Cloud Computing Capstone project. Part 1

Project scope

The assignment required to answer any 2 questions from Group 1, any 3 questions from Group 2 and both questions from Group 3. I decided to answer questions 1.1, 1.2, 2.1, 2.2, 2.4, 3.1 and 3.2. Consequently questions 1.3 and 2.3 were excluded and were not considered in any project activity.

Data requirements

Each question has its own requirements to the input data. The transportation data set has the table `airline_ontime` that contains records of individual flights.

Question 1.1: requires for each flight the airport of departure and the airport of arrival.
Question 1.2: requires for each flight the airline (the carrier) and the arrival delay.
Question 2.1: requires for each flight the airport of departure, the carrier and the departure delay.
Question 2.2: requires for each flight the airport of departure, the airport of arrival and the departure delay.
Question 2.4: requires for each flight the airport of departure, the airport of arrival and the arrival delay.
Question 3.1: requires the results to question 1. 1.
Question 3.2: requires for each flight the airline, the flight number, the airport of departure, the airport of arrival, the date of departure, the scheduled departure time and the arrival delay.

To answer question 1.1, I decided to include canceled flights and exclude the canceled flights from consideration when answering other questions. This decision requires for each flight the indicator of whether the flight was canceled or not.

Available data

To explore the available data, I created a new t2.micro instance in us-east-1c zone with Ubuntu Server 16.04 LTS (HVM) AMI, created an EBS volume from the snapshot snap-e1608d88 15GB in size in the same zone and attached the volume to the instance as `sdf`. I created a mounting point directory rawdata and mounted the volume. The needed data was found in the `aviation/airline_ontime` directory, grouped by years in subdirectories each containing zip-compressed files for each month of that year. Each compressed file contained a CSV file with the data with the file extension `.csv` and an HTML file with the file extension `.html` describing the data fields. The format of the data CSV files included the header line with the names of the fields, the fields were separated by commas, text fields were inside quotes, some text fields contained commas inside the fields values. Typical length of a row is 321 characters. The dataset contained 116,753,952 records. The data fields required to answer the questions of the project are listed in table 1.1 (field indices are 1-based):

Data requirement	**Data filed name**	**Field index in the source files**	**Field index in the input dataset files**
Airport of departure	`Origin`	12	1
Airport of arrival	`Dest`	18	2
Airline (Carrier)	`UniqueCarrier`	7	3
Flight number	`FlightNum`	11	4
Flight date	`FlightDate`	6	5
Scheduled Departure time	`CRSDepTime`	24	6
Departure delay	`DepDelay`	26	7
Arrival delay	`ArrDelay`	37	8
Flag if the flight was cancelled	`Cancelled`	42	9

Input data format

Because the available data files contained lots of fields not needed for answering the chosen questions I decided to extract only the required fields to reduce the size of the input data set and the required input storage IO of MapReduce tasks. Different questions required the same data so it made sense to produce a single input dataset that would contain the full set of the required data fields.

I decided to remove header lines so it will not be necessary to have special handling of them in each MapReduce map task. I chose the Tab Separated Values format (`.tsv`) for the files of the input dataset because the required fields cannot contain tab characters and the format is used by Hadoop MapReduce. I decided to remove quotes from the text fields because it would reduce the size of each record by 10 bytes, which would give 1.17 GB reduction for the entire dataset. It would also eliminate the processing of the quotes in the map tasks. I decided to format the `Cancelled` field as `0` and `1` instead of `0.00` and `1.00` used in the source files, which reduces the size of the input dataset by 350 MB.

The required fields in the input dataset are in the order given in table 1.1.

I decided to group records by years but not by months. Splitting the input dataset into separate files for each month would produce files that are too small in size and would fix the minimum number of map tasks for each application. Joining all the records into a single file would prevent using the same data for question 3.2 which needs data only for the single year 2008.

I decided not to use the compression for the dataset because the size of the data set is relatively small and the dataset will be used 6 times by different MapReduce applications. Compression would also limit the ability of Hadoop to split the input files into map tasks effectively. For question 3.2 it would force a single map task.

Process of extracting and cleaning the data

The size of the source data was small enough so it did not force using the distributed processing. The extraction process reduced the size of the data significantly so it made sense to do extraction before putting the data into the distributed files system and reduce the storage and network traffic.

Different tools were considered to perform the extraction and cleanup. The use of the linux `awk` and `sed` command line utilities was limited because some of the fields in the source CSV files contained commas inside them. The command line utilities of `csvkit` package could parse and extract the fields but stripping of quotes, removing header lines, converting the Cancelled field format would require the sequential use of other tools such as `sed` and `awk` and would cause multiple buffering of the intermediate results on disk. For better performance and finer control, I decided to write a specialized conversion utility using “C”.

The utility was optimized for performance. It takes source lines from the standard input and outputs converted lines into the standard output. It gets the input line in the buffer and parses it “in place” into fields up to column 42 ignoring the rest. It strips the quotes from the text fields then output only the required fields in the tab separated values format. The utility `CCDataCleanup` was compiled with the C language compiler from the `gcc` package. Using `stdin` and `stdout` allowed for integrating the utility with `unzip` to process the compressed source files. It was used in a shell script `datacleanup.sh`:

```
#!/bin/sh
RAWDIR="$1"
OUTDIR="$2"
for dir in ${RAWDIR}/aviation/airline_ontime/*/;
 do unzip -p ${dir}'*.zip' '*.csv' | /home/ubuntu/CCDataCleanup
>${OUTDIR}/data$(basename ${dir}).tsv;
done
```

The actual conversion was performed on the NameNode instance of the Hadoop cluster. The same volume was attached and mounted on the NameNode instance then the conversion was done using the local storage by executing the commands:

```
mkdir dataset
./datacleanup.sh rawdata dataset
```

The converted data was put into HDFS by using the command:

```
hdfs dfs -put /home/ubuntu/dataset /.
```

Hadoop cluster configuration

I decided to run Hadoop in a cluster consisting of 5 t2.xlarge instances with 30 GB EBS storage each running Ubuntu Server 16.04 LTS (HVM) AMI.

The cluster had one NameNode, one Secondary NameNode and 3 DataNodes. On each node Oracle Java 8 and Hadoop 2.7.4 were installed and configured. Replication level for HDFS was set to 3. Passwordless ssh access from the NameNode to the other nodes was setup. Internal IP-addresses from default VPC of the instances were used in the cluster configuration to allow the instances to be stopped and started. Shell scripts start-hadoop.sh and stop-hadoop.sh were added on the NameNode instance to simplify the start and stop of the Hadoop cluster.

Cassandra cluster configuration

I decided to use the same 5 instances to run the Cassandra cluster because of the educational nature of the project. I would recommend using separate instances for production environments.

Cassandra 3.11.1 was installed on all nodes from downloaded tarball file. The NameNode and the Secondary NameNode were configured as seeds. The shell scripts and aliases were added to facilitate the start and shutdown of the Cassandra cluster and to access `cqlsh` utility.

I chose the `cassandra-loader` utility to upload the results of MarReduce jobs to Cassandra. The utility is a self-contained executable which I downloaded from https://github.com/brianmhess/cassandra-loader/releases/download/v0.0.27/cassandra-loader.

Tools used to build MapReduce applications

I decided to use Java to implement Hadoop MapReduce applications. I started from the sample WordCount application as the base for my code. I used text editors to edit and the hadoop command to compile the applications. I used jar to create applications' jar files.

Question 1.1

The application to answer question 1.1 is very similar to the sample WordCount application. The mapper outputs the <*airport*><*number of flights*> pairs and the reducer calculates the sum of the number of flights. The combiner is the same as the reducer. The optimization is achieved by using "in-mapper aggregation" pattern. From the lookup table for the airports it is known that there could be maximum 6452 different airports. In-mapper aggregation uses an in-memory HashTable to aggregate the number of flights for each airport. Then at the end the mapper emits the pair for each entry in the HashTable. The number of pairs emitted by the mapper is reduced significantly. Without in-mapper aggregation each mapper would emit (number of entries in the split * 2) pairs, or total 233,507,904 pairs from all Map tasks. With the in-mapper aggregation each mapper would emit no more than 6452 pairs, or total (6452 * number of Map tasks) pairs for the job, which with 40 mappers gives 129,040 pairs maximum. In reality, the input dataset contains only 351 different airports which make the optimization even more efficient with the maximum map output of 14040 pairs for a job with 40 map tasks (compare with 233,507,904 pairs without optimization). The job executed on the cluster in 40 seconds.

The top 10 was retrieved from the result by using the command:

```
hdfs dfs -cat /results/air11/part-r-00000 | sort -n -k2 -t$'\t' -r | head -n10
```

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

Question 1.2

The mapper outputs a single pair for each carrier: <*UniqueCarrier*><*avgArravialDelay;count*>, where *avgArravialDelay* is the average arrival delay aggregated over the *count* entries. The combiner receives the set of the pairs from the mapper for the same key and aggregate them into a new pair of the same format. The reducer aggregates all partial sums (*avgArravialDelay* * *count*) to calculate the final average arrival delay. The optimization is achieved by using "in-mapper aggregation" pattern. From the lookup table for the `UniqueCarrier` field it is known that there could be maximum

1634 different airlines. The mapper aggregation uses an in-memory HashTable to aggregate the partial sums and counts for each airline. Then at the end the mapper emits a pair for each entry in the HashTable. The number of pairs emitted by the mapper is reduced significantly. There could be maximum of 75164 pairs for a job with 46 map tasks with optimization vs maximum of 116753952 pairs for a job without optimization. In reality, the input dataset contains only 29 different airlines. The job executed on the cluster in 36 seconds with 645 map output records.

The top 10 best performing airlines were retrieved from the result by using the command:

```
hdfs dfs -cat /results/air12/part-r-00000 | sort -n -k2  -t$'\t' | head -n10
```

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

Question 2.1

The mapper produces a single pair for each origin and airline combination: <*Origin*><*UniqueCarrier*;*avgDepDelay;count*>. The combiner aggregates the partial sums for each airline then emits the aggregated values in the same format as the mapper. The reducer aggregates the partial sums for each airline, then sort the aggregated entries and output the resulting pairs as <*Origin*><*rank*<tab>*UniqueCarrier*<tab>*AvgDepDelay*> where rank is the index in the sorted array and <tab> is the tab character. The result is in the TSV format. The in-mapper aggregation is used for optimization similar as for questions 1.1 and 1.2. The job executed on the cluster in 38 seconds with 52669 map output records. The sample queries:

CMI	**BWI**	**MIA**	**LAX**	**IAH**	**SFO**
OH 0.61	F9 0.76	9E -3.0	MQ 2.41	NW 3.56	TZ 3.95
US 2.03	PA(1) 4.76	EV 1.20	OO 4.22	PA(1) 3.98	MQ 4.85
TW 4.12	CO 5.18	TZ 1.78	FL 4.73	PI 3.99	F9 5.16
PI 4.46	YV 5.50	XE 1.87	TZ 4.76	US 5.06	PA(1) 5.29
DH 6.03	NW 5.71	PA(1) 4.20	PS 4.86	F9 5.55	NW 5.76
EV 6.67	AA 6.0	NW 4.50	NW 5.12	AA 5.70	PS 6.30
MQ 8.02	9E 7.24	US 6.09	F9 5.73	TW 6.05	DL 6.56
	US 7.49	UA 6.87	HA 5.81	WN 6.23	CO 7.08
	DL 7.68	MA(1) 7.50	YV 6.02	OO 6.59	US 7.53
	UA 7.74	FL 8.57	US 6.75	MQ 6.71	TW 7.79

Question 2.2

Similar to question 2.1 but the *Dest* field is used instead of the *UniqueCarriage* field. The job executed on the cluster in 39 seconds with 172474 map output records. The sample queries:

CMI	**BWI**	**MIA**	**LAX**	**IAH**	**SFO**
ABI -7.00	SAV -7.00	SHV 0.00	SDF -16.00	MSN -2.00	SDF -10.00
PIT 1.10	MLB 1.16	BUF 1.00	IDA -7.00	AGS -0.62	MSO -4.00
CVG 1.89	DAB 1.47	SAN 1.71	DRO -6.00	MLI -0.50	PIH -3.00
DAY 3.12	SRQ 1.59	SLC 2.50	RSW -3.00	EFD 1.89	LGA -1.76
STL 3.98	IAD 1.79	HOU 2.91	LAX -2.00	HOU 2.17	PIE -1.34
PIA 4.59	UCA 3.65	ISP 3.65	BZN -0.73	JAC 2.57	OAK -0.81
DFW 5.94	CHO 3.74	MEM 3.75	MAF 0.0	MTJ 2.95	FAR 0.00
ATL 6.67	GSP 4.20	PSE 3.98	PIH 0.0	RNO 3.22	BNA 2.43
ORD 8.19	SJU 4.44	TLH 4.26	IYK 1.27	BPT 3.60	MEM 3.30
	OAJ 4.47	MCI 4.61	MFE 1.38	VCT 3.61	SCK 5.00

Question 2.4

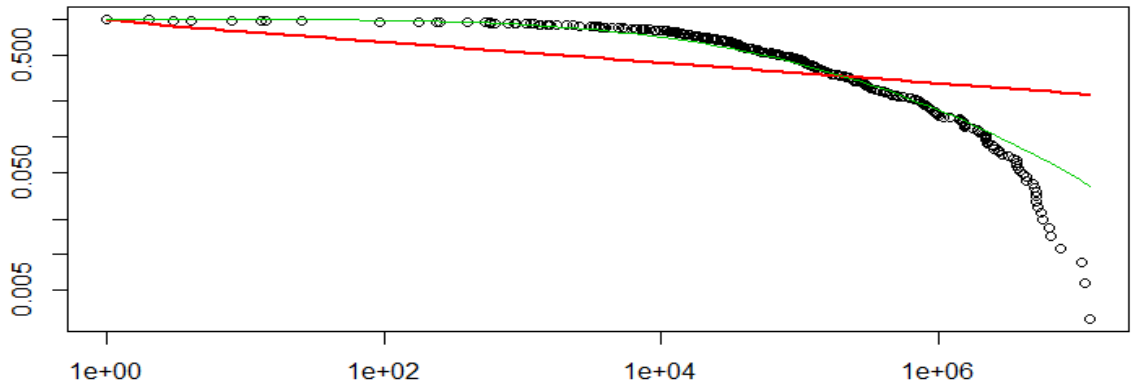
The mapper produces pairs <*Origin,Destination*><*avgArrDelay;count*>. The combiner aggregates partial sums and emits the new pair in the same format as the mapper. The reducer aggregates partial sums, calculates the final average and output <*Origin*><*Dest*<tab>*avgArrDelay*>. The in-mapper aggregation is used for optimization. The job executed on the cluster in 39 seconds with 170856 map output records. The sample queries:

CMI -> ORD	**IND -> CMH**	**DFW -> IAH**	**LAX -> SFO**	**JFK -> LAX**	**ATL -> PHX**

10.14	2.90	7.65	9.59	6.64	9.02

Question 3.1

I downloaded the results for question 1.1 from the cluster to my local Windows 7 machine, loaded them in "R" and extracted the second column as a vector for analysis. Below is the screenshot of CCDF with fitted power-law distribution in red and fitted log-normal distribution in green. It is obvious that the distribution of airports popularity does NOT follow Zipf (power-law) distribution.



Question 3.2

The mapper ignores cancelled flights.
For the flights departing before noon the mapper issues the pair
<*Dest,FlightDate*><*Origin;Dest;UniqueCarrier;FlightNum;FlightDate;CRSDepTime;ArrDelay*>.
For the flights departing afternoon the mapper issues the pair
<*Orig,FlightDate*-2 days><*Origin;Dest;UniqueCarrier;FlightNum;FlightDate;CRSDepTime;ArrDelay*>
There is no combiner because there is no aggregation in this case. The reducer receives all pairs with the same <stopover airport, original departure date> key. It builds 2 maps: one for the first leg flights (departing before noon) and the other for the 2nd leg flights, (departing afternoon). The 1st leg map uses *Origin* as its key and stores only the flight with the minimum *ArrDelay*. The 2nd leg map uses the *Dest* as its key and also stores only the flight with the minimum *ArrDelay*. The permutation of contents of the two maps produces possible routes satisfying the criteria that start on the given date and connect at the given stopover airport. The reducer outputs a TSV record for each possible combination. In this application there is no aggregation but a joining reducer is used. The reduce tasks are independent and the application should be optimized to have more reduce tasks to utilize the parallelism. I set the number of reduce tasks to 12 considering the configuration of the cluster (one task for each processing core in the cluster). The job executed in 1 minute 9 seconds on the cluster and produced 56,919,450 output records. The sample queries:

Route	**Dep. Date**	**Leg 1**	**Leg 1 delay**	**Leg 2**	**Leg 2 delay**	**Total delay**
CMI->ORD->LAX	2008-03-04	CMI->ORD MQ 4278 2008-03-04 07:10	-14	ORD->LAX AA 607 2008-03-06 19:50	-24	-38
JAX->DFW->CRP	2008-09-09	JAX->DFW AA 845 2008-09-09 07:25	1	DFW->CRP MQ 3627 2008-09-11 16:45	-7	-6
SLC->BFL->LAX	2008-04-01	SLC->BFL OO 3755 2008-04-01 11:00	12	BFL->LAX OO 5429 2008-04-03 14:55	6	18
LAX->SFO->PHX	2008-07-12	LAX->SFO WN 3534 2008-07-12 06:50	-13	SFO->PHX US 412 2008-07-14 19:25	-19	-32
DFW->ORD->DFW	2008-06-10	DFW->ORD UA 1104 2008-06-10 07:00	-21	ORD->DFW AA 2341 2008-06-12 16:45	-10	-31
LAX->ORD->JFK	2008-01-01	LAX->ORD UA 944 2008-01-01 07:05	1	ORD->JFK B6 918 2008-01-03 19:00	-7	-6