Problem Set 6

Alexander Brandt
SID: 24092167

October 19 2015

Friendly Collaborators: Milos Atz, Alex Ojala. Most of our discussion centered around RSQLite and Spark syntax. Though I did convince some new folks to try CyberDuck.

1

The file takes almost 40 minutes to generate on an xl.large instance, and baloons from 1.7 Gb to about 18 Gb. The code to generate is detailed in the main chunk.

Listing 1: ls results

```
-rw-r--r--  1 ubuntu ubuntu  18G Nov  2 12:14 FlightDatabase.sqlite
```

2

2.1

The filtering step (detailed in the main code, below) is pretty much the same for both steps: it consists of removing NA's, and in the case of spark, getting rid of header lines from the CSV. Both are easily accomplished with a few lines to remove the offending files. In the case of R, we tag each NA as a numeric code (0.1234) before deleting from the database.

2.2

The code for Spark and PySpark is largely modified from what Chris gave us in Unit 7. Instead of computing a "median" for the time delay, we instead bin the times, then map the key/tuple to a string and write it to a file in the hadoop filesystem. By and large, the SPARK method is incredibly faster (all operations can be performed within a few minutes, vs. the arduously long loading times needed by R). For Spark, the output has been slightly modified to highlight the runtimes, without the verbosity.

For SPARK/PySPARK:

```
# Import all our necessary pacakges
import time
from operator import add
import numpy as np
from pyspark import SparkContext

sc = SparkContext()

# Read in all of our .bz2 files
```

```python
lines = sc.textFile('/data/airline')
# Repartition so our processes exceed our cores
lines = lines.repartition(96).cache()

numLines = lines.count()

# Build the key based on our values, using the same string padding
# technique I used in R
def computeKeyValue(line):
    vals = line.split(',')
    if vals[4] is not 'NA':
        vals[4] = str(vals[4]).zfill(4)[0:2]
    time_delay = vals[15]

# Return the key AND a single element list to be concatenated based
# on the key value.  The list contains the time delay.
    return("-".join([str(vals[8]), str(vals[16]),
                     str(vals[17]), str(vals[1]),str(vals[3]),
                     str(vals[4])]),[vals[15]])

# Simple method for counting all the various times that operates on the
# list passed by the key value function

def binFun(input):
    c30 = 0
    c60 = 0
    c180 = 0
    t = 0
    for i in input[1]:
        if int(i) > 180:
            c180 += 1
        if int(i) > 60:
            c60 += 1
        if int(i) > 30:
            c30 += 1
        t += 1
    return((input[0],( c30, c60, c180, t)))

# QUESTION 2A) - filter out lines missing critical information
lines = lines.filter(lambda line: line.split(',')[15] != "NA"
                     and line.split(',')[15] != "DepDelay")

# Convert the key/counts tuple to a string to write to a text file
def printable(input):
    vals = input[0].split('-')
    c30  = input[1][0]
    c60  = input[1][1]
    c180 = input[1][2]
    t    = input[1][3]

    return "%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%f" % (str(vals[0]), str(vals[1]),
    str(vals[2]), str(vals[3]), str(vals[4]),
    str(vals[5]), str(c30), str(c60), str(c180), str(t), float(c60)/float(t))
```

```
print "Query start/stop times..."
# Actually create the keys
output = lines.map(computeKeyValue).reduceByKey(add)
# Used for testing:
# print output.collect()[0:10]

# Create the bins based on keys
print "Delay binning start/stop times..."
myResults = output.map(binFun)
# Print to text file
myResults.map(printable).repartition(1).saveAsTextFile('/data/airline_processed')
```

Output:

Listing 2: SPARK timing

```
15/11/02 10:15:47 INFO spark.SparkContext: Running Spark version 1.5.1
15/11/02 10:15:48 WARN spark.SparkConf:
SPARK_WORKER_INSTANCES was detected (set to '1').
This is deprecated in Spark 1.0+.

Please instead use:
 - ./spark-submit with --num-executors to specify the number of executors
 - Or set SPARK_EXECUTOR_INSTANCES
 - spark.executor.instances to configure the number of instances in the spark config.

15/11/02 10:15:48 INFO spark.SecurityManager: Changing view acls to: root
15/11/02 10:15:48 INFO spark.SecurityManager: Changing modify acls to: root
15/11/02 10:15:48 INFO spark.SecurityManager: SecurityManager: authentication disabled
15/11/02 10:15:49 INFO slf4j.Slf4jLogger: Slf4jLogger started
15/11/02 10:15:49 INFO Remoting: Starting remoting
15/11/02 10:15:49 INFO Remoting: Remoting started; listening on addresses :[akka.tcp:/
15/11/02 10:15:49 INFO util.Utils: Successfully started service 'sparkDriver' on port
15/11/02 10:15:49 INFO spark.SparkEnv: Registering MapOutputTracker
15/11/02 10:15:49 INFO spark.SparkEnv: Registering BlockManagerMaster
[...]
15/11/02 10:26:13 INFO remote.RemoteActorRefProvider$RemotingTerminator: Remote daemon
```

And if you wanted to see a few lines from the built file (note, they are in no way ordered):

Listing 3: head of SPARK output

```
Alexanders-MBP:airline_processed Alex$ head part-00000
DL,JFK,ORD,10,6,18,2,0,0,24,0.083333
YV,CLT,GSO,1,7,00,1,1,1,1,1.000000
AA,HDN,DFW,3,7,13,2,0,0,74,0.027027
OH,JFK,DTW,5,7,19,1,1,0,9,0.111111
OO,ICT,DEN,2,5,14,1,1,0,1,1.000000
AA,DFW,ATL,9,5,12,4,3,0,90,0.044444
EV,ATL,SHV,10,2,13,0,0,0,3,0.000000
US,SFO,PHL,1,6,22,0,0,0,17,0.000000
HP,LAS,MIA,1,3,22,0,0,0,1,0.000000
AS,SEA,GEG,9,5,23,2,1,0,32,0.062500
```

For R/RSqlite:

```r
# Build up the filenames for year.csv.bz2 for our range
years <- seq(from=1987, to=2008)
years_strings <- sapply(years,toString)
fns <- sapply(years_strings, paste, sep="", ".csv.bz2")

# install.packages("RSQLite")
library("RSQLite")
# install.packages("str_pad")
library("stringr")
my_path <- "~/"
setwd(my_path)
# Create our flight database file
database_filename = "FlightDatabase.sqlite"
# Read in based on the filenames
ptm <- proc.time()
for (i in seq(length(fns)))
{
  print(fns[[i]])
  my_bz <- bzfile(fns[[i]])
  my_csv <- read.csv(my_bz,header=TRUE)

  my_csv[is.na(my_csv)] <- 0.1234
  my_csv$DepTime <- substr(str_pad(my_csv$DepTime, 4, pad="0"), 1, 2)

  drv <- dbDriver("SQLite")
  db <- dbConnect(drv, dbname = database_filename)

  dbWriteTable(conn = db, name = "flight_info",
               value = my_csv, row.names = FALSE, append = TRUE)
}
proc.time() - ptm

dbSendQuery(db, "delete from flight_info where DepDelay==0.1234")
dbSendQuery(db, "delete from flight_info where DepTime is '0.'")

# 1)

# I ended up using 'ls' but here is a way to check the file
# size using R itself
file.info(database_filename)

# Query

# 2b) Query based on the departure delays using a sum/case to
# count the number of "offending" flights
ptm <- proc.time()
x <- fetch(dbSendQuery(db, "select UniqueCarrier, Origin,
                       Dest, Month, DayOfWeek,
                       DepTime,
                       SUM(CASE WHEN DepDelay > 30 THEN 1 ELSE 0 END) as DelayedCounts,
                       Count(*) as TotalFlightCounts,
                       CAST(SUM(CASE WHEN DepDelay > 30 THEN 1.0 ELSE 0.0 END) AS FLOAT) / Count(*)
                       as DelayFraction
```

```
                              from flight_info group by
                              UniqueCarrier, Origin, Dest, Month, DayOfWeek, DepTime
                              order by DelayFraction desc"),n=-1)
y <- fetch(dbSendQuery(db, "select UniqueCarrier, Origin,
                              Dest, Month, DayOfWeek,
                              DepTime,
                              SUM(CASE WHEN DepDelay > 60 THEN 1 ELSE 0 END) as DelayedCounts,
                              Count(*) as TotalFlightCounts,
                              CAST(SUM(CASE WHEN DepDelay > 60 THEN 1.0 ELSE 0.0 END) AS FLOAT) / Count(*)
                              as DelayFraction
                              from flight_info group by
                              UniqueCarrier, Origin, Dest, Month, DayOfWeek, DepTime
                              order by DelayFraction desc"),n=-1)

z <- fetch(dbSendQuery(db, "select UniqueCarrier, Origin,
                              Dest, Month, DayOfWeek,
                              DepTime,
                              SUM(CASE WHEN DepDelay > 180 THEN 1 ELSE 0 END) as DelayedCounts,
                              Count(*) as TotalFlightCounts,
                              CAST(SUM(CASE WHEN DepDelay > 180 THEN 1.0 ELSE 0.0 END) AS FLOAT) / Count(*)
                              as DelayFraction
                              from flight_info group by
                              UniqueCarrier, Origin, Dest, Month, DayOfWeek, DepTime
                              order by DelayFraction desc"),n=-1)
proc.time() - ptm
```

Results for 30, 60, 180 minute delays (I had put ”90” where I intended to put ”30”).

Listing 4: Timing for the loading, initial (non-indexed) query

```
> proc.time() - ptm
    user   system  elapsed
3041.288   64.808 3124.991

> proc.time() - ptm
    user   system  elapsed
1846.000  141.804 2270.214
```

2.3

We perform the same calcuation, just using python instead of R. It gives the right answer ONLY if the floating point is cast first to Digit. If the code chunk is run “as is” from the problem statement, it sums to 1.

2.4

We add a index, which speeds up our calculation precipitously! Note, I ran this in the middle of the night, and I think the process got hung in an odd way, but the user and system time were much faster. When I tested on just single eyar databases on my local system, adding the key always sped up the searches, sometimes by as much as a factor of 2.

```
# Question 2D)
# Add the key based on our search values:
dbSendQuery(db, "create index delay_index on flight_info
```

```
            (UniqueCarrier, Origin, Dest, Month,
            DayOfWeek, DepTime)")
# How we might REMOVE the key, used for testing
# dbSendQuery(db, "drop index delay_index")

# Run the same searches, now much faster
ptm <- proc.time()
x <- fetch(dbSendQuery(db, "select UniqueCarrier, Origin,
                       Dest, Month, DayOfWeek,
                       DepTime,
                       SUM(CASE WHEN DepDelay > 30 THEN 1 ELSE 0 END) as DelayedCounts,
                       Count(*) as TotalFlightCounts,
                       CAST(SUM(CASE WHEN DepDelay > 30 THEN 1.0 ELSE 0.0 END) AS FLOAT) / Count(*)
                       as DelayFraction
                       from flight_info group by
                       UniqueCarrier, Origin, Dest, Month, DayOfWeek, DepTime
                       order by DelayFraction desc"),n=-1)

y <- fetch(dbSendQuery(db, "select UniqueCarrier, Origin,
                       Dest, Month, DayOfWeek,
                       DepTime,
                       SUM(CASE WHEN DepDelay > 60 THEN 1 ELSE 0 END)
                       as DelayedCounts,
                       Count(*) as TotalFlightCounts,
                       CAST(SUM(CASE WHEN DepDelay > 60 THEN 1.0 ELSE 0.0 END) AS FLOAT) / Count(*) as
                       from flight_info group by
                       UniqueCarrier, Origin, Dest, Month, DayOfWeek, DepTime
                       order by DelayFraction desc"),n=-1)

z <- fetch(dbSendQuery(db, "select UniqueCarrier, Origin,
                       Dest, Month, DayOfWeek,
                       DepTime,
                       SUM(CASE WHEN DepDelay > 180 THEN 1 ELSE 0 END) as DelayedCounts,
                       Count(*) as TotalFlightCounts,
                       CAST(SUM(CASE WHEN DepDelay > 180 THEN 1.0 ELSE 0.0 END) AS FLOAT) / Count(*)
                       as DelayFraction
                       from flight_info group by
                       UniqueCarrier, Origin, Dest, Month, DayOfWeek, DepTime
                       order by DelayFraction desc"),n=-1)
proc.time() - ptm
```

Listing 5: Timing for SQLite query

```
> proc.time() - ptm
     user    system   elapsed
  832.760   139.484 12617.140
```

2.5

Using R, we take our object generated by RSQLite and then subset based on flights with at least 150 entries. Then we view the top 10 for each of the 30, 90, 180 minute delays, respectively.

```r
# Question 2E)
#
xb <- subset(x, TotalFlightCounts > 149)
yb <- subset(y, TotalFlightCounts > 149)
zb <- subset(z, TotalFlightCounts > 149)

head(xb,n=10)
head(yb,n=10)
head(zb,n=10)
```

Results for 30, 60, 180 minute delays (I had put "90" where I intended to put "30").

Listing 6: Flight Delay (Top 10)

```
head(xb,n=10)
        UniqueCarrier Origin Dest Month DayOfWeek DepTime DelayedCounts
1945772            WN    HOU  DAL     2         5      19            61
1946374            WN    DAL  HOU     6         5      20            62
1971173            WN    DAL  HOU     2         5      21            63
1974391            WN    DAL  HOU     5         5      21            61
1985521            WN    HOU  DAL     2         5      20            58
1994481            WN    HOU  DAL    10         5      20            61
1995486            UA    LAX  SFO    12         5      18            52
1997129            WN    DAL  HOU    12         5      20            53
1997219            WN    HOU  DAL     6         5      21            56
1997436            WN    HOU  DAL     6         5      20            56
        TotalFlightCounts DelayFraction
1945772               153     0.3986928
1946374               158     0.3924051
1971173               168     0.3750000
1974391               165     0.3696970
1985521               162     0.3580247
1994481               175     0.3485714
1995486               150     0.3466667
1997129               155     0.3419355
1997219               164     0.3414634
1997436               165     0.3393939

> head(yb,n=10)
        UniqueCarrier Origin Dest Month DayOfWeek DepTime DelayedCounts
1638876            WN    HOU  DAL     6         5      18            36
1666191            WN    HOU  DAL     5         4      21            31
1666878            WN    HOU  DAL     2         5      19            26
1666978            WN    HOU  DAL    10         5      18            33
1732659            WN    HOU  DAL     5         4      19            29
1744973            WN    HOU  DAL    10         5      20            28
1745327            WN    HOU  DAL     6         4      19            28
1749951            WN    DAL  HOU     2         5      21            26
1761811            WN    DAL  HOU     4         5      21            25
1761812            UA    LAX  SFO    10         5      12            23
        TotalFlightCounts DelayFraction
1638876               189     0.1904762
1666191               180     0.1722222
1666878               153     0.1699346
1666978               195     0.1692308
```

```
1732659               174     0.1666667
1744973               175     0.1600000
1745327               177     0.1581921
1749951               168     0.1547619
1761811               163     0.1533742
1761812               150     0.1533333
> head(zb,n=10)
       UniqueCarrier Origin Dest Month DayOfWeek DepTime DelayedCounts
378918            WN    HOU  DAL     7         7      19             5
383602            WN    HOU  DAL     4         5      20             5
397917            WN    HOU  DAL     4         2      21             4
399799            WN    HOU  DAL     7         3      20             4
403164            WN    DAL  HOU     5         4      19             4
403202            WN    HOU  DAL    10         5      20             4
413930            WN    DAL  HOU     6         2      21             3
414237            AA    ORD  DFW    12         4      18             3
415160            UA    SFO  LAX    10         7      16             3
415161            UA    SFO  LAX    12         7      16             3
       TotalFlightCounts DelayFraction
378918               157    0.03184713
383602               167    0.02994012
397917               161    0.02484472
399799               166    0.02409639
403164               173    0.02312139
403202               175    0.02285714
413930               150    0.02000000
414237               153    0.01960784
415160               153    0.01960784
415161               153    0.01960784
```

3

```r
# Question 3)
# install.packages("parallel")
library("parallel")

# Build a function, getDelays, which takes a time value
# and a "string" which represents a letter of the alphabet
# this is used in a regex way to break down the search by
# flight code starting letter (handy way to ensure no doubled
# results)
getDelays <- function(x,s) {
  # print(x)
  s <- toString(s)
  # print(s)
  drv <- dbDriver("SQLite")
  db <- dbConnect(drv, dbname = "Big_v4.sqlite")
  query <- sprintf("select UniqueCarrier, Origin,
                       Dest, Month, DayOfWeek,
                       DepTime,
                       SUM(CASE WHEN DepDelay > %i THEN 1 ELSE 0 END) as DelayedCounts,
                       Count(*) as TotalFlightCounts,
```

```
                        CAST(SUM(CASE WHEN DepDelay > %i THEN 1.0 ELSE 0.0 END) AS FLOAT) / Count(*)
                        as DelayFraction
                        from flight_info where Origin like '%s%%'  group by
                        UniqueCarrier, Origin, Dest, Month, DayOfWeek, DepTime
                        order by DelayFraction desc", x, x, s)
  tmp <- fetch(dbSendQuery(db, query),n=-1)
  return(tmp)
}

alphabet = c("A","B","C","D","E","F","G","H","I","J","K","L",
             "M","N","O","P","Q","R","S","T","U","V","W","X",
             "Y","Z")
times <- c(30, 60, 180)
# All combinations of alphabet letters and delay times
tlc <- expand.grid(times,alphabet)
names(tlc) <- c("x","s")

ptm <- proc.time()
question_3 <- mcmapply(getDelays, tlc$x, tlc$s, mc.cores=4)
proc.time() - ptm
```

Listing 7: Timing for all_preprocess.sh

```
> proc.time() - ptm
   user  system elapsed
245.192  22.508 375.559
```

4

The preprocessing I used here was basically derived from my solution in ps2, just generalized to work with more than one bzip2 file. The code and a wrapper script is shown below below. The file almost takes about 10 minutes, which is m3.xlarge instance. It is probably worth it, for the case of R, because reading in the files can take a very long time.

Listing 8: all_preprocess.sh

```
myyear=$1
# Extract the header so we can find our columns of interest
bzcat $myyear.csv.bz2 | head -n 1 > $myyear.header.txt
# We will use the file line coordinates as the proxy for index columns
sed -e $'s/,/\\\n/g' $myyear.header.txt > $myyear.header.nsv
# Our desired headers
for i in "UniqueCarrier" "Origin" "Dest" "Month" "DayOfWeek" "DepTime" "DepDelay"
do
    x=`grep -n ^$i$ $myyear.header.nsv | cut -d':' -f 1`
    v="$v $x"
done
echo $v

# Now $v contains our columns of interest, which we just need
# to separate by commas to use with cut. A sed command will
# accomplish this with ease.
```

```
bzcat $myyear.csv.bz2 | \
     cut -d, -f`echo $v | \
     sed 's/ /,/g'` | bzip2 > $myyear.pp.csv.bz2
```

This script is called with:

Listing 9: preprocess.sh

```
date
for f in `seq 1987 1 2008`
do
    echo $f
    ~/preprocess.sh $f
done
date
```

And the result:

Listing 10: Timing for all_preprocess.sh

```
Mon Nov  2 08:25:32 UTC 2015
Mon Nov  2 08:35:15 UTC 2015
```