1. Cleaning and storing of the dataset.

I first retrieved and extracted the dataset from the EBS volume snapshot. It is to be noted that for some years data for several months is missing and for year 2008 the zip files containing the data for months of November and December are incomplete/corrupted, and as a consequence the archive can't be retrieved. By analyzing the requirements for the for the groups of questions 1,2 and 3, I decided to extract from the raw data two datasets. First will be used for for the questions in groups 1 and 2 and also for question 3.1. The second will be used just for question 3.2 as this one deals with data pertaining to year 2008 only. The only raw data table that we need for all the tasks is the On-Time Performance data table. For the first dataset the fields that I found relevant for all questions in groups 1 and 2, and question 3.1 are listed below along with their description from On-Time Performance data table: **DayOfWeek** - Day of Week; **UniqueCarrier** - Unique Carrier Code. When the same code has been used by multiple carriers, a numeric suffix is used for earlier users, for example, PA, PA(1), PA(2). Use this field for analysis across a range of years; **Origin** - Origin Airport; **Dest** - Destination Airport; **DepDel15** - Departure Delay Indicator, 15 Minutes or More (1=Yes); **ArrDelay** - Difference in minutes between scheduled and actual arrival time. Early arrivals show negative numbers; **ArrDel15** - Arrival Delay Indicator, 15 Minutes or More (1=Yes).
The dataset is extracted from data available for all years in the raw dataset.
A flight is counted as "on **time**" if it operated less than 15 minutes later the scheduled **time** shown in the carriers' Computerized Reservations Systems (CRS). Arrival **performance** is based on arrival at the gate. **Departure performance** is based on **departure** from the gate.
According to the above definition for the task 1.1 we will need the **Origin** and **Dest** fields. For task 1.2 we need **UniqueCarrier** and **ArrDel15**. For task 1.3 we need **DayOfWeek** and **ArrDel15** fields.For the task 2.1 we need **Origin**, **UniqueCarrier** and **DepDel15**. For task 2.1 we need **Origin**, **Dest** and **DepDel15** fields. For task 2.3 we need **Origin**, **Dest**, **UniqueCarrier** and **ArrDel15** fields. For task 2.4 we need **Origin**, **Dest** and **ArrDelay** fields. For task 3.1 we will use the entire result set from task 1.1.
As mentioned, for task 3.2 we will use a different dataset that will extract relevant data just for the year 2008. As the set of fields required for this task is different I decided to create a separate dataset.The relevant fields for this dataset are listed below:
FlightDate - Flight Date (yyyymmdd); **UniqueCarrier** - Unique Carrier Code. When the same code has been used by multiple carriers, a numeric suffix is used for earlier users, for example, PA, PA(1), PA(2). Use this field for analysis across a range of years; **FlightNum** - Flight Number; **Origin** - Origin Airport; **Dest** - Destination Airport; **CRSDepTime** - Computerized Reservations Systems Departure Time (local time: hhmm); **ArrDelay** - Difference in minutes between scheduled and actual arrival time. Early arrivals show negative numbers.

To further simplify the processing of task 3.2 by considering the condition b) I split this dataset in two datasets. First will have flights departing before 12:00 PM which will always represent the first leg of the trip, and the second will have flights departing after 12:00 PM which will always represent the second leg of the trip. We will disregard flights departing at 12:00 PM to satisfy condition b). The two datasets are disjoint and will simplify the processing. All the processing for data extraction was done with shell and awk scripts

2. Setup and System Integration

In order to execute all tasks I used a Hadoop cluster with 4 AMIs. I have a name node and 3 data nodes. I uploaded the 3 data files that I have in the hdfs /input directory as following:
- **adi_all.dat** - containing data for all years for use for the tasks 1, 2 and 3.1
- **adi2008_AM.dat** - containing all flights before 12:00 PM for year 2008
- **adi2008_PM.dat** - containing all flights after 12:00 PM for year 2008

For each of the tasks I created a separate map-reduce job packaged in a jar file named Task_x_x.jar.
For the tasks 2 and 3.2 I have created another Task_cas.jar map-reduce task that will load all entries for the respective task into a Cassandra database.
I have a single node Cassandra DB server installed and running on the name node. I need to store the data just for the purpose of quering the results. For each task that requires the data to be queried I created a separate table in Cassandra. All are part of the "**dev**" keyspace. The structure of the tables is the same:
>**CREATE TABLE Task_x_x(t_key text PRIMARY KEY, t_data text);**
All output files for tasks 2 and 3 have a similar structure key-value that will allow to be processed by the same Cassandra map reduce job Task_cas.jar.
The hadoop command issued for all 1 and 2 tasks look like:
>**hadoop jar Task_x_x.jar Task_x_x /input/adi_all.dat /output/Task_x_x**
For task 3.2 the hadoop command I used is:
>**hadoop jar Task_3_2.jar Task_3_2 /input/adi2008_AM.dat /input/adi2008_PM.dat /output/Task_3_2**
For the tasks 2 and 3.2 I am running the following command to load the data in the corresponding Cassandra table:
>**hadoop jar Task_cas.jar Task_cas /output/Task_x_x/part-m-00000 Task_x_x**

3. Processing and results

Task 1.1 processing
This task is a regular count map reduce task with two jobs, one for aggregation based on airport and one for secondary reverse sorting based on the popularity count. The first mapper will read each input line, split it into tokens, and will emit two key-value pairs one for origin airport and one for the destination: **Origin** - 1, **Dest** - 1
Given the small number of airports I use a Hashmap<String,Integer> that will aggregate data and emit just one **Airport-Count** pair per mapper. I also use a combiner that is similar to the reducer. The reducer will aggregate the values for each key and will emit one key-value pair for each airport. For this job I set the combiner class to be the same as the reducer class. The second mapper will be an inverse mapper that will output the reversed input key-value pair as value-key. This is necessary for secondary sorting based on popularity. The second job will use a custom sort comparator class that will sort the popularity count values in reverse order. The reducer for this job will be an inverse reducer to return the sorted data to the format: airport - popularity. The first job completed in 41 sec and the secondary sort in 13 sec.

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

Task 1.2 processing
This task consists of two chained jobs. The first calculates the on time arrival performance for each airline, and the second performs secondary sorting based on time arrival performance in descending order. For on time arrival performance we will use **ArrDel15** field which has a value of 1 if there is a delay more than 15 minutes and 0 otherwise. The performance will be computed as a percentage: how many flights arrive with a delay of less than 15 minutes from the total number of flights. The first mapper will read each input line, split it into tokens and will emit one key-value pair **UniqueCarrier** - (**ArrDel15_Count, Total_Count**) per mapper. Given the small number of carriers I use a **HashMap<String, (ArrDel15_Count, Total_Count)>**. I am using also a combiner which will further aggregate local entries and emit a pair **UniqueCarrier** - (A**rrDel15_Count, Total_Count**) .The first reducer will aggregate the data pertaining to each carrier and

compute the percentage of flights arriving on time. The formula is: **(Total_count - Delayed_arrival_count)*100. /Total_count**.
The second job does a secondary sorting based on the arrival performance. The mapper is an inverse mapper emitting a pair **Arrival_Performance - UniqueCarrier**, with the **Arrival_Performance** as the key. The reducer is an inverse reducer that will output the data in the format **UniqueCarrier - Arrival_Performance**. I use a custom mapper that will sort **Arrival_Performance** in decreasing order. The first job completed in 40 sec and the secondary sort in 13 sec.

HA	**AQ**	**PS**	**ML (1)**	**WN**	**OO**	**EA**	**NW**	**9E**	**F9**
93.793	90.578	89.038	84.590	82.837	82.236	81.597	80.900	80.826	80.425

Task 1.3 processing
This task is similar to task 1.2 with the difference that key for the first job is the **DayOfWeek** rather than **UniqueCarrier**. The first job completed in 40 sec and the secondary sort in 13 sec.

Saturday	**Tuesday**	**Monday**	**Sunday**	**Wednesday**	**Thursday**	**Friday**
82.82716	81.16303	80.40859	80.21672	79.57186	77.02254	76.12842

Task 2.1 processing
This task consists of 2 chained map reduce jobs. The first job will compute the on time departure performance for each pair (**Origin, UniqueCarrier**). We will use the same definition as before for computing the on time departure performance. The job will use a composite key (**Origin, UniqueCarrier**). I am using some aggregation inside the mapper with a **HashMap<String,(DepDel15_Count,Total_Count)>**. The composite key is transformed to a string and later on extracted back from HashMap string key in order to increase performance. The map gets flushed every 1000 elements to avoid memory problems. The mapper will read each input line, split it into tokens, add values to the HashMap and will emit key-value pairs: **(Origin,UniqueCarrier) - (DepDel15_Count, Total_Count)** every time the HashMap has 1000 elements and during the cleanup. I am using a combiner that further aggregates the data. The reducer will compute the on time departure as above, as a percentage. The same approach is used for all other tasks from section 2.
The second job will compute the top 10 carriers based on the departure performance for each airport. The mapper will use a secondary composite key **(Origin, Departure_Performance)**. Will process the key-value pairs output by first job and for each **(Origin,UniqueCarrier) - Departure_Performance** pair will emit a **(Origin, Departure_Performance) - (UniqueCarrier,Departure_Performance)** pair. For this job we will use partitioning and grouping. They will be performed based on **Origin** part of the **(Origin, Departure_Performance)** key. Also the secondary keys are sorted in descending order.
The secondary reducer will receive all values according to the custom grouping comparator based on **Origin**, and as a consequence all **(UniqueCarrier,Departure_Performance)** values for a certain **Origin**. These entries are be sorted based on the **Departure_Performance** in descending order, due to the custom sorting order of the secondary key **(Origin, Departure_Performance)**. The reducer will aggregate the first 10 **(UniqueCarrier,Departure_Performance)** value pairs and emit a key - value pair in the format **Origin -(UniqueCarrier,Departure_Performance)(UniqueCarrier,Departure_Performance)…**
The first job completed in 43 sec and the secondary sort in 13 sec.

CMI	(US,96.14) (TW,92.58) (OH,90.66) (PI,90.51) (EV,86.24) (MQ,83.42) (DH,83.37)
BWI	(F9,92.79) (CO,89.18) (AA,88.96) (NW,87.77) (UA,85.44) (DL,85.41) (EA,85.20) (9E,85.16) (YV,84.99) (US,84.71)

MIA	(9E,100.00) (EV,91.19) (PA (1),90.15) (XE,89.99) (NW,89.21) (TZ,88.36) (UA,87.43) (US,86.86) (ML (1),85.53) (PI,84.35)
LAX	(PS,91.69) (HA,90.79) (MQ,89.66) (TZ,88.68) (NW,88.22) (OO,88.16) (ML (1),86.50) (CO,86.36) (AA,86.00) (FL,85.72)
IAH	(PI,91.84) (NW,91.09) (AA,89.61) (PA (1),89.55) (WN,88.38) (US,87.90) (TW,87.39) (DL,86.71) (OO,86.39) (XE,85.64)
SFO	(TZ,88.91) (PS,87.71) (PA (1),87.28) (DL,87.24) (HA,87.16) (NW,86.70) (DH,86.05) (AA,85.68) (CO,85.34) (MQ,84.14)

Task 2.2 processing

This task is similar to task 2.1 with the difference that the original composite key that will be used is **(Origin, Destination)** rather than **(Origin,UniqueCarrier)**. All other processing is the same. The first job completed in 46 sec and the secondary sort in 14 sec.

CMI	(PIT,96.35) (DAY,93.89) (STL,92.73) (PIA,92.08) (DFW,89.51) (CVG,88.94) (ATL,86.24) (ORD,82.90)
BWI	(MLB,96.60) (IAD,95.45) (DAB,94.58) (CHO,92.46) (UCA,92.12) (SRQ,92.11) (SJU,91.90) (OAJ,91.32) (BGM,90.83) (GSP,90.52)
MIA	(BUF,100.00) (SAN,96.17) (HOU,94.03) (ISP,93.57) (PSE,91.30) (SLC,90.87) (TLH,90.76) (MEM,90.64) (GNV,90.39) (SJC,89.17)
LAX	(MAF,100.00) (SDF,100.00) (BZN,100.00) (LAX,100.00) (PMD,100.00) (VIS,96.10) (MEM,93.27) (IYK,92.19) (HDN,91.46) (SNA,91.37)
IAH	(MLI,100.00) (MSN,100.00) (HOU,95.32) (AGS,95.03) (EFD,92.56) (JAC,91.07) (RNO,90.80) (MDW,90.70) (VCT,89.99) (CLL,89.68)
SFO	(SDF,100.00) (FAR,100.00) (MSO,100.00) (SCK,100.00) (LGA,96.97) (PIE,95.95) (BNA,94.58) (OAK,94.27) (MKE,90.93) (MEM,90.29)

Task 2.3 processing

This task is similar to the task 2.1 with the following differences. The primary composite key is **(Origin,Destination, UniqueCarrier)**. The second composite key is **(Origin,Destination, Departure_Performance).** The partitioning and grouping is done on the **(Origin,Destination)** part of the secondary key. The first job completed in 49 sec and the secondary sort in 14 sec.

CMI->ORD	(MQ,77.50)
IND->CMH	(AA,100.00) (CO,93.01) (HP,85.80) (US,85.34) (NW,82.77) (DL,79.33) (EA,77.10)
DFW->IAH	(PA (1),89.47) (CO,85.26) (OO,82.56) (UA,82.25) (XE,81.96) (EV,81.93) (DL,81.40) (AA,79.64) (MQ,74.14)
LAX->SFO	(TZ,90.48) (F9,87.26) (PS,87.18) (EV,79.74) (AA,78.29) (US,78.25) (MQ,77.82) (CO,77.04) (WN,76.07) (UA,75.69)
JFK->LAX	(UA,78.68) (AA,73.86) (HP,73.19) (DL,70.65) (TW,66.76) (PA (1),64.84)
ATL->PHX	(FL,76.63) (US,75.37) (HP,74.07) (EA,73.67) (DL,70.15)

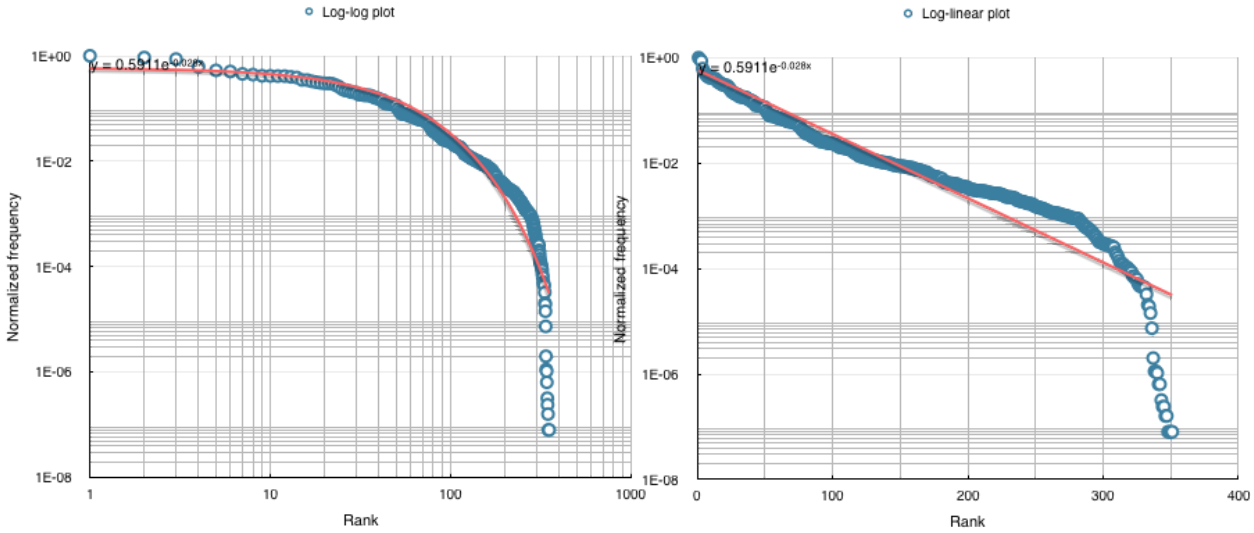
Task 2.4 processing

This task consists of just one job, however is using a composite key **(Origin, Destination).** The mapper will read each input line, split it into tokens and will emit one key-value pair: **(Origin,Destination) - ArrDelay**. The reducer aggregates the data and computes the average delay for each **(Origin,Destination)** key, and outputs a key - value pair in the format **(Origin,Destination) - AverageDelay.**The task completed in 46 sec.

CMI->ORD	IND->CMH	DFW->IAH	LAX->SFO	JFK->LAX	ATL->PHX
10.143662	2.8999038	7.6544423	9.589283	6.635119	9.021342

Task 3.1 processing

Below are two plots of popularity rank - flight frequency data. First is a log-log plot and the second a log-linear one. On the X axis we have popularity rank. On the Y axis we have normalized frequency an airport is being used. This is defined as flights to/from an airport divided by the number of flights to/from top ranking airport. For a Zipf distribution the log-log plot should be linear. As it is not, I conclude that the airport popularity does not follow Zipf law. Based on the second plot I conclude that for the first 300 airports the flight frequency distribution follows a log-linear model.



Task 3.2 processing

In order to satisfy the condition c) the flight we need to consider for each leg of the trip is one of those with minimum delay (if more than one found, the first encountered will be picked). This task will completed through a MapReduce job for each of the AM, PM datasets.

The task consists of 5 chained jobs. The first will process the AM data set followed by a sort job, then I process the PM data set followed by a sort job, and the last job will perform a join on the resulting datasets. The processing of AM and PM data sets is similar but not identical. The first job is using a composite key **(Origin, Destination, FlightDate)**. The mapper will read each input line, split it into tokens, and will emit one key-value pair:**(Origin, Destination, FlightDate) - (Origin, Destination, UniqueCarrier_FlightNumber, CRSDepTime, FlightDate, ArrDelay)**. In this case the key will be sorted in this order: **Destination, FlightDate, Origin** which will have as a consequence that data will properly sorted after the reduce task on **(Destination, FlightDate)** key. Unfortunately without an intermediate sorting step based on the new key, the last join operation did not work properly.The mapper of the PM dataset will use a similar key with the difference that it would be sorted in this order: **Origin, FlightDate, Destination**. The reducer will process the data, and output just the value with the minimum **ArrDelay** for each **(Origin, Destination, FlightDate)** key. The reducer will use another composite key **(Airport, Day_of_the_Year)**.

For AM dataset the key will be filled with **(Destination, Flight_Day_of_the_Year)** and for PM dataset with **(Origin, Flight_Day_of_the_Year-2)**. The **Destination** of the first flight must match the **Origin** of the second, and the flight date of second flight must be two days later than that of the first flight. The reducer will output a key-value pair as following: **(Airport, Day_of_the_Year) - (Origin, Destination, UniqueCarrier_FlightNumber, CRSDepTime, FlightDate, ArrDelay)**. These jobs for the AM and PM data sets, will employ the same number of reduce tasks in order to facilitate the join operation of the last job. The last job performs a inner join on the **(Airport, Day_of_the_Year)** key. The mapper will get a tupple for each key and output a key-value pair of the type Text-Text. The key will have the format **"Airport->Airport->Airport:FirstFlightDate"**. The value will have the format: "**(Origin, Destination, UniqueCarrier_FlightNumber, CRSDepTime_FlightDate, ArrDelay) of the first flight | (Origin, Destination, UniqueCarrier_FlightNumber, CRSDepTime_FlightDate, ArrDelay) of the second flight | Total_delay"**. First job took 33 sec., and the sorting took 34sec. The second job took 32 sec. and the sorting took 39 sec. The last join job took 5mins, 26sec. The final output produced 56,844,078 records.

	First flight	Second flight	Delay
CMI->ORD->LAX 2008-03-04	CMI->ORD,MQ 4278,07:10 2008-03-04,-14	ORD->LAX,AA 607,19:50 2008-03-06,-24	-38
JAX->DFW->CRP 2008-09-09	JAX->DFW,AA 845,07:25 2008-09-09,1	DFW->CRP,MQ 3627,16:45 2008-09-11,-7	-6
SLC->BFL->LAX 2008-04-01	SLC->BFL,OO 3755,11:00 2008-04-01,12	BFL->LAX,OO 5429,14:55 2008-04-03,6	18
LAX->SFO->PHX 2008-07-12	LAX->SFO,WN 3534,06:50 2008-07-12,-13	SFO->PHX,US 412,19:25 2008-07-14,-19	-32
DFW->ORD->DFW 2008-06-10	DFW->ORD,UA 1104,07:00 2008-06-10,-21	ORD->DFW,AA 2341,16:45 2008-06-12,-10	-31
LAX->ORD->JFK 2008-01-01	LAX->ORD,UA 944,07:05 2008-01-01,1	ORD->JFK,B6 918,19:00 2008-01-03,-7	-6

4. Optimizations

For the tasks that included more than 1 job, I employed the approach of chaining the jobs in just one application. For the output/input between the chained jobs I used **SequenceFileOutputFormat / SequenceFileInputFormat** with BLOCK compression type. For most of the tasks I used composite keys and secondary sorting. For the tasks 2.X and 3.2 I used partitioning and grouping based on the natural key. All tasks employed some sort of combiner which improved performance significantly. The approach of providing aggregation in the mapper through a HashMap that gets flushed once it reaches a predetermined number of elements greatly improved the performance.