Problem Set 6

Thibault Doutre, Student ID 26980469

STAT243 : Statistical Computing
University of California, Berkeley

I worked on my own.

1 SQL

1.1 Airline Database

I create a script which I can execute via EC2. The script builds a database "airline.db" and store every file into a single table, using RSQLite. In order to do it, I first download the files in a directory named "data" and create the database. In this database, I store a table called "airline" in wich I will append the data for every year.

```
#Download
url = 'http://www.stat.berkeley.edu/share/paciorek/1987-2008.csvs.tgz'
download.file(url, ".file")

# Untar
untar(".file", compressed = 'bzip2', exdir = "./data/")

# Create Database
library(RSQLite)
drv <- dbDriver("SQLite")
db <- dbConnect(drv, dbname = "airline.db")

# Create airline Table
dbSendQuery(conn = db,
            [1333 chars quoted with '"']
)
```

Then, for every year:

- Open a connection to the file with bzcat
- Get the data and store it into a variable "line"
- Create a temporary table from line
- Append this table to "airline" using INSERT
- Remove the temporary table
- Close the connection

```
# Append years into airline table
for (i in 1987:2008){
  con=pipe(paste("bzcat ",i,".csv.bz2",sep=""), open = 'r')
  lines = read.csv(con, header = TRUE)
  dbWriteTable(db, paste("y",i,sep=""),lines)
 dbSendQuery(db,paste("INSERT INTO airline SELECT * FROM y",i,sep=""))
  dbRemoveTable(db,paste("y",i,sep=""))
  close(con)
}
```

Then, I print the total numer of lines in the database:

```
# Size in Gb
file.size("./airline.db")/2^30
# 9.273604

# Number of lines
dbGetQuery(db, "SELECT COUNT(*) FROM airline")
# 123534969
```

We can see that the database is 9 Gb big, it is less than the original CSV of 12 Gb but significantly bigger than the bzipped file of 1.7 Gb. We can see that the airline table in the database has one hundred million rows a mere.

1.2 Aggregating

In this section, I aggregate information into categories specified by the problem set. In order to do it, I write a Rscript which I can run using EC2. In the script, I first load the database.

```
# Load Database
print('Loading db...')
library(RSQLite)
fileName="airline.db"
drv = dbDriver("SQLite")
db = dbConnect(drv, dbname = fileName)
```

Then, I create a new table, which is in fact a subset of the airline dataset, when eliminating missing values for DepDelay and outliers. I chose to eliminate the same points as in unit7 i.e. departure delays which are less than 720 and more than -30.

```
# Create a new table from subsetting airline
print('Creating a subset of airline...')
dbSendQuery(db, "CREATE TABLE airline1 AS
            SELECT *
            FROM   airline
            WHERE  DepDelay > -30
            AND DepDelay < 720
            AND DepDelay != 'NA'")
```

Now, I send a querry, according to the threshold we want for DepDelay (30, 60 and 180). I use GROUP BY method over the selected columns. I also mapped the number of month with the name of the month and the number of day in the week with the corresponding names of them, using substr. Moreover, I display the hour of the CRSTime Departure by rounding the CRSDepTime value. Finally, I use count and coalesce functions in order to extract the percentage of Delay above the specified threshold. I also print the count of DepDelay per category, because I think that it is relevant to our statistics and it will be usefull for the next step.
I could have also done three different querries for the three thresholds but it would have been much slower.

```
# Create required table with percentage of DepDelays
print('Aggregating table...')
system.time(
dbSendQuery(db,"CREATE TABLE Delay30 AS
  SELECT
           Origin,
           Dest,
           substr('Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec ',
(Month * 4) - 3, 3) AS Month,
         substr('Mon Tue Wed Thu Fri Sat Sun ', (DayOfWeek * 4) - 3, 3)
AS Day,
           ROUND(CRSDepTime/100) AS Hour,
           coalesce(round(count(case when DepDelay > 180 then 1 end)/
(count(DepDelay)+.0)*100), 0) as PercentageDelay,
           coalesce(round(count(case when DepDelay > 60 then 1 end)/
(count(DepDelay)+.0)*100), 0) as PercentageDelay,
           coalesce(round(count(case when DepDelay > 30 then 1 end)/
(count(DepDelay)+.0)*100), 0) as PercentageDelay,
           count(DepDelay) AS count
           FROM airline1
           GROUP BY Origin, Dest, Month, Day, Hour")
)
```

The output looks like this:

```
Origin Dest Month Day Hour PercentageDelay180 PercentageDelay60 PercentageDelay30
1    ABE  ATL   Jan Fri    7                 0                0             25
```

```
2   ABE  ATL   Jan Fri   13              0              0              0
3   ABE  ATL   Jan Fri   18              0              0              0
4   ABE  ATL   Jan Mon    7              0              0              0
5   ABE  ATL   Jan Mon   13              0             20             20
6   ABE  ATL   Jan Mon   18              0              0              0
```

I have the following complexity results for aggregating values:

```
   user  system elapsed
560.296  42.220 683.313
```

1.3 Indexing

Let's compare the speed of aggregating when indexing the variables. I create a new script, which creates 3 new tables in the exact same way than previously but with indexing the subsetted airline1 table. I index the table this way:

```
# Create index
dbSendQuery(db,"CREATE INDEX index_name ON
          airline1 (Origin,Dest,Month,DayOfWeek,CRSDepTime,DepDelay)")
```

I got the following complexity results:

```
   user  system elapsed
410.720  36.892 505.591
```

The results are significantly better when indexing the tables.

1.4 Top results

In order to have the top 10 groupings keys in terms of proportion of late flights for groupings with at least 150 flights, I send this query, using the previous tables with the count field:

```
dbGetQuery(db,"SELECT * FROM Delay180I
           WHERE (count >150)
           ORDER BY PercentageDelay DESC
           LIMIT 10 ")
# Result
   Origin Dest Month Day Hour PercentageDelay count
1     EWR  ORD   Jun Wed   17              10   167
2     EWR  LAX   Jun Wed   17               8   170
3     EWR  ORD   Jun Wed   18               8   165
4     EWR  ORD   Jul Fri   17               8   165
```

```
5     EWR  ORD   Jul Fri   18                8   161
6     EWR  ORD   Jul Wed   18                8   171
7     EWR  ORD   Jun Fri   17                7   175
8     EWR  ORD   Jun Wed   16                7   196
9     EWR  ORD   Jun Wed   19                7   151
10    EWR  ORD   Aug Fri   18                7   175
```

2 Parallel Processing

In order to parallel the three queries we can use a foreach loop. In this loop I compute each of the querries and aggregate them using the .combine option. I artificially add the first columns Origin, Dest, Month, Day, Hour in another parallelized task. Also, we have to be aware that the database is already indexed. In fact, the real time spent to compute all the tasks should be approximately three times less than computing the three querries in a row. However, and it is very important to notice, calculating the three ratios in the same querry should be as fast as the parallelized code because SQL may have a smart way to compute the operations since they are asked in the same querry.

```r
library(foreach)
library(doMC)

# Set up 4 cores
registerDoMC(cores = 4)

system.time({
  print('computing aggregation...')
  out=foreach(i = c(-1,180,60,30), .combine = c) %dopar% {
    if (i==-1){
      db0=dbConnect(drv, dbname = fileName)
      s=dbGetQuery(db0," SELECT
                     Origin,
                     Dest,
                 substr('Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec '
                     ,(Month * 4) - 3, 3) AS Month,
                 substr('Mon Tue Wed Thu Fri Sat Sun ', (DayOfWeek * 4) -
                     3, 3) AS Day,
                     ROUND(CRSDepTime/100) AS Hour
                     FROM airline1
                     GROUP BY Origin, Dest, Month, Day, Hour")
      return(s)
      dbDisconnect(db0)
    }
    else{
      db1=dbConnect(drv, dbname = fileName)
      s=dbGetQuery(db1,paste("
                              SELECT
                          coalesce(round(count(case when DepDelay >",i,
                          " then 1 end)/(count(DepDelay)+.0)*100), 0) as
                              PercentageDelay",i,"
                              FROM airline1
                              GROUP BY Origin, Dest, Month, DayOfWeek,
                              ROUND(CRSDepTime/100)",sep=""))
      return(s)
      dbDisconnect(db1)
    }
  }
  print('writing table...')
  dbWriteTable(db,"summary",as.data.frame(out))
})
```

We indeed see that the task is parallelized by noticing that the user time is higher than the elapsed time. However, as mentionned above, the elapsed time is similar tothe elapsed time spent with one query. Here are my results on a m3xlarge instance on the whole data:

```
    user   system  elapsed
1543.842  227.148  773.935
```

3 Spark

Using spark, I compute the following chunk of code after connecting myself to ec2. Basically the commands before entering spark are:

– Log into EC2

```
export SPARK_VERSION=1.5.1
export CLUSTER_SIZE=12  # number of slave nodes
export mycluster=sparkvm-thibault.doutre # need unique
name relative to other users

cd /usr/local/lib/spark/ec2/

export AWS_ACCESS_KEY_ID=`grep aws_access_key_id stat
243-fall-2015-credentials.boto | cut -d' ' -f3`
export AWS_SECRET_ACCESS_KEY=`grep aws_secret_access
_key stat243-fall-2015-credentials.boto | cut -d' ' -f3`

######### NB CLUSTERS ###########
```

```
export NUMBER_OF_WORKERS=12

######### 10 minutes
./spark-ec2 -k thibault.doutre@berkeley.edu:stat243-
fall-2015 -i ~/.ssh/stat243-fall-2015-ssh_key.pem
--region=us-west-2 -s ${NUMBER_OF_WORKERS} -v 1.5.1
launch sparkvm-thibault.doutre

######### login
./spark-ec2 -k thibault.doutre@berkeley.edu:stat243-fall
-2015 -i ~/.ssh/stat243-fall-2015-ssh_key.pem --region=
us-west-2 login sparkvm-thibault.doutre
```

- Get the files in the right folder

```
export PATH=$PATH:/root/ephemeral-hdfs/bin/

hadoop fs -mkdir /data
hadoop fs -mkdir /data/airline

df -h
mkdir /mnt/airline
cd /mnt/airline

wget http://www.stat.berkeley.edu/share/paciorek/1987
-2008.csvs.tgz
tar -xvzf 1987-2008.csvs.tgz

hadoop fs -copyFromLocal /mnt/airline/*bz2 /data/airline
```

- Log into spark

```
# python 2.7
yum install -y python27-pip python27-devel
# numpy
pip-2.7 install 'numpy==1.9.2'
# pyspark is in /root/spark/bin
export PATH=${PATH}:/root/spark/bin
# start Spark's Python interface as interactive session
pyspark
```

Then comes the spark code. These are the steps:

- Filter out the NA values and the outliers
- Repartition
- Mapping
- Reducing

The screen function is used to filter data and the stratify function to map the three values we want:

- cnt1 corresponds to the 30 threshold
- cnt2 corresponds to the 60 threshold
- cnt3 corresponds to the 180 threshold

```
from operator import add
import numpy as np

lines = sc.textFile('/data/airline')

def screen(vals):
    vals = vals.split(',')
    return(vals[0] != 'Year' and vals[15] != 'NA' and float(vals[15])
           < 720 and float(vals[15]) > -30 )


def stratify(line):
    vals = line.split(',')
    if vals[0] == 'Year':
        return('0', [0,0,0])
    else:
      keyVals = '-'.join([vals[1],vals[3],str(int(float(vals[5])/100))
                                 ,vals[16],vals[17]])
        v=int(vals[15])
        cnt1=0
        cnt2=0
        cnt3=0
        if v>30:
            cnt1=1
            if v>60:
                cnt2=1
                if v>180:
                    cnt3=1
        return(keyVals, [cnt1,cnt2,cnt3])



output = lines.filter(screen).repartition(48).map(stratify).
reduceByKey(add)
result = output.collect()
```

Then, in order to put the result, which is a list into a text file, I do another mapping and save the result with saveAsTextFile.

```
def stratify2(elt):
    keyVals = ','.join(map(str,elt))
    return(keyVals)

result.map(stratify2).saveAsTextFile('/data/summary')
```

4 Preprocessing using Bash

Since we only use a subset of the data, we can eliminate useless columns using bash. And we know that this is really fast. I will write a script which actually does this. First, I create a new folder where I can put the files.

```
mkdir dataSubset
```

Then, I run the following script which process data using bzcat and save it to a bz2 file, for every year. I store the new data in dataSubset.

```
# Create script
vim Script_subset.sh
```

```
#!/bin/bash

subset_data()
{
for i in {1987..2008}
do
        echo $i
        bzcat data/$i.csv.bz2 | cut -d',' -f2,4,6,16,17,18 |
bzip2 > dataSubset/$i$subset.csv.bz2
done
}
```

Now I can execute the script:

```
source Script_subset.sh ; subset_data
```

The process takes approximately 10 minutes (9min50s) to run on a m3.xlarge instance. Normally, to create the SQL database, it takes 50 minutes approximately. But with this preprocessing it takes 10 minutes, which is 20 minutes total approximately. Therefore, it is better to use preprocessing in bash.