Stat 243: Problem Set 6

Eugene Yedvabny

11/03/2014

Since most of the code for this problem set was run off the EC2 instance, the following code is just a copy of the commands interspersed with descriptive comments.

1

There are 21 zipped CSV files containing the 1987-2008 Airline data. The combined tarball clocks in at 1.7 Gb. The following script was used to read the CSVs one by one into R, replace NAs in DeptDelay with `1000`, and append to the SQLite db.

```r
library(RSQLite)

# Start timing
system.time({

  # Create a new database for the airline data
  air_db <- dbConnect(SQLite(),"airline_db")

  # There are 29 columns in the data set
  air_classes <- c(rep("numeric",8),"character","numeric","character",
                   rep("numeric",5),rep("character",2),rep("numeric",4),
                   "character",rep("numeric",6))

  # Read in each year and write it to the db
  years <- 1987:2008
  for (year in years){
    fname <- sprintf("%d.csv.bz2",year)

    cat(paste("Reading in ",fname,"\n"))

    temp_df <- read.csv(bzfile(fname), colClasses=air_classes, stringsAsFactors=F)

    # Set the NA delay to 1000 to ease filtering
    temp_df$DepDelay[is.na(temp_df$DepDelay)] <- 1000

    cat(paste("Writing out ",fname,"\n"))

    # Add the data frame to the table
    dbWriteTable(conn=air_db, name="Airline", value=temp_df, row.names=F, append=T)
  }

  # Close the db connection
  dbDisconnect(air_db)

})
```

The processing took ~ 20 minutes and resulted in a **8.8 Gb** database file.

2

I unfortunately could not get the Spark cluster properly instantiated, so the following is done only for ff and SQL

The following code completes 2a-d for the SQL database:

```r
library(RSQLite)
library(doParallel)
library(foreach)

# Connect to the DB
air_db <- dbConnect(SQLite(),"airline_db")

# Create a subset view
cmd1_1 <- "create view filteredAirline as select * from Airline"
cmd1_2 <- "where DepDelay > -30 and DepDelay < 720"
dbSendQuery(air_db,paste(cmd1_1,cmd1_2))

# Time the fetching of SFO & OAK information
cmd2_1 <- "select * from filteredAirline where"
cmd2_2 <- "Origin = 'SFO' or Origin = 'OAK'"
system.time(sfo_or_oak <- dbGetQuery(air_db,paste(cmd2_1,cmd2_2)))

# Time getting the mean & median delay by airport
cmd3_1 <- "select Origin, AVG(DepDelay) as avg_delay from filteredAirline"
cmd3_2 <- "group by Origin order by Origin"
system.time(avg_delay <- dbGetQuery(air_db,paste(cmd3_1,cmd3_2)))

# Create an index on the departure airports
system.time(dbSendQuery(air_db,"create index orig_idx on Airline(Origin)"))

# Rerun the previous queries to check for a speed-up
system.time(sfo_or_oak <- dbGetQuery(air_db,paste(cmd2_1,cmd2_2)))
system.time(avg_delay <- dbGetQuery(air_db,paste(cmd3_1,cmd3_2)))

# Compute the mean departure delay using parallel processing
# The EC2 instance has 4 cores, so we'll be running at max 4 threads
registerDoParallel(4)

# First let's get the unique departure cities (should be past post-indexing)
system.time(airports <- dbGetQuery(air_db,"select distinct Origin from filteredAirline"))

# Function to fetch the average delay for an airport
get_avg_delay <- function(airport){
  db <- dbConnect(SQLite(),"airline_db")
  q1 <- "select Origin, AVG(DepDelay) as avg_delay from filteredAirline"
  q2 <- sprintf("where Origin = '%s'",airport)
  result <- dbGetQuery(db,paste(q1,q2))
  dbDisconnect(db)
  return(result)
}

# Loop through the airports and fetch the avg delay
```

```
system.time(
  avg_delay_par <- foreach(airport = airports$Origin, .combine=rbind)
                    %dopar% get_avg_delay(airport)
)
```

- Full dataset: **123,534,969 entries**
- Filtering out bad delays leaves **121216293 entries**
- Serial subsetting to SFO & OAK: **50.4 sec** resulting in 3826011 rows
- Serial calculation of the mean delay: **252.2 sec** with 347 unique airports
- Adding an index on departure airport: **252.3 sec**
- Indexed subsetting: **15.2 sec** for SFO+OAK, **148.4 sec** for delays
- Parallel calculation of the mean delay: **79.9 sec**

Parallelization in this case is not spectacular- 2x the speed for 4x the computing power- so perhaps there is some bottlenecking around concurrent db access (since the averaging does happen on the SQL side).

Something interesting that emerged from playing around with filtering: SQL views do not seem to cache the indices of the subset data. Post-indexing, selecting distinct origin airports from the entire dataset takes ~13s, but selecting distinct airports from the dataset with delay > -30 and < 720 takes 130 seconds. This is regardless of whether I use the view or explicitly write out "where . . . " in the query. If storage is not a consideration, it would thus be faster to store the filtered table as its own entity rather than just a view to speed up access.

The following code repeats all of the above, this time using the ff package. For some reason the EC2 instance was horridly slow when running from the home directory (the 100Gb /) so I moved all the files into `/mnt/airline` to speed things up a bit. So perhaps the timings are not completely comparable between this and SQL.

```
library(ff)
library(ffbase)

# Load in the saved ff dataset
system.time(
  ffload("AirlineDataAll")
)

# Remove the entries with 'bad' departure delays
system.time(
  filtered_dat <- subset(dat,DepDelay > -30 & DepDelay < 720)
)

# Get the entries from SFO and OAK
system.time(
  sfo_or_oak <- subset(filtered_dat, Origin == "SFO" | Origin == "OAK")
)

# Calculate the average delay for each airport
# Use batches of 1Gb for processing, so need to account
# for multiple splits fed into the function
get_dep_mean <- function(orig_group){
  aggregate(DepDelay ~ Origin,data=orig_group,mean)
}
```

```r
system.time(
  avg_delay <- ffdfdply(x = filtered_dat, split = filtered_dat$Origin,
                        FUN = get_dep_mean, BATCHBYTES = 1000000000)
)
```

- Loading in the dataset: **100.2 sec**
- Initial subsetting to remove bad delay times: **368.2 sec**
- Extract OAK and SFO: **79.4 sec**
- Calculate the mean delay: **19.8 min**

In general the ff package runs a lot slower than the SQL queries. Not sure whether that's due to the optimizations of the SQLite db or limitation of EC disk and memory access. The `ff` package was slow even on my desktop running locally.

3

```r
library(RSQLite)

# Connect to the db
air_db <- dbConnect(SQLite(),"airline_db")

# Get the count information for each origin airport
cmd1 <- "select Origin, count(1) as DepCount from filteredAirline group by Origin"
system.time(
  orig_counts <- dbGetQuery(air_db,cmd1)
)

# Write the new table into the db
dbWriteTable(air_db,name = "DeptCounts",value = orig_counts)

# Join the tables and select observations with longest delay
cmd2_1 <- "select FA.Origin,DepDelay,DepCount from"
cmd2_2 <- "filteredAirline as FA join DeptCounts as DC"
cmd2_3 <- "on FA.Origin = DC.Origin"
cmd2_4 <- "where DepCount > 1000000 and DepDelay > 700"
system.time(
  delays <- dbGetQuery(air_db,paste(cmd2_1,cmd2_2,cmd2_3))
)

# Sort the delays and drop
delays <- delays[order(-DepDelay)[1:20],]
```

- Table creation: **614.5 seconds**. Would have parallelized this by airport had I known it would take this long. Although, system time was only at 130 sec, so it looks like there was some IO bottleneck somewhere.
- Table merge and retrieval ran for over **15 minutes** before I killed it. Playing with dbSendQuery showed that the command runs perfectly fine, but the comparators are taking a horrid amount of time. There is again something wonky with the IO- the CPU is running at only 4% and elapsed time is much larger than system or user time. Perhaps this is the same issue Chris ran in to with m3 vs c3.

```r
library(ff)
library(ffbase)

# Load in the saved ff dataset
system.time(
  ffload("AirlineDataAll")
)

# Remove the entries with 'bad' departure delays
system.time(
  filtered_dat <- subset(dat, DepDelay > -30 & DepDelay < 720)
)

# Calculate the number of departures for each airport
# Use batches of 1Gb for processing, so need to account
# for multiple splits fed into the function
get_dep_num <- function(orig_group){
  aggregate(TailNum ~ Origin,data=orig_group,length)
}
system.time(
  num_deps <- ffdfdply(x = filtered_dat, split = filtered_dat$Origin,
                       FUN = get_dep_num, BATCHBYTES = 1000000000)
)
names(num_deps)<- c("Origin","NumDeps")

# Merge the two ffdfs
system.time(
  merged_dat <- merge(filtered_dat,num_deps,by="Origin")
)

# Extract the longest delays
system.time(
 delays <- subset(merged_dat, NumDeps > 1000000 & DepDelay > 700)
)
delays <- delays[order(-DepDelay)[1:20],]
```

- I've reused the filtered dat from Q2, but included the filtering lines in the script in case it needs to be run stand-alone.
- Creation of the num_delay took the same ~**20 min** as getting the mean since the functional signature is essentially the same, just using `length` instead of `mean`
- I could not get the timing for the merge on EC2 as I ran out of mnt space to create the merged set. The copying aspect of the ff tables really adds up. The code runs fine on a smaller subset of data.

4

The datasets were exported and combined into one large CSV using the command provided by Chris in ps9:

```bash
for yr in {1987..2008}
do
  bunzip2 ${yr}.csv.bz2 -c | tail -n +2 >> AirlineDataAll.csv
done
```

The resulting unzipped CSV is, as expected, 12 Gb. Subsetting it is trivially done with AWK:

```
time \
awk -F "," '{if ($17 == "SFO" || $17 == "OAK") print $0}' \
AirlineDataAll.csv > AirlineSubset.csv
```

This operation takes **242 seconds** , so we're not actually winning much *here* by using SQL or ff.