

PS6 - STAT 243

Alexander Fred Ojala
Student ID: 26958060

Collaborators: Milos Atz and Alexander Brandt

November 2nd 2015

1 Problem 1

For Problem 1 the R code below was run in the EC2 environment. First the file was downloaded and unzipped so that we had access to all the .csv files. The important thing to note is that we use *append = TRUE* in the *dbWriteTable* function to sequentially load in the data into one table in the SQLite database, here using a for loop. The implementation is presented below the code.

```
library(RSQLite)
options(stringsAsFactors = FALSE)

## Problem 1 ----- DOWNLOADING
url="http://www.stat.berkeley.edu/share/paciorek/1987-2008.csvs.tgz"
download.file(url,destfile = "data.tgz")
system("tar -xzf data.tgz") #untar the data.tgz file
system("bunzip2 -vv *") #unzip all the bz2-files

#CREATING THE DATABASE

fileName <- "airline.db"
db <- dbConnect(SQLite(), dbname = fileName) #setup connection to a db

yrs=seq(1987,2008)
files=paste(yrs,".csv",sep="") #create string with all file names

#below we read in the data into one table, sequentially for every year
for (i in 1:length(yrs)) {
  dbWriteTable(conn = db, name = "airData", value = files[i],
               row.names = FALSE, header = TRUE, append = TRUE)
  print(i) #only to see the progress
}

#Set numeric code to all the NA values (could be done for any of the numeric columns:
#DepTime, DayOfWeek, Month, DepTime)
#THIS command IS NOT USED, instead subset and remove values DepDelay='NA' in 2 a)
query <- "UPDATE airData SET DepDelay=123456789 WHERE DepDelay='NA'"
dbGetQuery(db, query)
```

The result of doing the above operations (except altering the table, replacing NA's) gives that the file size of *airline.db* is approximately 11.3514GB. So it is a lot larger than the zipped tgz file (1.7GB), but slightly smaller than the original CSV (12GB). See print out of the size from the EC2 instance in the Figure 1.

```
-rw-r--r-- 1 ubuntu ubuntu 11G Oct 31 00:25 airline.db
-rw-rw-r-- 1 ubuntu ubuntu 1.6G Oct 30 23:33 data.tgz
drwxr-xr-x 2 ubuntu ubuntu 4.0K Oct 15 19:57 Desktop
drwxr-xr-x 16 ubuntu ubuntu 4.0K Oct 15 20:30 miniconda
-rwxr-xr-x 1 ubuntu ubuntu 2.0K Oct 15 19:51 setup_ipython_notebook.sh
i> system("stat -c%s airline.db")
11351425024
```

Figure 1: Print out of the storage for the full airline database (airline.db) on the EC2 Virtual Machine

2 Problem 2

2.1 2a)

SQLite solution:

In order to remove the NA values and the unreasonable values (bigger or smaller than two days = ± 2880 minutes) for the DepDelay, I simply deleted them from the database in SQLite (as they would not be of any use later on, also validated that this was OK to do from Chris). This creates the subset we want. This is done with the query below and the database table is changed when we execute the query on the db with the command *dbGetQuery*.

```
#DELETE NA'S AND UNREASONABLE VALUES
query <- "DELETE FROM airData WHERE DepDelay='NA'"
dbGetQuery(db, query)
query <- "DELETE FROM airData WHERE DepDelay > 2880 or DepDelay < -2880"
dbGetQuery(db, query)
```

Spark solution:

In Spark I filtered the dataset and changed the NA's to a negative value that is unreasonable and will not add to the counts of DepDelay greater than or the proportions when doing the grouping by key. In this way they will not add to the total count when we perform that operation (also a confirmed solution). See the last three lines in the function defined in the code chunk below (which is part of the Spark solution in 2b)):

```
#Function below put a key on the equal categories of interest
def computeKeyValue(line):
    vals = line.split(',')
    # keyVals is Carrier-Month-DayOfWeek-DepTime-Origin-Destination
    keyVals = '-'.join([vals[x] for x in [8,1,3,4,16,17]])
    if vals[0] == 'Year':
        return('0', [0,0,0,0,1,1]) #header info
    # FILTER THE DATASET, SET NA DEPDELAYS SO THAT WE DO NOT COUNT THEM
    if vals[15] == 'NA': #Change NA's
        vals[15] = '-99999'
    return(keyVals, [int(vals[15])])
```

2.2 2b)

SQLite solution:

First the DepTime was transformed into an integer only indicating the hour of the day with the Update query below (divide by 100 and cast the result as an integer only extract the integer value, which in SQLite query language is the same as round down to the int value).

After that we specify a function *getStat* that sends a query to only extract the statistics for the number of minutes we want to look at for the DepDelay as its argument. Inside *getStat* we specify one BIG query that carries out all the operations. It categorizes the data set for the columns of interest, subsets it and extracts the total number of flights with regards to the categories, the number of delayed flights greater than the certain amounts of minutes defined for that category, and the proportion of flights delayed compared to the total number of flights in that category. We want it to be in descending order (so that the greatest proportions/fractions come first), then we do not need to sort the result later when we want to extract the top 5 dealed proportions. Then we can extract the summary statistics into R and use system.time to see how much time it takes to carry out. We do this for all categories and calculate the proportions that have more than 30, 60 and 180 minutes in their DepDelay. See the code below for the implementation and the resulting output can be seen in Figure 2

```
#Only shows hour of departure (OK, tested so it rounds down)
query<-"UPDATE airData SET DepTime = CAST(DepTime / 100 AS INT)"
dbGetQuery(db, query)

#Function for extracting proportions Total number of flights,
#Number of delayed flights greater than a certain amounts of minutes and Delayed Proportion
getStat = function(minute) {

  query<-paste("SELECT UniqueCarrier, Origin, Dest, Month, DayOfWeek, DepTime,
                SUM(CASE WHEN DepDelay > ",minute, " THEN 1 ELSE 0 END) AS NoOfDelayed,
                COUNT(*) AS Total, SUM(CASE WHEN DepDelay > ",minute," THEN 1.0 ELSE 0.0 END)
                / COUNT(*) AS DelayedProp FROM airData GROUP BY UniqueCarrier, Origin,
                Dest, Month, DayOfWeek, DepTime ORDER BY DelayedProp DESC")

  result<-dbGetQuery(db,query)

  return(result)
}

system.time(stat30<-getStat(30))
system.time(stat60<-getStat(60))
system.time(stat180<-getStat(180))
```

```
ubuntu@ip-172-31-27-70: ~ — ss...
> system.time(stat30<-getStat(30))
  user  system elapsed 
599.968  32.064 675.815 
> system.time(stat60<-getStat(60))
  user  system elapsed 
599.084  31.788 660.227 
[> system.time(stat180<-getStat(180))
  user  system elapsed 
601.832  33.616 663.297
```

Figure 2: The time it took to extract the summary statistics for the categories for the different values DepDelay greater than 30, 60 and 180 minutes on the EC2 instance

Spark Solution

In Spark I started an instance with 12 nodes/workers according to the instructions. Then I downloaded and unzipped all the data files as well as installed pyspark according to Chris' solutions. Locally I wrote a .py script that I submitted via Cyberduck (thanks Harold!) and then I ran that on the Spark cluster with `./spark/bin/spark-submit code.py`. We repartitioned the data so that it could be distributed across the cores. The script submitted (together with comments for what it does) can be seen below with comments.

N.B. Here the counts needed for all the number of DepDelays greater than 30, 60 or 180 minutes for each category are computed and added in the correct format. The proportions are never explicitly calculated, since it said in the problem set that we only needed to collect *the counts needed to compute the proportions without actually finishing the calculation of computing the proportions*. Also when we read in the data into spark with `lines = sc.textFile('/data/airline')` the DepTime is of a strange format so I could not use the length of the string and the position of the numbers to determine the hour, and also it was impossible to cast it as a float, divide it by 100, use the integer part and cast that as a character, hence I could not transform it into an integer/float value and divide it by 100 (I tried several different implementations, but Spark always aborted and returned an error, even though it worked locally when doing it on similar DepDelay strings in python). However to extract the correct groupings and proportions, with the correct format of the DepTime, this is only post processing that can be done by reducing by key again when we read in the results from below and calculate proportions. **The solution computes the counts needed to compute the proportions**, which was the only thing asked for in the PS manual, hence I did not carry out the post-processing operations as explained above.

```
from operator import add
import numpy as np
from pyspark import SparkContext #in order to read in the db

sc = SparkContext()

lines = sc.textFile('/data/airline') #make RDD connection
lines = lines.repartition(96).cache() #divide into 96 jobs

#Function below put a key on the equal categories of interest
def computeKeyValue(line):
    vals = line.split(',')
    # keyVals is Carrier-Month-DayOfWeek-DepTime-Origin-Destination
    keyVals = '-'.join([vals[x] for x in [8,1,3,4,16,17]])
    if vals[0] == 'Year':
        return('0', [0,0,0,0,1,1]) #header info
    # 15 is Departure delays
```

```

    if vals[15] == 'NA': #Change NA's
        vals[15] = '-99999'
    return(keyVals, [int(vals[15])])

#Function that counts the Delays that are greater than 30, 60 and 180
def countTimes(x):
    key = x[0]
    if len(x)==2: #defensive programming
        if len(x[1])>0: #defensive programming
            DepTimes = x[1]
            totalFlights = len(DepTimes) #count total number of flights
            #array with totalFlights, cnt30, cnt60, cnt180
            countArray = [totalFlights, 0, 0, 0]
            #Below Checks all departure times, which are greater
            for i in range(0,totalFlights):
                if DepTimes[i] > 30:
                    countArray[1] = countArray[1] + 1
                if DepTimes[i] > 60:
                    countArray[2] = countArray[2] + 1
                if DepTimes[i] > 180:
                    countArray[3] = countArray[3] + 1
            return(key,countArray)
        else:
            return(key,[-123456789,-123456789,-123456789,-123456789]) #return error
    else:
        return(key,[-987654321,-987654321,-987654321,-987654321]) #return different error

#Function that converts output to correct string format
def stringConvert(x):
    key = x[0]
    countArray = x[1]
    output = str(key)+' ',''+str(countArray[0])+' ',''+str(countArray[1])+' ',''+str(countArray[2])+' '
    ,'+str(countArray[3])
    return(output)

#The stuff below was used for comparing to the SQLite results, and the results where equal
# It was NOT USED for the solution in 2 c)
allDepDelays = lines.map(computeKeyValue).reduceByKey(add) #Extract all depDelays
countDelays = allDepDelays.map(countTimes).map(stringConvert).collect() #Count the Delays
print countDelays[0:1000]

```

Result for Spark and Comparison with RSQLite:

The time it took to carry out the operations above and collecting all the information needed can be seen in Figure 3. The time was obtained by running the command `time ./spark/bin/spark-submit code.py` in Spark (it prints the time to carry out all the operations in the code chunk). The time obtained was for writing the results to a text file, as in Problem 2 c).

This is the total time for getting all the data for all the depDelays greater than 30, 60 and 180. This

operation, doing it three times in RSQLite on the EC2 instance took about 676+660+663 seconds which is roughly 33 mins (elapsed time). This shows that Spark is faster (16 minutes elapsed time), roughly half of the time for gathering the results than RSQLite. (However we could have done the query in RSQLite a bit faster, by extracting all the summary statistics for 30, 60 and 180 at once).

```
15/11/01 05:59:16 INFO util.ShutdownHook$
real    16m26.191s
user    0m30.174s
sys     0m5.324s
root@ip-172-31-3-88 ~]$
```

Figure 3: Time for running the Spark script, gathering all the data needed to collect all summary statistics for every category and their counts

2.3 2c)

Below the one line of code is presented for how I saved the output (the resulting aggregated dataset) as a single data file on the master node. And in Figure 4 you can see the file size of the total data stored on disk in the Spark master node as well as a print out of the first values in the output data file.

```
#We carry out all the operations from the pyspark script shown in prob 2 b)
#(except for the last comparison lines)
#To print everything out we use "map piping" on the data in lines
#repartition(1) saves the result to the master node

lines.map(computeKeyValue).reduceByKey(add).map(countTimes).
  map(stringConvert).repartition(1).saveAsTextFile('/data/airline/dat.txt') #write to file
```

```
root@ip-172-31-3-88 dat.txt]$ ls -lah
total 2.1G
drwxr-xr-x  2 root root 4.0K Nov  1 05:29 .
drwxr-xr-x 21 root root 4.0K Nov  1 05:27 ..
-rw-r--r--  1 root root 2.1G Nov  1 05:28 part-00000
-rw-r--r--  1 root root    0 Nov  1 05:27 _SUCCESS
root@ip-172-31-3-88 dat.txt]$ ls -l
total 2117116
-rw-r--r--  1 root root 2167922012 Nov  1 05:28 part-00000
-rw-r--r--  1 root root          0 Nov  1 05:27 _SUCCESS
root@ip-172-31-3-88 dat.txt]$ head part-00000
NW-8-3-1401-MEM-MSP,2,0,0,0
UA-2-4-1716-BOS-SFO,4,0,0,0
YV-6-7-1515-BUR-PHX,1,0,0,0
CO-11-5-1940-OKC-IAH,1,0,0,0
UA-6-7-1835-MCO-IAD,4,0,0,0
NW-8-5-815-BDL-MSP,1,0,0,0
B6-12-2-948-FLL-JFK,1,0,0,0
DL-12-7-1247-DCA-CVG,2,0,0,0
WN-11-3-1855-HOU-BHM,1,1,1,0
XE-9-6-1502-DTW-EWR,1,0,0,0
root@ip-172-31-3-88 dat.txt]$
```

Figure 4: File size of the output file created with spark as well as the first lines stored in the data file

2.4 2d)

Here we add an index to the `airData` table in the database for all the columns of interest when we group/-categorize. We update the database with a `dbGetQuery`, so that the index is added, and then we carry out the same operations as in Question 2d). As can be seen the times to make the query are more than twice as fast when we add an index.

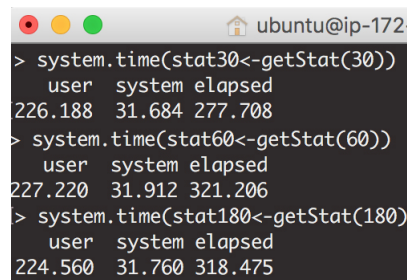
N.B. After every read in we delete the variable `stat30`, `stat60` or `stat180` so that we have an equal amount of RAM available when make a new query for the next minute specification.

The resulting output can be seen in Figure 5

```
query<-"CREATE INDEX idx ON airData(UniqueCarrier, Origin, Dest, Month, DayOfWeek, DepTime)"
dbGetQuery(db, query)

system.time(stat30<-getStat(30))
rm(stat30)
system.time(stat60<-getStat(60))
rm(stat60)
system.time(stat180<-getStat(180))
rm(stat180)
```

In total this took 277+321+318, which is about 15 minutes. This is equivalent to the speed it took to carry out the operations in Spark for problem 2 b) (but then we do not take into account the actual time it took to add an index to the database table).



```
ubuntu@ip-172-
> system.time(stat30<-getStat(30))
  user  system elapsed 
226.188   31.684  277.708 
> system.time(stat60<-getStat(60))
  user  system elapsed 
227.220   31.912  321.206 
> system.time(stat180<-getStat(180))
  user  system elapsed 
224.560   31.760  318.475
```

Figure 5: The time it took to categorize and get the proportion for the different values of DepDelay greater than 30, 60 and 180 minutes WITH an index added to the database table grouping columns, done on the EC2 instance

2.5 2e)

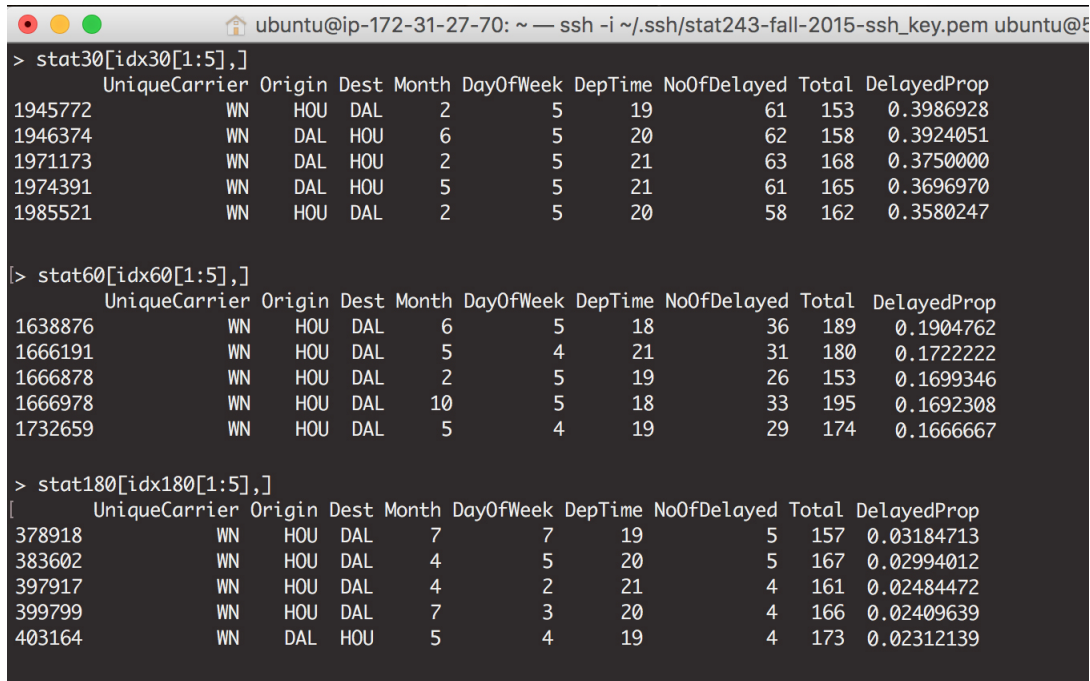
I used the results from SQLite on the EC2 instance to print out the top 5 groupings for DelayDep greater than 30, 60 and 180 with at least 150 flights in total for that category. In order to do this we need to first find all the indices for groupings/categories where the total number of flights is greater or equal to 150 for every group (since it is an integer value we can take $\lceil 149 \rceil$). The results in the different `stat` variables are already ordered (done in the query) so that the highest proportion of late flights will be at the top of the results (it is ordered by DelayedProp in descending order). Therefore we only need to print the first five values of the index vectors formed for total flights greater or equal to 150. The implementation can be seen in the code chunk below and the results are presented in Figure 6. As can be seen the carrier WN that have many flights on the route from (or to) HOU to (or from) DAL seems to have had the most problems with delayed flights in all DepDelay minute categories.

```

#Find indicies for the groups where total number of flights 150 or greater
idx30<-which(stat30$Total>149)
idx60<-which(stat60$Total>149)
idx180<-which(stat180$Total>149)

#Print the first five values, they are already sorted in the statXYZ variables
#So the highest proportion of Delayed flights will be at the top of the variable
stat30[idx30[1:5],]
stat60[idx60[1:5],]
stat180[idx180[1:5],]

```



```

> stat30[idx30[1:5],]
  UniqueCarrier Origin Dest Month DayOfWeek DepTime NoOfDelayed Total DelayedProp
1945772        WN   HOU  DAL     2         5      19           61    153  0.3986928
1946374        WN   DAL  HOU     6         5      20           62    158  0.3924051
1971173        WN   DAL  HOU     2         5      21           63    168  0.3750000
1974391        WN   DAL  HOU     5         5      21           61    165  0.3696970
1985521        WN   HOU  DAL     2         5      20           58    162  0.3580247

[> stat60[idx60[1:5],]
  UniqueCarrier Origin Dest Month DayOfWeek DepTime NoOfDelayed Total DelayedProp
1638876        WN   HOU  DAL     6         5      18           36    189  0.1904762
1666191        WN   HOU  DAL     5         4      21           31    180  0.1722222
1666878        WN   HOU  DAL     2         5      19           26    153  0.1699346
1666978        WN   HOU  DAL    10         5      18           33    195  0.1692308
1732659        WN   HOU  DAL     5         4      19           29    174  0.1666667

> stat180[idx180[1:5],]
  UniqueCarrier Origin Dest Month DayOfWeek DepTime NoOfDelayed Total DelayedProp
378918        WN   HOU  DAL     7         7      19           5    157  0.03184713
383602        WN   HOU  DAL     4         5      20           5    167  0.02994012
397917        WN   HOU  DAL     4         2      21           4    161  0.02484472
399799        WN   HOU  DAL     7         3      20           4    166  0.02409639
403164        WN   DAL  HOU     5         4      19           4    173  0.02312139

```

Figure 6: The top 5 or 10 groupings in terms of proportion of late flights for groupings with at least 150 flights

3 Problem 3

For problem 3 we used parallelization on the m3.xlarge EC2 instance (4 cores) in order to speed up the queries even more. A sequence of subqueries was done for the DepTime. The unique values was obtained by a query `verb;depT;-dbGetQuery(db,"SELECT DISTINCT DepTime FROM airData")`; which gave sequence of values from midnight (00:01, presented as 1, is just 0) to midnight (24:00, presented as 2400, is 24), hence 25 subqueries in total. Below only defined as `depT=seq(0,24)`. Also a new function `taskFun` was built based on the `getStat` function that is being called in `foreach` and `mclapply`. `taskFun` opens a new connection to the database every time a new task was initialized and takes DepTime and minutes as input arguments. Find the implemented solution below and the results after the code chunk.

N.B. A lot of experimentation was done with this, and what I figured was that in order to obtain the speed increase one had to drop the old index, and add a new one for the columns of interest, however indexing on the columns where the subqueries get made FIRST. Hence we create an index on the database table where we index on DepTime first and then the other columns for the groupings/categories.

N.B.2 Parallelization on the database was kind of buggy, some times when I ran it (like indexing and doing sub-queries on the month it actually took a lot longer to run and sometimes it was faster). I also tried modifying the *taskFun* function with the lines below, so that I first removed the database connection with `rm(db)` in R, and established a new one for every task, however it was not quicker with this procedure and `dbSendQuery` compared to not doing it:

```
[...]
#The alternative ending lines for taskFun (seen below), they did not speed up the process.

result<-fetch(dbSendQuery(db,query),n=-1)
dbDisconnect(db)
return(result)
dbClearResult(result)

library(parallel)
library(doParallel)
library(foreach)
library(iterators)

## The old index from 2d) is dropped, add index on DepTime first so that the parallelization
# really speeds up the process
query<-"CREATE INDEX idx ON airData(DepTime, Origin, Dest, Month, DayOfWeek, UniqueCarrier)"
dbGetQuery(db, query)

#Function to get called from mclapply and foreach
taskFun <- function(depT,minute){
  db <- dbConnect(SQLite(), dbname="airline.db")

  query<-paste("SELECT UniqueCarrier, Origin, Dest, Month, DayOfWeek, DepTime,
    SUM(CASE WHEN DepDelay > ",minute, " THEN 1 ELSE 0 END) AS NoOfDelayed,
    COUNT(*) AS Total, SUM(CASE WHEN DepDelay > ",minute," THEN 1.0 ELSE 0.0 END)
    / COUNT(*) AS DelayedProp FROM airData WHERE DepTime=",depT," GROUP BY
    UniqueCarrier, Origin, Dest, Month, DayOfWeek, DepTime ORDER BY
    DelayedProp DESC",sep="")

  result<-dbGetQuery(db,query)

  return(result)
}

depT<-seq(0,24) #all the unique Departure Times (Integer hours)
nCores <- 4 # to set manually, can be found by detectCores()
registerDoParallel(nCores) #For foreach

##mclapply solution
system.time(res<-mclapply(depT, taskFun, mc.cores = nCores,minute=30))
rm(res)
system.time(res<-mclapply(depT, taskFun, mc.cores = nCores,minute=60))
rm(res)
system.time(res<-mclapply(depT, taskFun, mc.cores = nCores,minute=180))
rm(res)
```

```

#foreach solution
mins=c(30,60,180) # run for all three, will print out system.time
for (j in 1:length(mins)) {
  print(mins[j])
  print(system.time(out <- foreach(i = 1:length(depT)) %dopar% {
    cat('Starting ', i, 'th job.\n', sep = '') #print job status
    outSub <- taskFun(depT[i],minute=mins[j])
    cat('Finishing ', i, 'th job.\n', sep = '') #print job status
    outSub # part of the out object
  })))
  rm(out) #to clear up RAM
}

```

As can be seen below from the resulting output picture the parallelization solution for DepDelay greater than 30 minutes was about 50 seconds faster than the indexed solution in 2d), which is about 20 percent faster in comparison to doing the full query. foreach gave the same result. This was kind of buggy on the m3.xlarge EC2 instance (sometimes the elapsed time was much greater), but it worked everytime on my local machine with four cores, giving a slightly greater speed up (about 40 percent).

```

[> system.time(res<-mclapply(depT, taskFun, mc.cores = nCores,minute=30))
      user  system elapsed
301.260   67.192  225.960

```

Figure 7: Speed when applying mclapply and doing parallelization on the EC2 instance (m3.xlarge)

4 Problem 4

In order to cut out the columns not needed R was first used to extract the indicies of the columns needed and to print them to a file *index.txt* that could be accessed in bash (A Piazza note later said that we could hard code the column indicies, but I kept my solution). Could also have extracted this info in bash, but the R solution is more elegant:

```

head<-readLines(bzfile("1987.csv.bz2"),1) # Extract data header

cols=c("UniqueCarrier", "Origin", "Dest", "Month", "DayOfWeek", "DepTime", "DepDelay")
#specify the columns to work with

head=strsplit(head,",") #split the strings
headtxt<-unlist(head) #unlist result
index=rep(0,length(cols)) #find indices
for (i in 1:length(cols)) {
  index[i]=match(cols[i],headtxt)
}
index=sort(index) #index in the right order

cat(index,sep=",",file="index.txt") # save indices to a text file.

```

Once we had the column indices we could run the following bash script as a for loop for all the .bz2 zipped year files on the EC2 m3.xlarge instance and pipe out the cut columns (with an added ',' delimiter between the data entries) into new zipped bz2-files (without unzipping). The code below prints the time the operation took for each year. The year is printed with *echo* to show progress:

```
for yr in {1987..2008}
do
time (bunzip2 -c ${yr}.csv.bz2 | cut -d, -f$(cat index.txt) | bzip2 > ${yr}clean.csv.bz2)
echo $yr
done
```

When this code was run on a m3.xlarge instance it took 722 seconds to pre-process all the bz2-files (about 12 minutes real/elapsed time). They take up about 1/4 of storage compared to the original zipped bz2 files. This pre-processing step would also speed up all the operations above, since we would only need to work with a subset of the data set in our database. I would definitely make this pre-processing if I worked with a large data set.

You can see the time result in figure 8 below:

```
ubuntu@ip-172-31-27-70: ~ -- ssh -i ~/.ssh/stat243-fall-2015-ssh_key.pem ub...
ubuntu@ip-172-31-27-70:~$ for yr in {1987..2008}
> do
> time (bunzip2 -c ${yr}.csv.bz2 | cut -d, -f$(cat index.txt) | bzip2 > ${yr}cle
an.csv.bz2)
> echo $yr
/> done

real    0m5.889s   real    0m25.035s   real    0m36.963s
user    0m7.132s   user    0m30.452s   user    0m45.612s
sys     0m0.184s   sys     0m1.576s   sys     0m1.460s
1987

real    0m23.072s   real    0m25.221s   real    0m34.201s
user    0m28.164s   user    0m31.108s   user    0m42.188s
sys     0m0.716s   sys     0m1.244s   sys     0m1.340s
1988

real    0m23.425s   real    0m25.064s   real    0m34.371s
user    0m29.264s   user    0m31.192s   user    0m42.736s
sys     0m0.888s   sys     0m0.920s   sys     0m0.972s
1989

real    0m25.509s   real    0m28.145s   real    0m38.764s
user    0m31.164s   user    0m33.972s   user    0m47.060s
sys     0m0.864s   sys     0m1.812s   sys     0m1.780s
1990

real    0m22.362s   real    0m29.107s   real    0m34.174s
user    0m27.264s   user    0m35.700s   user    0m42.620s
sys     0m1.084s   sys     0m1.100s   sys     0m1.192s
1991

real    0m21.856s   real    0m28.150s   real    0m38.764s
user    0m27.048s   user    0m35.056s   user    0m47.060s
sys     0m1.152s   sys     0m1.008s   sys     0m1.192s
1992

real    0m22.053s   real    0m24.885s   real    0m34.174s
user    0m27.672s   user    0m31.056s   user    0m42.620s
sys     0m0.588s   sys     0m0.828s   sys     0m1.192s
1993

real    0m22.232s   real    0m33.186s   real    0m34.174s
user    0m27.764s   user    0m41.344s   user    0m42.620s
sys     0m0.776s   sys     0m1.120s   sys     0m1.192s
1994

real    0m26.763s   real    0m32.900s   real    0m34.174s
user    0m32.900s   user    0m41.344s   user    0m42.620s
sys     0m1.020s   sys     0m1.120s   sys     0m1.192s
1995
ubuntu@ip-172-31-27-70:~$
```

Figure 8: Timing of pre-processing and only cutting out columns we want to work with in bash (Includes three columns of output in the same picture)

And it can also be validated that it works if we print out the first five values of a pre-processed bz2 file. See the bash output from the command below:

```
ubuntu@ip-172-31-27-70:~$ bunzip2 -c 1992clean.csv.bz2 | head -5;
Month,DayOfWeek,DepTime,UniqueCarrier,DepDelay,Origin,Dest
1,4,748,US,-2,CMH,IND
1,5,750,US,0,CMH,IND
1,6,747,US,-3,CMH,IND
1,7,750,US,0,CMH,IND
```