Stat243: Problem Set 6, Due Monday Nov. 3

October 22, 2014

Comments:

- This covers Units 8 and 9.
- It's due at the start of class on Nov. 3.
- As usual, simply providing the raw code is not enough; make sure to describe how you approached the problem, the steps you took, and output illustrating what your code produces.
- Please note my comments in the syllabus about when to ask for help and about working together.
- As discussed in the syllabus, please turn in (1) a copy on paper, as this makes it easier for us to handle AND (2) an electronic copy through Git following Jarrod's instructions.
- Please see *howtos/startEC2virtualMachine.txt* and *howtos/startSparkVirtualCluster.txt* for instructions on starting up a single Amazon EC2 instance (virtual machine) and an Amazon EC2 Spark virtual cluster, respectively. We've tested these out, but I wouldn't be surprised if there are issues, so don't spin your wheels much before checking with us (in office hours or via Piazza) about problems.
- The airline dataset is available at *http://www.stat.berkeley.edu/share/paciorek/* as a set of .bz2 files (*1987.csv.bz2*, ... , *2008.csv.bz2*), a tarball containing all of the .bz2 files that may be more convenient to download (*1987-2008.csvs.tgz*), and the .ffData (*AirlineDataAll.ffData*) and .RData (*AirlineDataAll.RData*) files that are produced by *ffsave()*. You'll need the former for Spark.
- **Don't delay in getting started on this. A lot of what you'll try out here is likely new to many of you, and things are more complicated logistically to do in the cloud than just doing something on your local machine. We're happy to help as you run into snags, but not, for example, on the Sunday before it's due.**

Problems

1. Using an Amazon EC2 instance with at least 4 cores (you'll use the 4 cores in problem 2), create an SQLite database from R with a single table to hold the airline data. To do this you should not read the entire dataset into memory in R, even if you have a machine with sufficient memory. Rather, use the individual year files and build up your table in pieces. Note that because RSQLite appears to convert NAs to zeros in numeric fields, you should replace the missing values in the fields used in Problem 2 with a numeric code.
   How big is the database file, compared to the original CSV (12 Gb), a bzipped copy of the original CSV (1.7 Gb), and to the .ffData binary representation?

   Note: you should use the *colClasses* argument to control the type for the fields in the SQL table(s).

2. First some setup.

   - Create a Spark RDD containing the airline data. Use 12 slave nodes in your virtual cluster. Load the individual .bz2 files onto the HDFS and read the data into a Spark RDD. Repartition so the dataset is equitably spread across the slave nodes in your Spark cluster.
   - In the same Amazon EC2 instance as used in Problem 1, read in the dataset into R using the *ff* package. You can use the .ffData and .RData files provided to speed your implementation.

   (a) Subset/filter your datasets in your Spark, SQLite and ff data representations to omit flights with departure delays of less than (-30) minutes and greater than 720 minutes, as well as flights with missing values for the departure delay. In the remaining questions, use this filtered dataset.

   (b) Now do some exploration where you compare the speed of extracting subsets of the data using Spark, SQLite and ff, in particular:

      i. extracting the flights departing from SFO (San Francisco International Airport) or OAK (Oakland International Airport), and

      ii. finding the mean (and median when possible) departure delay by airport for all of the airports in the full dataset.

   (c) Add an index for departure airport to your SQLite database. How does having an index in your SQLite database affect speed?

   (d) Use *foreach*, *mclapply()*, or one of the parallel apply functions in R on your EC2 instance. How does the speed of calculating the mean departure delay by airport in parallel using the SQLite database compare to the other approaches used above? Note, you'll need to open separate database connections for the separate tasks for this to operate in parallel.

   (e) Extra credit: Look at the information in the section on "RDD Persistence" in the Spark Programming manual. Experiment with using the *cache()* method when carrying out multiple operations in separate lines of code. Does it seem to make a difference? How about forcing the RDD to be stored on disk rather than in memory? Finally, does changing the number of partitions affect timing?

   Note: when you use Spark, it uses lazy evaluation, so sometimes you may have to do an operation like `take(1)` to ensure that the computation actually gets done. So the timing numbers may be a bit fuzzy.

3. Now let's consider joins/merges. Compare the speed of the following operations using SQL and ff.

   (a) Create a second table in your database that contains the number of departing flights. You should do this calculation using the COUNT and GROUP BY syntax of SQL, about which there should be lots of information online and in the Murrell book. Merge the original and new table and extract the 20 observations with the longest departure delays only from airports with at least 1 million flights

   (b) Find the number of departing flights from each airport using ff functions and create a second ff dataframe with this information. Now merge the two dataframes. Again extract the 20 observations with the longest departure delays only from airports with at least 1 million flights.

   (c) Extra credit: Do the same in Spark. You'll likely need to first collect the counts on the master and then broadcast them out to the slaves to be added to the RDD in a map step.

Hint: do not try to sort any large tables. Select enough large values that you can sort on a smaller dataset held as a simple R object.

4. How about using UNIX tools from Unit 1 to extract the SFO/OAK flights from unzipped file(s). How fast is that compared to the other approaches above, without considering the time to unzip the files? You should be able to do this with a simple *grep* that uses regular expressions.