Christine Cai
STA250 HW2
Winter 2014

Method 1

This method uses read.csv() to read in entire csv files on gauss (ignoring all columns but arrival delays through the colClasses argument). Gauss is a cluster consisting of 12 nodes, with 32 cores (64MB of memory) on each node. It is running Ubuntu 12.04.

81 array jobs are set up, one for each csv file. Each job takes in the array job number, say j, and reads the j-th file in the "data" directory. A frequency table for arrival delays is then written to a txt file in the "output" directory. The individual frequency tables are concatenated when all the array jobs are done. Any duplicated arrival delay counts are merged (in R) and the desired statistics are calculated using the Hmisc package.

To execute this method, simply run the commands in gauss/commands.txt (in my git repo) sequentially. There are only two things the user may need/want to change, and they are direc and direc.out specified at the top of gauss.R. direc is the directory containing (only) the 81 csv files, and direc.out is the directory where the frequency tables will go. Note that since the csv files are in the directory data, the user should input 'data/'; the slash at the end is important.

I ran my code once when the server was busy and once when the server was not busy. When it wasn't busy, it took 2.5 minutes to make all the frequency tables. All 81 jobs were able to run simultaneously on gauss. It then takes another few seconds to concatenate the tables and get the desired statistics. This is a speed up of about 12x, compared to running this code serially.

While Gauss can be powerful, the limit of its power is quickly observed. When the server is busy, new jobs may need to wait a while before they are started. This can occur when too many users submit too many jobs or it may even be one user with 5000 jobs. The time your jobs spend in pending purgatory may not be negligible.

Method 2

This method implements MapReduce with Python. I tried using Amazon's Elastic MapReduce at first. I ran into so many problems that I've concluded that Amazon makes

their web services hard to use on purpose so the user would buy into the higher support tiers. Also, as far as I know, Amazon's Hadoop doesn't handle .tar.bz2 files. So I would have to download the .tar.bz2 file on my laptop, uncompress it, upload the uncompressed files to s3, and copy the uncompressed files to hdfs. This may take forever. Although, I did a bit of reading on this subject. I believe if I compress each csv file to a .bz2 file, upload them to s3, then copy them to hdfs, Hadoop will automatically unzip each file when doing the computations. I didn't attempt this as I was very frustrated with Amazon by this point. Also, I didn't want to download the files to my laptop. I have the files on a remote server, and I couldn't find a way to upload the files to s3 from a remote server.

I ended up using the new statistics department server hadoop.ucdavis.edu; this went very smoothly compared to using AWS. This server is running Hadoop 1.2.1. The mapper loops over and processes every line of every file, and outputs <key, value> pairs. Since our goal is a frequency table, each key is an arrival delay and every value is 1.

The reducer sums up the counts for the same keys. First, the current key is initialized to 'None', this is done so that python can recognize that the first key it reads in is a new key and it should start counting (the number of occurrences for this key) from 1. If the new key is the same as the current key, simply increment the current count by 1 and move on to the next <key, value> pair. If the new key is different from the current key, and we're not processing the first <key, value> pair in a file, then print the result. Notice that with this scheme, the last result will not be printed in the for-loop, so we have to add an extra print at the end (outside of the for-loop).

The code can be tested on a non-Hadoop machine with

```
cat data/1987.csv data/2008_May.csv | python mapper.py | sort -k1,1 | python reducer.py > test.txt
```

The data must be copied to hdfs before the user finds the path to the streaming-jar file and runs the program. Note that the output folder cannot be preexisting on hdfs (Hadoop wants to create that for you). Once the job finishes, the output (frequency table) must be retrieved from hdfs to the local machine, and the statistics are calculated in R with the Hmisc package.

To execute this method, simply run the commands in mapReduce/commands.txt (in my git repo) sequentially. For these commands, please change directory names as

needed. This job took 13.5 minutes; it took 6.5 minutes for the mapper to finish (the reducer started before the mapper was finished).

Method 2.5

This method implements MapReduce with Java. I used the Java code given to the class with some additions to the mapper. The original mapper would only work for the pre-2008 files, and now it works for the post-2008 files as well. Basically, after splitting a line by commas, if there are less than 45 elements, the 15^{th} element is dubbed as the key. If there are more than 45 elements, the 45^{th} element is dubbed as the key. Note that this means array elements 14 and 44 for languages with array index 0. The arrival delay entries in the post-2008 files have decimals in them, which Java (and python) doesn't like. So they must be parsed accordingly. This is also how the mapper works in Method 2. The program can be ran after compiling the .java files and creating a jar file. When the program is done, the output (frequency table) must be retrieved from hdfs to the local machine, and the statistics are calculated in R with the Hmisc package.

I couldn't get the GNUmakefile provided to work. So to execute this method, please run the commands in mapRedJava/commands.txt (in my git repo) sequentially. For these commands, please change directory names as needed. This job took 7 minutes; it took 6 minutes for the mapper to finish (the reducer started before the mapper was finished). So this Java code is about twice as fast as my python code. Both mappers took about six minutes to finish, but the Java reducer is much faster than the python reducer. This makes sense since the Java reducer works on the bucketed outputs from the mapper, where each key has its own bucket, while the python reducer simply loops over each line of output from the mapper.

Conclusions

My results from all three methods are the same as those in assignment 1; mean = 6.57, median = 0.00, and sd = 31.56.

After working with this data some more, I've realized that its structure is actually very simple and clean (even though the pre-2008 files are different than the post-2008 files). While coding, there were several times when I was able to find easy fixes only because of the consistency of the data.

Please refer to the plot below for computation times of the three methods and the non-parallel read.csv() method from assignment 1. Note that this plot does not take into

the account the few minutes it may take to copy the data to hdfs. Gauss seems to be the winner here. It is not only extremely simple to implement, but it is also the fastest. The data probably has to be much bigger for Hadoop to outperform gauss. For this problem, (other than for educational purposes), it isn't worthwhile to implement MapReduce.