STA 242: Assignment 4

Olivia Lau

March 5, 2013

1 Introduction

One of the common issues faced today with the superfluous amount of data is being able to appropriately utilize it in a manageable fashion. Here, we are presented with airline data for all US flights from 1987 until 2008. This report will explore several different avenues: shell (and a bit of R), R, and SQL. And with these tools, we will endeavor to accomplish three tasks: count the number of flights departing from LAX, OAK, SFO, and SMF, and calculate the mean and standard deviation of Arrival Delays from LAX, OAK, SFO, and SMF. Although these calculations seem easy enough, the computing time is dramatic for a set with a total of 122,464,881 flights. The largess of this data is nothing short of massive, thus performing even simple calculations can prove to be a timely task.

2 Shell

The shell, as archaic as it looks, is a very powerful tool. In its essence, it is a command line interface that enables a user to communicate with the operating system. Where there are a range of commands, such as the commonly utilized `ls`, `cd`, `cp`, etc., we can use a more diverse set of commands to accomplish the task at hand. The key commands used to count the lines were `cut`, to grab only the 17th element in a comma separated line, `grep`, to pick the elements that matched our desired airports`sort`, to, well, sort, `uniq`, and finally to count the number of unique lines. But rather than entering each command, we made good use of the pipe, '|'. This allowed us to funnel the results from one operation in one smooth transaction. At the end of it we were only given the desired results.

To address the calculation of means and standard deviation, we made use of shell and R together. This time, we used the commands `awk`, to return individual lines that matched the airport, and `cut`, to grab only the arrival delay times. Easily enough, we can pass this up to R to quickly calculate the mean and standard deviation using the `mean()` and `sd()` commands. Note: an alternative way of doing this piece would be to do the calculations in R in blocks, but we explore that in the next section.

Shell has proven itself to be very quick, despite the fact that it applies the commands line-by-line. Although one line is also a block, it may be interesting to consider larger blocks of calculations. A slight nuance to also be conscious of is the encoding of the file. Some of the files has special characters which are not recognizable by the default. One remedy I used here was to set the language for the entire environment using `LC_ALL = C`.

Table 1 and 2 displays the results.

	Mean
LAX	4057452
OAK	1151897
SFO	2711958
SMF	806133

Table 1: Flight counts in Shell

	LAX	OAK	SFO	SMF
mu	6.007046	5.07336	7.95151	5.359327
sd	27.30415	22.05919	30.28721	25.09679

Table 2: Mean and Standard Deviation using Shell and R

3 R

R unfortunately would have a very difficult time managing a dataset of this magnitude. Although it can handle large pieces in its own right, this dataset is a whole other league. But that doesn't cause us to rule out R altogether. Instead, we can use a connection to read a file (or files) in and extract only the necessary information. As opposed to actually reading our file as a table, `readLines()` essentially treats a line as a string of characters rather than attempting to parse and interpret each entry. A connection becomes useful as we are able to read a block of lines and pick up where we left off. We approached this in two manners: (1) saving all the raw delay times and simply computing the mean and standard deviation with R's standard functions; and (2) calculating simple math in blocks. The former (1) was perhaps the easiest to implement, but not necessarily the cheapest way. For (1), I also did everything in R including the parsing. So I used the `file()` connection.

The latter was slightly more involved. Here I incorporated a bit of shell with the connection by using `pipe()` to parse a large portion of it in shell. We calculated the delay sums, delay sums squared, and the delay counts. This avoids saving all the delay times. Although this was the most reasonable way to handle this mass of data, it took quite a bit of time. We did not run the calculations on the entire span of years using blocking and our own blocking, but verified the function on a smaller subset of years compared across all methods. (The "easy" results using only R resembled closely to Table 1 and 2.) It should be noted that the flight count and the delay count are not equivalent – the delay count does not include 'NA' delays. I believe that this portion is written fairly poor, there could be more efficient coding on my part here.

4 SQL

Databases are an important part of dealing with large sets of data and SQL is one language for accessing this information, typically for relational databases. For this particular problem, we do not utilize the relational aspect of databases. Using all of the .csv files, we create a database with two tables – one with the full set of variables, and another with the arrival delays for our airports of interest. One thing we did to speed up some computation was to index the database by certain variables of interest like year and origin. SQL is well versed in handling large sets of data, so it is no surprise that it was able to handle such computations easily.

One nuance of SQLite, a slight variation of SQL, and doesn't not have all of the same syntax or functionality. For example`STDEV` and `SQRT` not a function thus we had to use another function. Another issue that arose was how SQL handled 'NA's. This was brought to our attention when comparing answers between the different approaches the mean was slightly off in initial stages (See Table 4). To remedy this, we removed all values matching 'NA' before calculating the mean and standard deviation in SQL. Table 5 displays our final results. Another note is that the SQLite database is saved locally, so there is no barrier for transmission, logging in, etc..

	Count	Mean
LAX	4057452	5.89296
OAK	1151897	5.00703
SFO	2711958	7.77490
SMF	806133	5.29945

Table 3: Initial run of SQL code

5 Speed and Memory

When running in shell, one thing we used to track the processes was the table of processes, with the `top` command. This illustrated the inner workings to process the commands in real time. What was most

	Count	Mean	Variance	SD
LAX	4057452	6.007046	745.5163	27.30414
OAK	1151897	5.073360	486.6076	22.05918
SFO	2711958	7.951510	917.3150	30.28721
SMF	806133	5.359327	629.8483	25.09678

Table 4: Final Calculations in SQL

useful of this was its break down of %CPU usage, as well as the time incurred for that one command. Not surprisingly, `egrep` is taking most of this up. In total, the time elapsed for merely counting the lines took just under 20 minutes. Although this may seem like quite a while, it is far more time consuming in R solely. When the same calculations were implemented solely in R, it still managed to take a long while longer than the alternatives. Thus, we compared computing time for computing airline statistics on a single file. Table 7 illustrates this.

	User	System	Elapsed
Shell	62.899	0.490	52.094
R	231.611	1.338	233.991
SQL	11.284	6.640	17.832

Table 5: System time for counting lines in a single file

One important aspect aside from processing time is the memory usage incurred. Unfortunately, we do not have Windows or we could have utilized the `memory.size()` command before and after running processes. Without knowledge of other comparable functions, we will merely infer the differences in memory usage. For SQL, it is easiest to imagine since it is stored in a database that is quickly accessible by SQL queries thus little to no memory is used here. Next, in R solely, we read the files in B blocks versus in shell, we actually read one line at time. Intuitively, one would imagine R to be faster at the computation, but instead it is by far the slowest. One of the most obvious points is that shell can pipe the answers smoothly from one job to another and its generality.

6 Conclusion and Discussion

This project addressed several different important interesting topics. Firstly, it was a comparison of three different approaches to a single problem. In this, we had to grasp shell commands, R commands, as well as SQL queries. Moreover, we had to understand computing arithmetic in blocks. I had underestimated this part of the project, hence initially saving all the arrival delay times (not memory efficient). A better approach to truly blocking the computations in R is an aspect I'd like to explore further, especially as it relates back to the idea of parallel computing of performing computations in pieces independently and then bringing it back. Shell scripting and SQLite was a new aspect for me as well. Although I use some simple shell commands daily, this task really coerced some exploration beyond that. SQLite also differs slightly from what I have used in the past, so learning what functionality was and wasn't there was essential.

7 Appendix B: Code

```
setwd("~/Documents/STA 242/Assignment4/Data")
options(warn = -1)

#Goal: tally line counts for each "ORIGIN" airport, mean and standard deviation
#Compare in Shell(+R), only R, SQLite

strsplit(readLines("2000.csv", 1), ",") ##17 = ORIGIN, #15 = ARRDELAY
airports = c("LAX", "OAK", "SFO", "SMF")
filename = list.files(pattern = "[12]*.csv")
###############
#LAX: 4057452,5.89296459945798
#OAK: 1151897,5.00703448311785
#SFO: 2711958,7.77489732510607
#SMF: 806133,5.29944686546761
###############

###############
#####IN R#####
###############

countOriginsR = function(origin, B = 1000, files){
  total = structure(integer(length(origin)), names = origin)
  sub = sapply(files, function(x){
  con = file(x, "r")
  while(TRUE){
    ll = readLines(con, n=B)
    if(length(ll) == 0)
      break
    tmp = sapply(strsplit(ll, ","), `[[`, 17)
    subCounts = table(factor(tmp, origin))
    subCounts[is.na(subCounts)] = 0
    total = total + subCounts
  }
  close(con)
  total})
  apply(sub, 1, sum)
}

countOriginsR(airports, 1000, files = filename)

delayStatR1 = function(origin, B = 1000, files){
  con = file(files, "r")
  delay = integer()
  while(TRUE){
    ll = readLines(con, n = B)
    if(length(ll) == 0)
      break
    tmp = sapply(strsplit(ll, ","), `[[`, 17)
    dat = as.integer(sapply(strsplit(ll[tmp == origin], ","), `[`, 15))
    delay = c(delay, dat)
  }
  list(mu = mean(delay, na.rm = TRUE), sd = sd(delay, na.rm = TRUE))
}

sapply(airports, delayStatR1, files = filename)

delayStatR2 = function(origin, B = 1000){
  countDelay = structure(integer(length(origin)), names = origin)
  sumDelay = structure(integer(length(origin)), names = origin)
  sumSqDelay = structure(integer(length(origin)), names = origin)
  con = pipe("LC_ALL=C cut -d',' -f15,17 [12]*.csv")
  while(TRUE){
    ll = readLines(con, n = B)
    if(length(ll) == 0)
      break
    tmp = sapply(strsplit(ll, ","), `[[`, 2)
    for(x in 1:length(origin)){
      dat = as.integer(sapply(strsplit(ll[tmp == origin[x]], ","), `[`, 1))
      countDelay[x] = countDelay[x] + length(dat[!is.na(dat)])
      sumDelay[x] = sumDelay[x] + sum(dat, na.rm = TRUE)
      sumSqDelay[x] = sumSqDelay[x] + sum(dat^2, na.rm = TRUE)
    }
  }
  close(con)
  rbind(countDelay, sumDelay, sumSqDelay)
}


myCalcsR = function(airports){
```

```
  temp = lapply(airports, delayStatR)
  sums = temp[[1]]
  for(i in 2:length(temp)){
    sums = sums + temp[[i]]}
  apply(sums, 2, function(x){
    mu = x[2]/x[1]
    sd = sqrt(x[3]/x[1] - mu)
  }
}
myCalcsR(airports)

###############
###IN SHELL###
###############
if(!exists("shell")){
  shell = system
}
countOriginsSh = function(origin){
  countsCmd = paste("LC_ALL=C cut -d',' -f17 [12]*.csv | egrep '(", paste(origin, sep = "", collapse = "|"), ")'| sort | uniq -c", sep = "")
  counts = shell(countsCmd, intern = TRUE)
  return(counts)
}
delayStatSh = function(origin){
  delayCmd = paste("awk -F',' '$17 == \"", paste(origin), "\" {print $0}' [12]*.csv | LC_ALL=C cut -d',' -f15", sep = "")
  stats = sapply(delayCmd, function(x){
    con = pipe(x)
    ll = readLines(con)
    mu = mean(as.numeric(ll), na.rm = TRUE)
    sd = sd(as.numeric(ll), na.rm = TRUE)
    close(con)
    list(mu = mu, sd = sd)
  })
  colnames(stats) = origin
  stats
}

countOriginsSh(airports)
delayStatSh(airports)

###############
####IN SQL#####
###############
library(RSQLite)
dr = dbDriver("SQLite")
con = dbConnect(dr, dbname = "airline")
countSQL = sqliteQuickSQL(con, "SELECT Origin, count(*) FROM subDelays GROUP BY Origin ORDER BY Origin;")
statSQL = sqliteQuickSQL(con, "SELECT Origin, AVG(ArrDelay) as mu, (AVG(ArrDelay* ArrDelay) - AVG(ArrDelay)*AVG(ArrDelay)) as var FROM subDelays WHERE ArrDelay != 'NA' GROUP BY Origin ORDER BY Origin;")
sqliteCloseResult(con)
statSQL = cbind(statSQL, sd = sqrt(statSQL[,3]))
```