Design

Maple

For our Maple implementation, we had a client-follower architecture to track the maple task. The client first would read the src file from sdfs. Then, the client would use hash partitioning to assign various followers key tasks (followers taken from the gossip membership list). The client then sends tasks to the followers to trigger the followers' getting of the key files, and running of the maple executable. If a follower prematurely terminates a connection, a new one is picked. Once all followers have saved the outputs to their local filesystem, they send acks to the sdfs leader indicating the final output has been created, and now there are several intermediate_prefix files with keys in sdfs for juice to use.

Juice

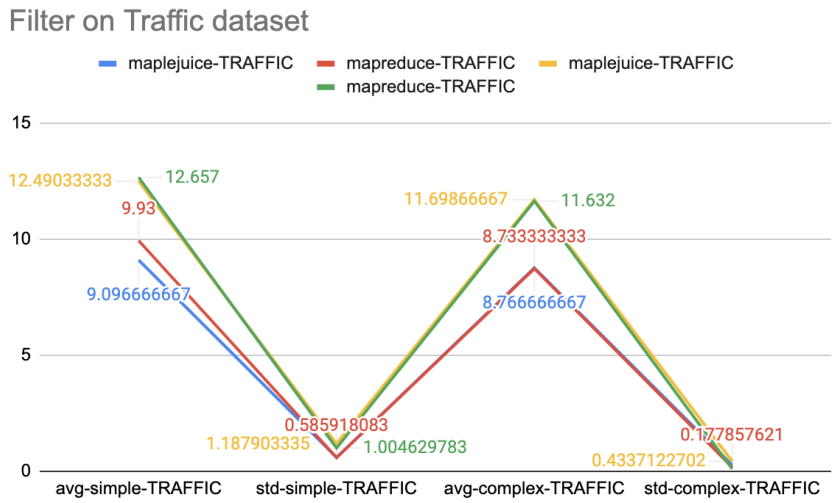
For our Juice implementation, we had a client-follower architecture to track the juice task. The client first would fetch all intermediate files specified, via the command line args. Then, the client would use hash/range partitioning to assign various followers key tasks (followers taken from the gossip membership list). The client then sends tasks to the followers to trigger the followers' getting of the key files, and running of the juice executable. If a follower prematurely terminates a connection, a new one is picked. Once all followers have saved the outputs to their local filesystem, they send acks to the sdfs leader indicating the final output has been created, and the user can use a regular get to retrieve the file.

SQL Commands

For the filter SQL command, we developed a Maple Juice program that would accept a csv. Our Maple executable would accept an input file and a regex representing the command to filter by. The output would be a set of key value pairs with the key being the matched regex and the value being the rest of the values of the row. From here, our Juice executable would take these key value pairs, aggregate them, and output them as one row.

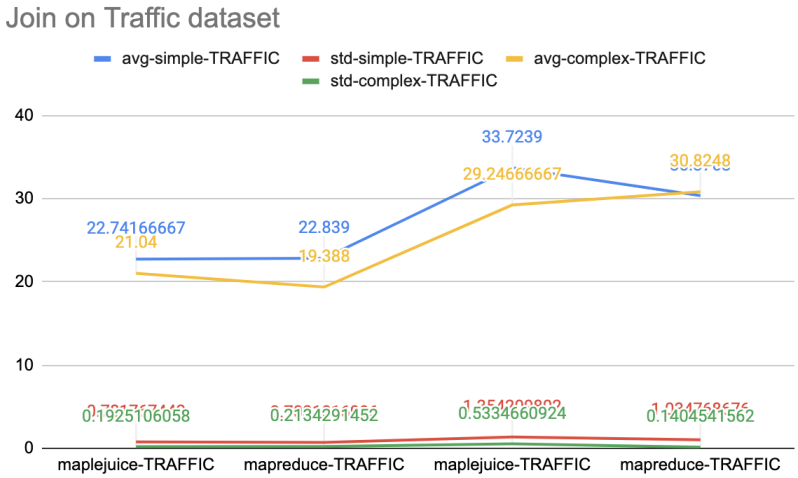
For our join SQL command, we developed a set of 3 Maple Juice programs that would accept two datasets and output all the rows on which there was a match in user inputted column values. Our initial pair of Maple Juice programs would each accept a dataset, setting the key of the output to be the values read on the user inputted column. Both pairs of MapleJuice programs would output to a sdfs file with the same prefix. From here, our final Maple would take the sdfs outputs and directly output it. Our final Juice would aggregate all keys with the same values into a 2D array. If there were two lists in our 2d array, then we knew that in the user inputted columns for D1 and D2, they both had elements/keys that were equal. Thus we could join their rows using their shared key. The output of the final MJ program was the joined values of D1 and D2 on the user inputted columns.

GRAPHS #1 - 5 VMs



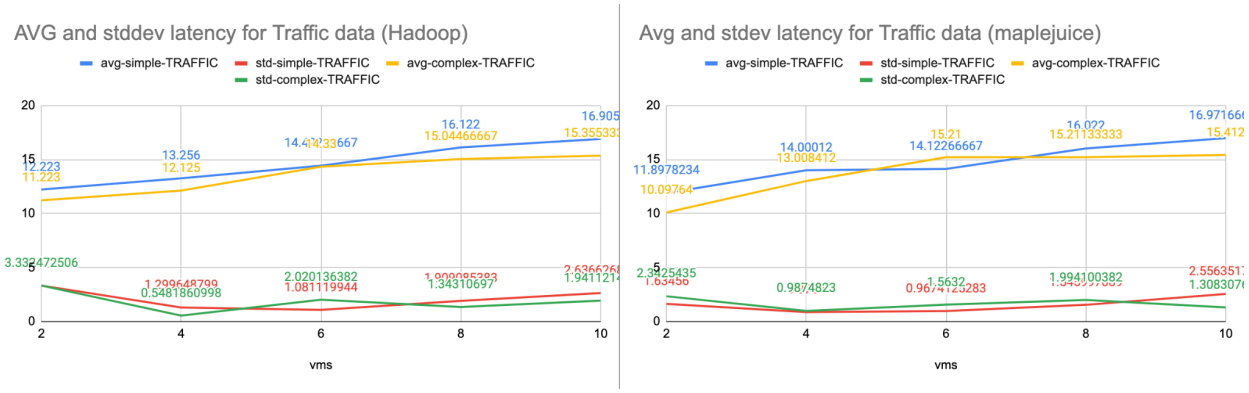
For the filter graphs, we used the champaign traffic dataset, as well as the facebook social network graph dataset from the Traffic dataset. The graphs show the hadoop vs sql filter speed, with average and stdevs. It seems as though with the current parameters, maplejuice performed slightly better. On a smaller scale, this makes sense, as we do not depend on sdfs for intermediate files in the same way that hdfs does, and instead send acks and simulate puts for individual files. **Filter 1 corresponds to the 'simple' datapoints, and Filter 2 corresponds to the 'complex' data points.**

GRAPHS #2 - 5 VMs

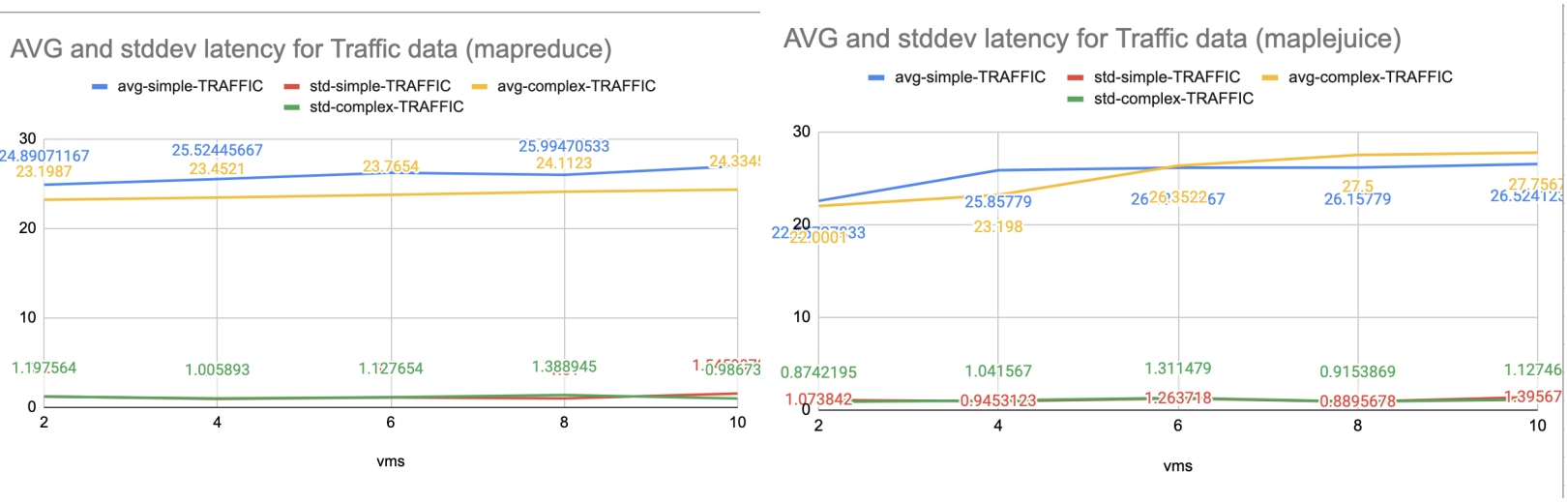


The above graph displays the results of a complex and simple join on the 2 previously used datasets. In this case, our implementation was unable to beat hadoop in the case of a complex query. **In this case, Join 1 = simple and Join 2 = complex.** One reasoning for the loss at the simple stage might be the introduction of more matches. Our simple regex had significantly more matches, and hadoop inheritly is more parallelized that our implementation, so for larger matchcounts our implementation might suffer.

GRAPH #3 - Varied VM



The above represents the varied vm size, with a constant amount of map and reduce tasks. As the number of vms increases, we hypothesize that the increase of network traffic but the static nature of the number of map and reduce tasks causes an increase in latency. While the standard deviations remain mostly the same, the averages increase across more vms. After a certain point, the threshold of average size would decrease, and as the number of map/reduce tasks proportionally increases with the number of vms, the latency would mostly likely eventually taper off.



GRAPH #4 - Varied VM for Join

In this case, we generally had very similar data across different vm sizes. It seems like the maplejuice implementation did better with smaller vm sizes, but with larger vm sizes, the mapreduce implementation performed better. Additionally, the variance for this round was extremely low, potentially because of the larger corpus of data being processed by the separate stages. Again, we see the latency increasing over vm size, due to a similar reason outlined in the previous portion.