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 in the for loop. The result 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 -xzvf data.tgz")
system("bunzip2 -vv *")


#CREATING THE DATABASE

fileName <- "airline.db"
db <- dbConnect(SQLite(), dbname = fileName) #setup clean database

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

#below we read in the data into the table 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 columns)
#THIS command IS NOT ACCTUALLY USED, instead we remove the values for DepDelay='NA' in 2 a)
query <- "UPDATE airData SET DepDelay=123456789 WHERE DepDelay='NA'"
dbGetQuery(db, query)
```

The result of doing this is that the file size of *airline.db* is approximately 11.3514GB. So it is a lot larger than the zipped bz2 file (1.7GB), but slightly smaller than the original CSV (12GB). See print out 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
> 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)

RSQLite 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 databse in SQLite (as they would not be of any use later on, also validated that this was OK to do from Chris). This is done with the query below and the database 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 we only changed the NA's to a negative value that is unreasonable when doing the grouping by key. In this way they will not add to the total count when we perform that operation (this was an OK solution as per instructions on OH). See the last three lines in the code chunk below (which is also 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
    # OMIT THE NA DEPDELAYS SO THAT WE DO NOT COUNT THEM WRONG
    if vals[15] == 'NA': #Change NA's
        vals[15] = '-99999'
    return(keyVals, [int(vals[15])])
```

2.2 2b)

RSQLite solution:
First the DepTime was transformed into an integer only showing the hour of the day with the Update query

below (divide by 100 and only extract the integer value, equal to rounding down).

After that we specify a function *getStat* that queries to only extract the number of minutes we want to look at for the DepDelay as its argument. Inside we specify one BIG query that 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 for that category, and the proportion of flights delayed. We want it to be in descending order (so that the greatest proportions come first), then we do not need to sort the data set later, when we want to extract the top 5 dealyed proportions. Then we can extract the values of interest into R and use system.time to see how much time it takes to carry out the operations. We do this for all categories 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

```r
#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))
```

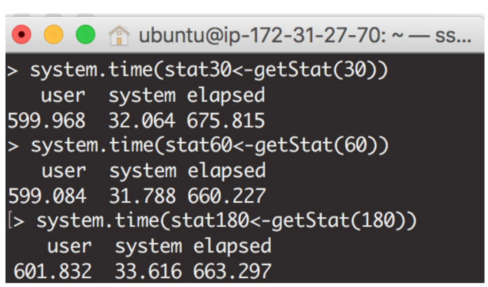


Figure 2: The time it took to categorize and get the proportion for the different values DepDelay greater than 30, 60 and 180 minutes on the EC2 instance

Spark Solution

For 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`. The script submitted 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 could only 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 formatted in a way that 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 in python). However to get the correct groupings and proportions this is only post processing that can be done by reducing by key again when we calculate the proportions. Since this was not necessary for this problem set, and we only needed to compute the counts, the solution below should be satisfactory.

```
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


def stringConvert(x): #Convert output to correct string format
    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 commented out 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[0:500]
```

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 the Spark cluster (it will print out 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. This shows that Spark is faster (16 minutes, 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 values at once).

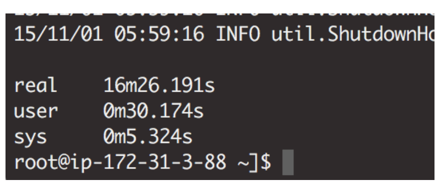


Figure 3: Time for running the spark script, gathering all the data needed to get all the counts needed

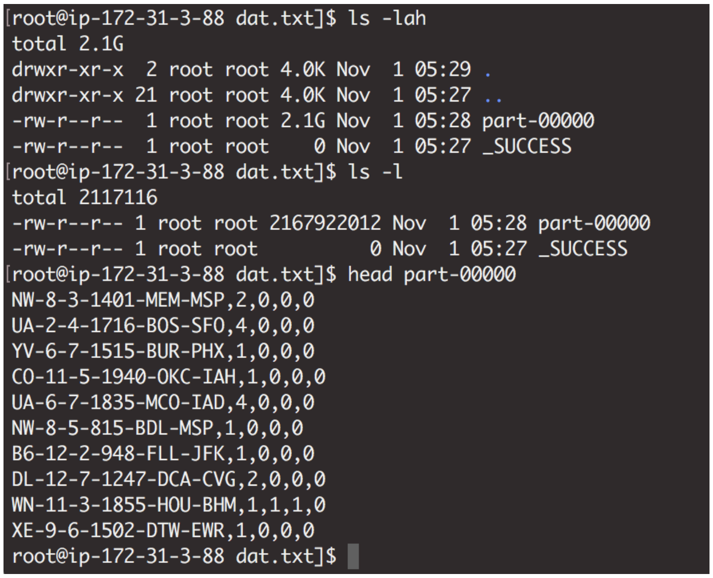
2.3 2c)

Below the line is shown for how to save the output as a file. 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 the first values stored in the output 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 database for all the columns of interest when we group/categorize. We update the database with a *dbGetQuery* 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 *stat*30, *stat*60 or *stat*180 so that we have enough RAM to make the 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))
system.time(stat60<-getStat(60))
system.time(stat180<-getStat(180))
```

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 time it took to add an index to the database.

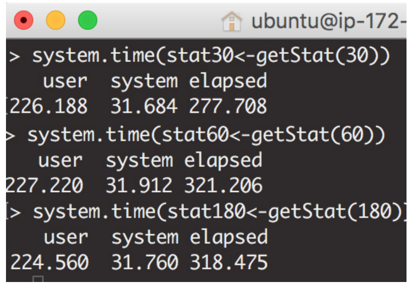


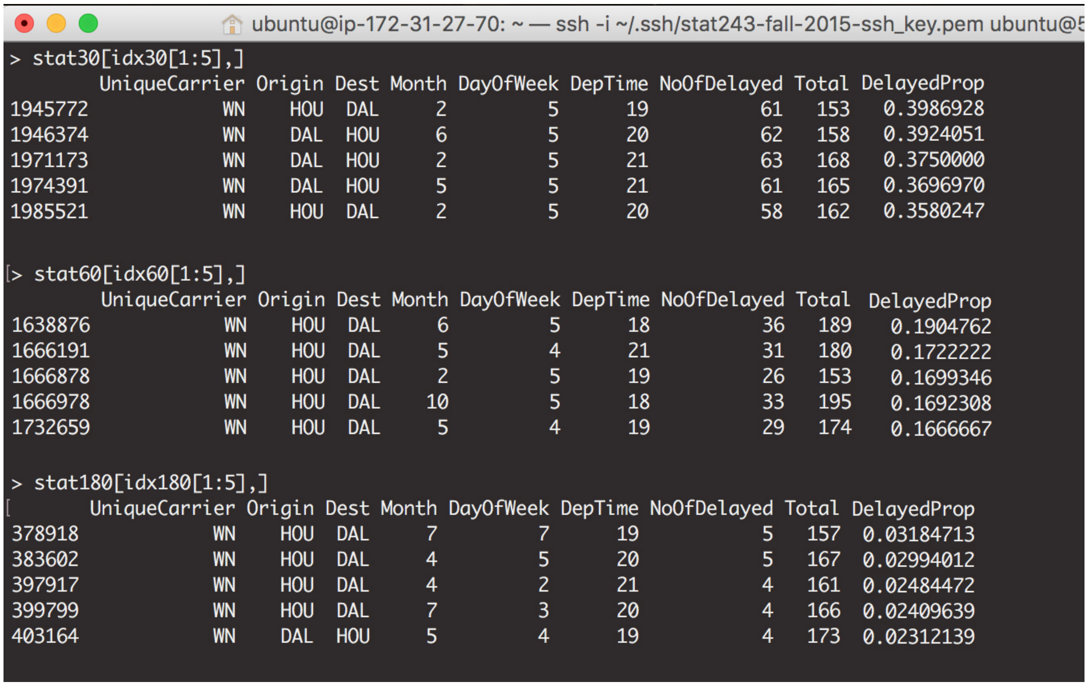
Figure 5: The time it took to categorize and get the proportion for the different values DepDelay greater than 30, 60 and 180 minutes WITH an index added to the databaseon the EC2 instance

2.5 2e)

I used the results from SQLite to print out the top 5 groupings for DelayDep greater than 30, 60 and 180 with at least 150 flights in total from the output in the EC2 instance. In order to do this we need to first find all the indicies for the total number of flights greater or equal to 150 for every group. Since the results in the *stat* variables are ordered so that the highest proportion of late flights will be at the top of the variable (it is ordered by DelayedProp in descending order). The results can be seen below in Figure 6.

```r
#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],]
```



```
ubuntu@ip-172-31-27-70: ~ — ssh -i ~/.ssh/stat243-fall-2015-ssh_key.pem ubuntu@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 in order to speed up the queries even more. A sequence of subquery was established for the DepTime (sequence of values from 0 to 24, hence 25 subqueries in total). Also for the function being called a new connection to the database was opened every time a new task was initialized. 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.

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 follwoing lines, first I removed the database connection with rm(db) in R, however it was not quicker with dbSendQuery, and clearing the db and the results 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)
```

```r
library(iterators)

## We have dropped the old index and instead 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))
system.time(res<-mclapply(depT, taskFun, mc.cores = nCores,minute=60))
system.time(res<-mclapply(depT, taskFun, mc.cores = nCores,minute=180))

#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 = '')
    outSub <- taskFun(depT[i],minute=mins[j])
    cat('Finishing ', i, 'th job.\n', sep = '')
    outSub # part of the out object
  }))
}
```

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 perecent faster in comparison to doing the full query. foreach gave the same result. This also worked on my local machine with four cores, giving a slightly greater speed up (about 40 percent).

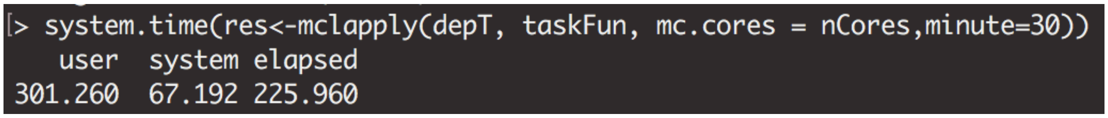


Figure 7: Speed when applying mclapply and doing parallelization on the EC2 instance (m3.xlarge)

4 Problem 4

In order to cut out the columns not needed R was first used to extract the columns of interest and their indicies with the code below:

```r
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 indicies we could run the following bash script in Unix in order to pipe out the output into new bz2-files, and also getting the time for each year.

```bash
for yr in {1994..1996}
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). They take up about 1/4 of storage compared to the original zipped files. 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:

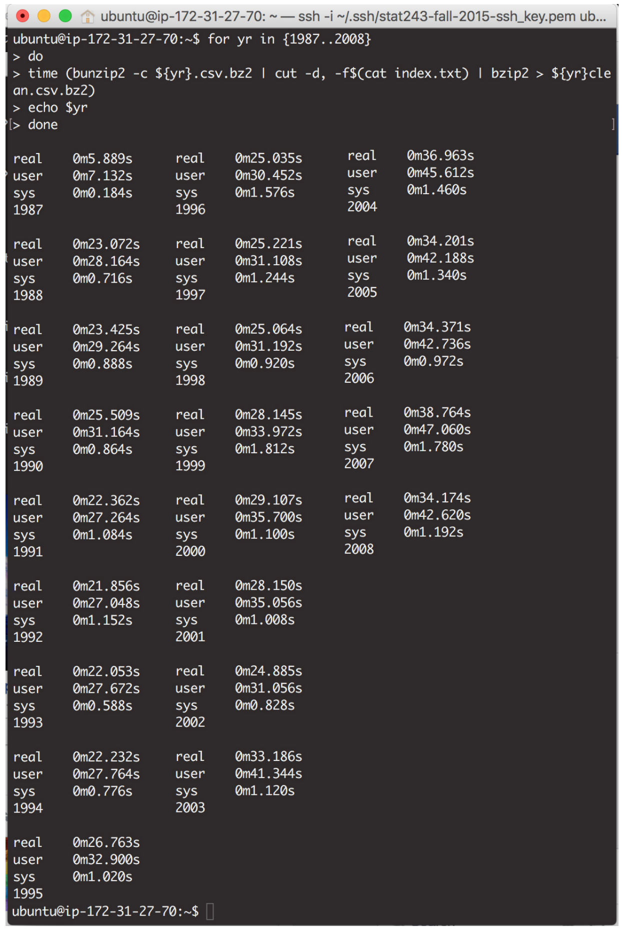


Figure 8: Timing of pre-processing and only cutting out columns we want to work with in bash (Includes three columns of output in the same picture)

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
```