Big Data and Analytics Programming: Assignment 4

Toon Nolten
toon.nolten@student.kuleuven.be

June 6, 2016

1 Note

I couldn't run on the clusters because of connection refused errors.

2 Exercise 1: Trip Length Distribution

I chose java for the first part so the program is included in *Exercise1.jar*. The approach is very simple: I read all trips one by one parsing enough information to calculate the distance, then I write all the distances to a file one by one. Incorrect trips were identified as follows: if the number of whitespace separated columns is not correct, the trip is discarded, then if any exception occurs during the parsing of those columns, the trip is also discarded. This approach is relatively foolproof but also very strict about the format of the input data. The trip distance distribution is then plotted by Gnuplot, see figure 1.

Because the problem is so simple my MapReduce solution only uses a mapper. This mapper parses the trips and calculates the distances one by one much like the simple solution. The output from the many mappers is then collected in one file with a simple Hadoop command. The identification of incorrect data is identical to the simple method. Again the trip distance distribution is plotted by Gnuplot, see figure 2. The distribution is the same but multiple bars are plotted on top of one another, I'm not sure why this is. Something in the output of the mappers must be confusing Gnuplot.

Most trips are very short, less than 5km, which seems reasonable as most people commute between different places in or around the city center. The second peak might be from trips to the airport. One thing to note is that there were many trips of longer distances, including ones of more than 10000km, these are probably erroneous and should be removed from the data. In this case the Hadoop implementation does not result in better run times because the problem is so simple and the framework incurs relatively high overhead costs. However if the data are bigger or calculation has to be done on a cluster the simple method is not an option.

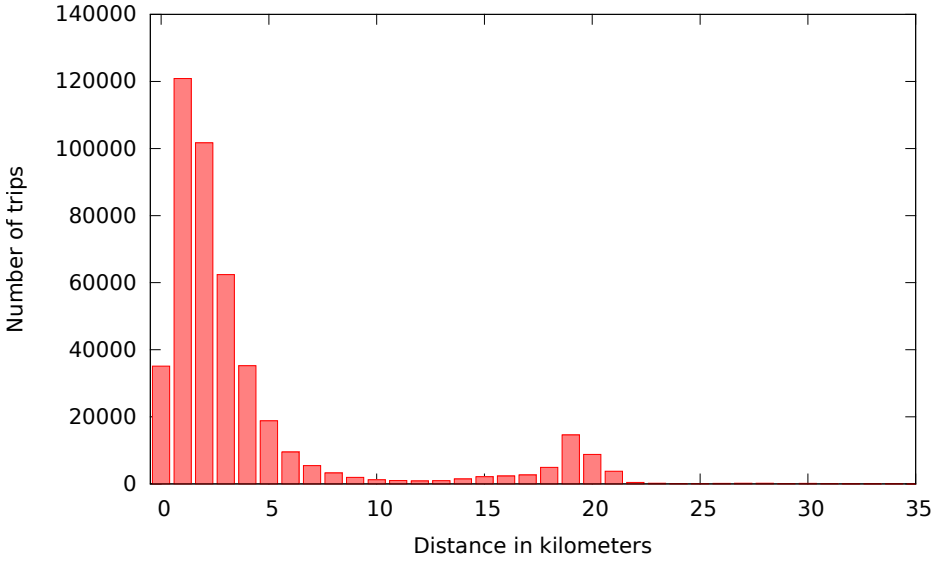

Figure 1: Trip Length Distribution: simple method

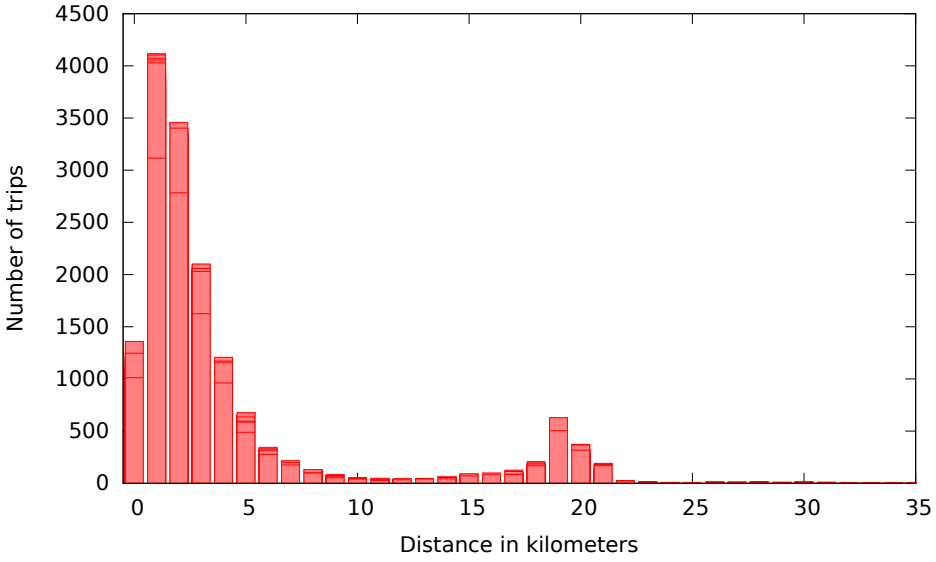

Figure 2: Trip Length Distribution: MapReduce

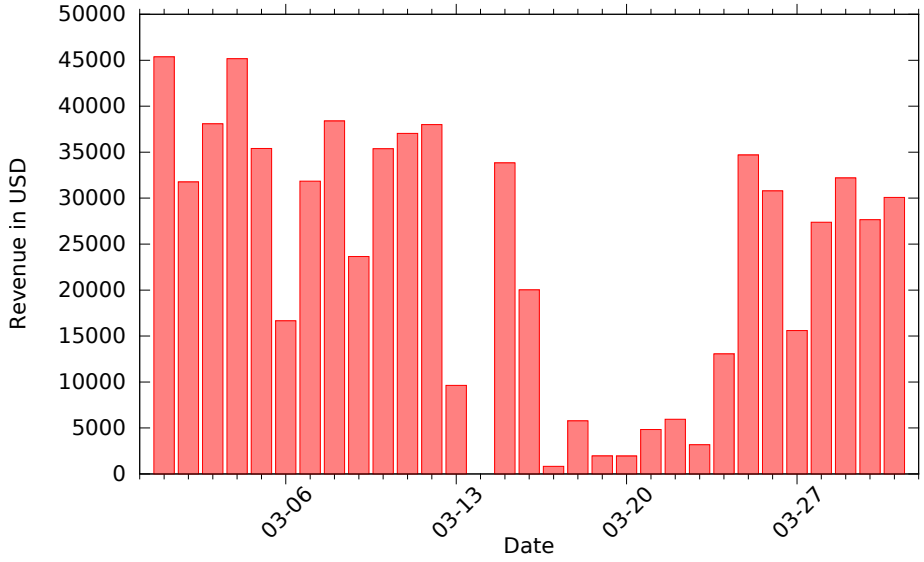

Figure 3: Airport Ride Revenue

3 Exercise 2: Computing Airport Ride Revenue

This problem was solved as suggested, first constructing trips then computing revenue. I used two MapReduce jobs to accomplish this. The first job constructs the trips as follows. The mappers read in the segments using the taxi-ID as the key for the reducers. The reducers are more complicated: they split there input segments into empty and full segments because those can never be part of the same trip, then for each of those sets they take the first segment and try to find a match among the remaining segments, if no match exists the segment represents a complete trip, if a match exists the segment is merged into it's match. This somewhat complicated algorithm is necessary because each segment has to be compared to every other segment. A simpler algorithm which doesn't achieve this even though it seems like it does, goes like this: use two collections, one contains the segments, the other will contain the trips, take a segment compare it to all the partial trips if it matches one, merge it in, otherwise add it to the partial trips. To show this wouldn't work here's an example: imagine a trip consisting of segments a, b and c in that order, imagine these come from the mappers in another order where the intermediate segment b is last, the simpler algorithm would put the first segment in the trips, let's say a, then compare the next segment, c, with a, which doesn't match so it's added to the trips, then b will match one of them but the resulting trips are not complete.

The second job calculates the revenue. The mappers reject each trip that doesn't start or finish within 1km of the airport or doesn't have the full status, empty taxi's don't earn money. The reducers then take each of the trips, calculate the cost and output the date and the cost of the trip. The revenue over time for airport rides is then plotted with Gnuplot, see figure 3.

I calculated the total revenue with an awk command because a third MapReduce job seemed overkill. The total revenue, for the *2010_03.segments* dataset,

added up to $716339.

Since I couldn't run the jobs on the cluster I couldn't tune the number of mappers and reducers. In general somewhere between 10 and 100 mappers per node seems appropriate and a number of reducers so the overhead is minimized while maximizing parallelism. As a result the inputsplit in my code is still set to a very low number which is not appropriate for any serious dataset. Since Hadoop takes the number of mappers and reducers as guidelines the best way to set the number of mappers seems to be to calculate an appropriate inputsplit and produce an appropriate amount of keys from the mappers. The ideal number of mappers for scalability should be relatively close to 10-100 times the number of nodes. For the reducers the ideal number is very difficult to predict and the best way to estimate it seems to have to do with the amount of overhead for a reduce task, the duration of such a task and the number of nodes in the cluster.