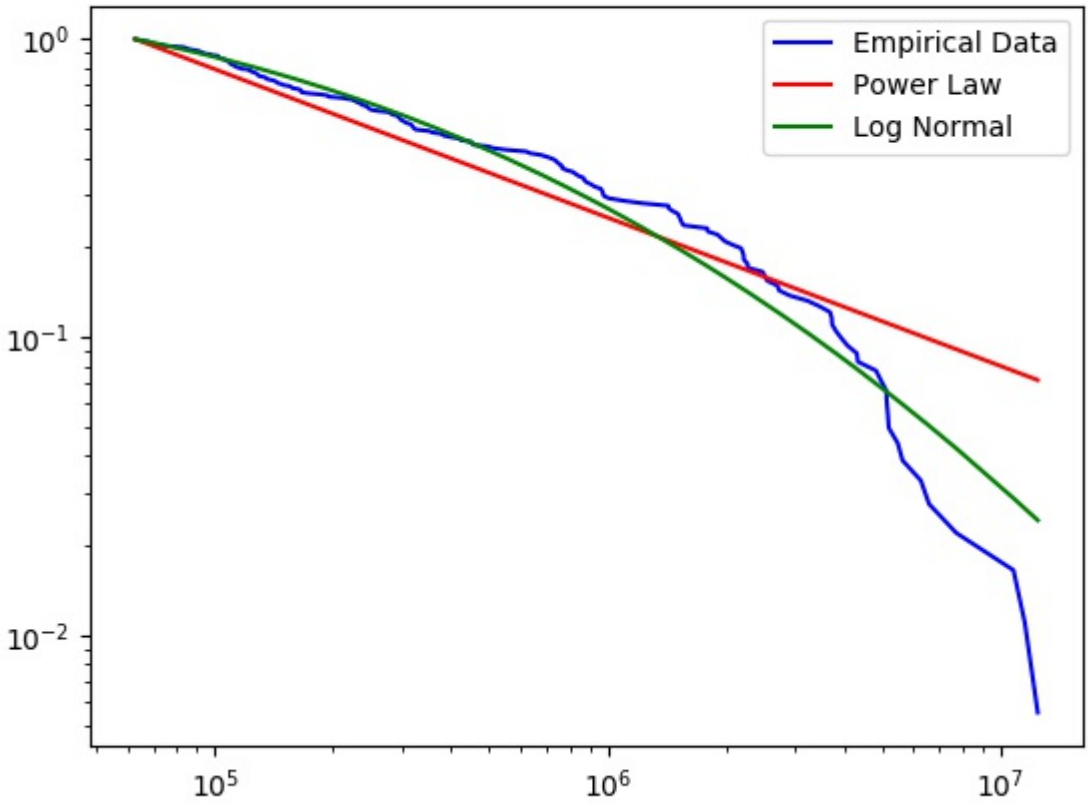
```
[[u'JFK', u'LAX', u'6.64']]
```

ATL - PHX

```
[[u'ATL', u'PHX', u'9.02']]
```

Question 3.1

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.

Loglikelyhood 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 is more likely follows a lognormal distribution.

References

- Python code in GitHub
- Python powerlaw package usage explanation

Question 3.2

CMI → ORD → LAX, 04/03/2008

```
MQ 4278 at 07:10, delay: -14.00
AA 607 at 19:50, delay: -24.00
```

JAX → DFW → CRP, 09/09/2008

```
AA 845 at 07:25, delay: 1.00
MQ 3627 at 16:45, delay: -7.00
```

SLC → BFL → LAX, 01/04/2008

```
OO 3755 at 11:00, delay: 12.00
OO 5429 at 14:55, delay: 6.00
```

LAX → SFO → PHX, 12/07/2008

```
WN 3534 at 06:50, delay: -13.00
US 412 at 19:25, delay: -19.00
```

DFW → ORD → DFW, 10/06/2008

```
UA 1104 at 07:00, delay: -21.00
AA 2341 at 16:45, delay: -10.00
```

LAX → ORD → JFK, 01/01/2008

```
UA 944 at 07:05, delay: 1.00
B6 918 at 19:00, delay: -7.00
```

Optimizations

Use Spark for fine-grained queries

Spark will deliver results faster, when executing same operations with different parameters. This is because Spark stores RDDs in local cache. Also it's using DAG and lazy evaluation of data-pipelines, which gives faster results, than map-reduce.

Map-reduce is executing all it's map and reduce operation in separate sub-processes, so in case of heavyweight transformations, when large set of files needs to be processed, it has a lower memory footprint, than Spark.

Use plain text CSV files instead of .zip format

Zipped files in HDFS are not splittable, so each map operation is tied to one compressed file. By extracting just the necessary CSV files, the map-reduce framework splits these files into blocks, which has better overall performance. Multiple map tasks can work on the same file in parallel, by reading different blocks on different nodes.

Using combiners in map-reduce jobs

Combiners get executed after each map task is finished. It's running on the same node as the map task itself, so Hadoop doesn't need to move data between nodes. By combining map task results into one record, network traffic is reduced throughout the cluster. (E.g. typically on word-count kind of problems we can combine each map task results into just one record, by counting occurrences locally).