STAT243: PS 6

Xinyue Zhou

November 2, 2015

1.

Log into the EC2 as the tutorial shown, and download data using "wget". Unzip the file and enter R interface.

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

In R, load all the packages that are needed.

```
require(DBL)
require(data.table)
library(RSQLite)
```

Then I begin to create the database.

```
filename<-"airline.db"
drv<-dbDriver("SQLite")
db<-dbConnect(drv,dbname=filename)
#unzip a file for a year to see the classes of column in it.
tmp <- read.csv(bzfile("1987.csv.bz2"))
sapply(tmp, class)
tableCL <- c(rep("integer",8), "factor","integer","logical","integer",
  "integer","logical","integer","integer","factor","factor","integer","logical",
         "logical","integer","logical","integer",rep("logical",5))
years <- seq(1987, 2008)

system.time(for (i in years){
        print(i) #print the index to track the process
        nm<-paste("bunzip2 -c ",toString(i),".csv.bz2",sep="")
        dt <- fread(nm,header=TRUE, colClasses=tableCL)
        dbWriteTable(conn=db,value=dt,name="airlines_data",
                row.names=FALSE,append=TRUE)
})
rm(dt)
```

Here I got the processing time of creating the database:

user	system	elapsed
863.508	38.340	903.281

Calculate the size of the database now. Run the following in the bash:

```
ls -s -h airline.db
```

8.8G airline.db

I got the size of data base is 8.8GB.

2.

2.1 Solve it using SQLite

First of all, login EC2 and create the database as I did in part1. Notice I used *initExtension*(*db*) in the code. This enable me to use many R packages in the SQL. And then I also simply evaluate the database by taking a look at the table lists and the column names in the table. Then I printed out the size of the database.

```
library(RSQLite)
install.packages("data.table")
library(data.table)
fileName <- "Airlines.db"
db <- dbConnect(SQLite(), dbname = fileName)
initExtension(db)

tableCL <- c(rep("integer",8), "factor","integer","logical","integer",
  "integer","logical","integer","integer","factor","factor","integer","logical",
         "logical","integer","logical","integer",rep("logical",5))

for (i in 1987:2008){
  unzip_name <- paste("bunzip2 -c ", i,".csv.bz2",sep="")
  tmp <- fread(unzip_name, header = TRUE, colClasses=tableCL)
  dbWriteTable(conn = db, name = "Airlines", value = tmp, append=TRUE, row.names = FALSE)
  print(i)
}
dbListTables(db)
#"Airlines"
dbListFields(db, "Airlines")
# [1] "Year"              "Month"             "DayofMonth"
# [4] "DayOfWeek"         "DepTime"           "CRSDepTime"
# [7] "ArrTime"           "CRSArrTime"        "UniqueCarrier"
# [10] "FlightNum"         "TailNum"           "ActualElapsedTime"
# [13] "CRSElapsedTime"    "AirTime"           "ArrDelay"
# [16] "DepDelay"          "Origin"            "Dest"
# [19] "Distance"          "TaxiIn"            "TaxiOut"
# [22] "Cancelled"         "CancellationCode"  "Diverted"
# [25] "CarrierDelay"      "WeatherDelay"      "NASDelay"
# [28] "SecurityDelay"     "LateAircraftDelay"
file.size("Airlines.db")
#9399877632
```

2.1.1 Non-indexing Method

First I finished task without using index. First, I updata the CSRDepTime in the table to be just hour left. To fulfill this, I divide CRSDepTime by 100 and find the nearest integer below that number.

Next, I filtered all the NA values out and create a temporary table called airline_ data1, which was to be used in the next step create the table we are interested in.

Finally, I counted the number of each case we were interested in and divided by total number.

```
db_date_new <- dbSendQuery(db, "UPDATE Airlines SET CRSDepTime = floor(CRSDepTime/100)")
db_filter <- dbSendQuery(db,
  "CREATE VIEW airline_data1 AS SELECT * FROM Airlines WHERE DepDelay != 'NA'")
system.time(db_table <-dbSendQuery(db,
"CREATE TABLE table1 AS SELECT UniqueCarrier, Dest, Origin, Month, DayOfWeek, CRSDepTime,
sum(case when DepDelay>30 then 1 else 0 end)*1.0/count(*) AS per30,
```

```
sum(case when DepDelay>60 then 1 else 0 end)*1.0/count(*) AS per60,
sum(case when DepDelay>180 then 1 else 0 end)*1.0/count(*) AS per80,
COUNT (*) AS total
FROM airline_data1 GROUP BY UniqueCarrier, Dest, Origin, Month, DayOfWeek, CRSDepTime"))
```

Here is the time I used for creating the interested table:

user	system	elapsed
506.852	38.468	580.333

2.1.2 Indexing Method

From the documents, I found that the indexing in SQL processes can help shrink the processing time, since it helped us to search the target groups. Althrough indexing was also time-costly, it helped a lot if we are forced to select table again and again. Index the table using following query:

```
system.time(db_index <- dbSendQuery(db,
  "CREATE INDEX indices ON Airlines (UniqueCarrier, Dest, Origin, Month, DayOfWeek, CRSDepTime)"))
#    user  system elapsed
# 506.852  38.468 580.333
```

Seeing from above at the time it used for creating index, which is not trivial. However, it did make a big difference when I further manipulate the database.

Then I filtered the data and created the new table as airline_ data2 just as I did in 2.1.1:

```
system.time(db_filter2 <- dbSendQuery(db,
  "CREATE VIEW airline_data2 AS SELECT * FROM Airlines WHERE DepDelay != 'NA'"))
system.time(db_table2 <-dbSendQuery(db,
"CREATE TABLE table2 AS SELECT UniqueCarrier, Dest, Origin, Month, DayOfWeek, CRSDepTime,
sum(case when DepDelay>30 then 1 else 0 end)*1.0/count(*) AS per30,
sum(case when DepDelay>60 then 1 else 0 end)*1.0/count(*) AS per60,
sum(case when DepDelay>180 then 1 else 0 end)*1.0/count(*) AS per180,
COUNT (*) AS total
FROM airline_data2 GROUP BY UniqueCarrier, Dest, Origin, Month, DayOfWeek, CRSDepTime"))
```

Time used in creating table step:

user	system	elapsed
245.876	53.984	403.838

2.2 Solve it using Spark

It costed a while to setup the Spark. For convenient, I wrote a bash script so that I can easily setup it by running this:

```
bash login_spark.sh
```

Then I run another bash code in remoted computer terminal:

```
bash startSpark.sh
```

Then I am supposed to entered Python-like Spark interface.

After installing numpy, setuping the pyspark, and copying all the data to the hadoop, I read all the data in lines using sc.textFile(). This function is really smart, which can read lines in the zip files in the given directory.

First of all, I create a helper function to delete the header and filter the data that contains "NA" in the DepDelay variable. Then I applied the funcition on lines and partitioned it to 96 parts and threw it to the momery of nodes. Notice, I created 12 slavers, and for each slave, it has two cores. So, 4 pieces of data were supposed to be thrown on each core.

```
from operator import add
import numpy as np
lines = sc.textFile('/data/airline')
#filter out the header and NA values
def screen(vals):
  vals = vals.split(',')
return (vals[15] != 'NA' and vals[15] != 'DepDelay')

lines = lines.filter(screen).repartition(96).cache()
```

Create a mapper function using the method instructed in the class. This time we I handle the CRSDepTime variable, to avoid redundant calculation, I directly abandoned the last two elements of the string in CRSDepTime. Then I created a indicator-like array to specify if the line belongs to a certain group. "count4" was used to count the total number of the delaying flights.

After I applied the mapper function on lines, I reduced the keys of unique combinations by adding up all the indicators. Then I evaluated after mapping and took first 5 output to print out.

```
#mapper: return to a pair fo values. (key value, return values)
def computeKeyValue(line):
vals = line.split(',')
 #key is UniqueCarrier, Dest, Origin, Month, DayOfWeek, CRSDepTime
vals[5] = vals[5][:-2]
keyVals = '-'.join([vals[x] for x in [8,16,17,1,3,5]])
count1 = 0
count2 = 0
count3 = 0
count4 = 1
if float(vals[15]) > 30:
count1 = 1
if float(vals[15]) > 60:
count2 = 1
if float(vals[15]) > 180:
count3 = 1
return(keyVals, np.asarray([count1, count2, count3, count4]))

mapIt = lines.map(computeKeyValue).reduceByKey(np.add)
mapIt.take(5)
# Those are unique combo of keys
# key is UniqueCarrier, Dest, Origin, Month, DayOfWeek, CRSDepTime
# [(u'EA-ATL-MLB-12-2-17', array([0, 0, 0, 9])),
# (u'DL-ATL-DAY-11-6-9', array([ 0,  0,  0, 21])),
# (u'WN-RDU-PHX-1-7-11', array([0, 0, 0, 3])),
# (u'CO-IAH-MIA-11-1-21', array([ 2,  1,  0, 23])),
# (u'AS-BUR-SEA-6-3-8', array([ 0,  0,  0, 12]))]
```

As required in the problem set, I joint unique combination with array using comma.

```
def comma_joint(line):
vals = ','.join([str(line[0]), str(line[1][0]), str(line[1][1]), str(line[1][2]), str(line[1][3])])
return(vals)

ret=mapIt.map(comma_joint)
ret.take(5)
# ['TW-STL-ICT-8-6-9,0,0,0,34',
# 'WN-DEN-MDW-3-2-14,1,1,0,9',
# 'WN-RDU-PHX-1-7-11,0,0,0,3',
```

```
# 'HA-HNL-LIH-8-3-12,2,0,0,23',
# 'UA-ORD-PWM-6-7-,0,0,0,9']
```

The following step was used to evaluate the time used in create table we are interested in. Basically, I just wrote lines out to a cloud directory called '/data/xinyue233'. Then I used stop time(stoptime) deduced by the time started running(currenttime) and got the duration.

```
import timeit
currenttime = timeit.default_timer()
ret.repartition(1).saveAsTextFile('/data/xinyue233')
stoptime = timeit.default_timer()
stoptime-currenttime
#107.21423101425171
```

Copy the result from remote to local directory '/mnt/airline'. Then I cound enter the result directory locally and take a glance at the results. First several lines of the results are listed below.

```
#copy the data in cloud to local
hadoop fs -copyToLocal /data/xinyue233 /mnt/airline
cd /mnt/airline
ls
# 1987-2008.csvs.tgz  1990.csv.bz2  1994.csv.bz2  1998.csv.bz2  2002.csv.bz2  2006.csv.bz2
# 1987.csv.bz2        1991.csv.bz2  1995.csv.bz2  1999.csv.bz2  2003.csv.bz2  2007.csv.bz2
# 1988.csv.bz2        1992.csv.bz2  1996.csv.bz2  2000.csv.bz2  2004.csv.bz2  2008.csv.bz2
# 1989.csv.bz2        1993.csv.bz2  1997.csv.bz2  2001.csv.bz2  2005.csv.bz2  xinyue233
cd xinyue233/
ls
# part-00000  _SUCCESS
cat part-00000 | head -n5
# WN-SLC-LAS-4-2-15,1,1,0,44
# WN-ALB-BWI-6-6-10,1,0,0,29
# WN-DET-MDW-1-5-8,1,0,0,18
# OH-ATL-MHT-9-5-11,1,1,0,1
# US-JAX-CLT-9-4-13,2,2,0,58
```

Comparing the 2.1.1 SQLite step with 2.2 Spark step, I found the efficiency of the finishing the task had be highly improved by parallelizing it on the spark.

3. Parallel SQL

Log in EC2 virtual cluster as I did in Problem1.

It is reasonable to using month as the key for parallelizing, as we can expect that for each month, the number of flight is almost the same. By doing this, each node will be assigned simiar load of work, which can optimize our efficiency. Moreover, I select one month from each season to each thread to make it more balanced.
Then I wrote a function for each thread to run. Basically, it was just the same thing in problem2 d), while I just partition month into four groups. After that, I used mcapply() to setup number of cores, and applied this function to 1-4, which were the subsets of monthes.

```
library(RSQLite)
library(parallel) # one of the core R packages
library(doParallel)
library(foreach)
library(iterators)

drv <- dbDriver("SQLite")
```

```
db <- dbConnect(drv, dbname = "Airlines.db")
qry <- "SELECT UniqueCarrier, Origin, Dest, Month, DayOfWeek,CRSDepTime,sum(case when DepDelay>30 then 1 e
count(DepDelay) as Time_Delay from year where DepDelay!='NA' and Month in

taskFun_Month <- function(i){
    tmp<-paste(qry,"(",toString(i),',',toString(i+4),',',toString(i+4*2),")",
      "GROUP BY UniqueCarrier, Dest, Origin, Month, DayOfWeek, CRSDepTime",sep='')
    db_tmp<-dbConnect(drv,dbname="Airlines.db")
    stat_tmp<-dbGetQuery(db_tmp,tmp)
    return(stat_tmp)
  }
system.time(
  res1 <- mclapply(1:4, taskFun_Month, mc.cores = 4)
)
#    user  system elapsed
# 369.752  40.272 204.517
```

As I expected, by parallelizing the program on several nodes, the speed was much faster than before.

4.

First, I will take a look at the fields in the database using the following code in R:

```
dbListFields(db,"airlines_data")
# [1] "Year"              "Month"             "DayofMonth"
# [4] "DayOfWeek"         "DepTime"           "CRSDepTime"
# [7] "ArrTime"           "CRSArrTime"        "UniqueCarrier"
# [10] "FlightNum"         "TailNum"           "ActualElapsedTime"
# [13] "CRSElapsedTime"    "AirTime"           "ArrDelay"
# [16] "DepDelay"          "Origin"            "Dest"
# [19] "Distance"          "TaxiIn"            "TaxiOut"
# [22] "Cancelled"         "CancellationCode"  "Diverted"
# [25] "CarrierDelay"      "WeatherDelay"      "NASDelay"
# [28] "SecurityDelay"     "LateAircraftDelay"
```

From the result above I found that following are the fields we are interested in: 2."Month", 4."DayOfWeek", 6."CRSDepTime", 9. "UniqueCarrier" 17."Origin" 18."Dest"

Next, I will store this index in a vector called f_ index and direct extract those columns out of the bzip files, using almost the same code in ps2. Those colunms are written and zip again, storing in newyear.csv.bz2.

```
#I used hard coding here, as the selected columns are limited.
f_index=(2,4,6,9,17,18)
for year in {1987..2008};
    do bunzip2 -c ${year}.csv.bz2 | cut -d',' -f$f_index | bzip2 >> new${year}.csv.bz2
    done
```

I got the time of filter step using bash preprocessing as I put following code in a bash script and run:

```
time bash filter.sh
#real  10m5.494s
#user  12m8.324s
#sys  0m24.248s
```

Then I created the database using the filtered data as following:

```
filename<-"airline2.db"
drv<-dbDriver("SQLite")
```

```
db2<-dbConnect(drv,dbname=filename)
#updata the table class, as we just keep several of them.
tableCL2 <- c(rep("integer",3),rep("factor",3))

system.time(for (i in years){
  print(i)
        nm<-paste("bunzip2 -c new",toString(i),".csv.bz2",sep="")
        dt <- fread(nm,header=TRUE, colClasses=tableCL2)
        dbWriteTable(conn=db2,value=dt,name="airlines_data_preprocessed",
                row.names=FALSE,append=TRUE)
})
rm(dt)
```

Here I got the processing time of creating the database after the preprocessing:

user	system	elapsed
246.288	11.060	257.901

Here I found that preprocessing can highly improve the efficiency of creating the database. Although preprocessing cost time, it deserves doing that.