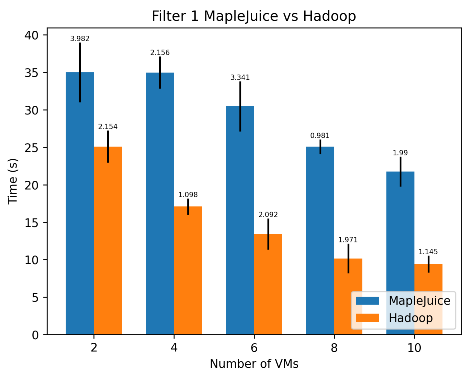
Samaah Khan (skhan330) & Amaan Khan (amaanmk2)

Design (architecture & programming framework) for MapleJuice - 0.75 pages

We have a leader node which handles scheduling of the different maple and juice jobs. A client can submit a maple/juice job to the leader, and the leader then adds it to a queue. When ready, it will pop a job from the queue, and execute it (we only allow 1 job to run at a time). For maple jobs, the leader sends a maple task request to *num_maples* workers with an assigned *task index.* Each of those workers will execute the task on its respective portion of the data that the leader told it. It gets the data by performing a GET operation from SDFS. It knows its respective portion of the data by splitting every file in the source directory into *num_maples* and choosing the chunk that is of the index *task index*, which was sent by the leader to the worker node. The worker nodes run the maple executable through our Go program. I pass in a few command line args into the maple executable, like the input file, starting line number, and number of lines to read. Additionally, for SQL jobs I also pass in the WHERE condition as an argument. This executable is ran on 60 lines at a time, which is done sequentially for its respective portion of the data. The executable then prints its output to STDOUT, from which the MapleJuice program will pipe the stdout data, read it, and then write it to a local file. After the worker is done running all its assigned tasks, it then sends back a MAPLE TASK RESPONSE to the leader with an output file containing all the key value pairs. The leader then reads this file line by line, and writes each key value pair to its respective separate file - one file per key. Once it gets all the maple task responses back, the maple job is done, so then it does an SDFS PUT operation on all those local files that it created which has 1 key per file, prepended with teh sdfs_intermediate_filename that the user typed in. Then it sends back an ACK to the client indicating its done. Juice jobs are very similar to this, except that the input is the intermediate filenames. The partitioning I used was *hash partitioning*. The leader decides which juice task is assigned to which key(s) based on hash partitioning. It tries to allocate 1 task per worker node, but if the number of tasks is greater than the number of worker nodes, then some nodes get more than 1 task (this is the same for maple). The worker nodes then run the juice executable on each file (in parallel by spawning new goroutines). The values from that are appended to one local file, which is sent back to the leader. Once the leader gets back all the juice output files, it adds them together into one file and then does a SDFS PUT to the destination file.

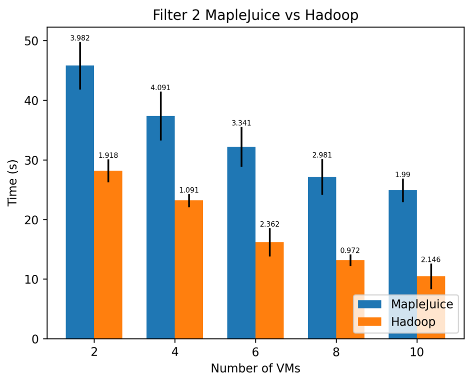
For failures, whenever the leader detects a node failed, it removes it from its list of available worker nodes. It then checks if this node had any job currently running, and if so, which tasks were assigned to it (this information is held in a struct + map). It then finds a new node that is active and sends that same information over to the new worker node, so that that worker node now executes the task instead.

Filter 1



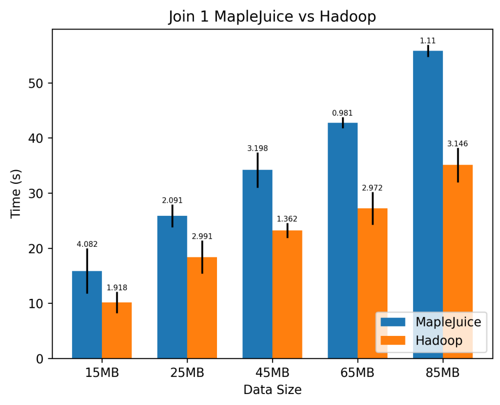
As the number of virtual machines increased, the time it took for our simple filter operation decreased. This is because by using more virtual machines our MapleJuice is more parallelized and the workload is being distributed among multiple machines, so this is expected. Hadoop was also significantly faster than our MapleJuice implementation. This was expected because Hadoop is much more than our MapleJuice implementation. For example, Hadoop is much more optimized with caching. In addition, we did not as highly parallelize the worker nodes like Hadoop does. Therefore our results were expected.

Filter 2



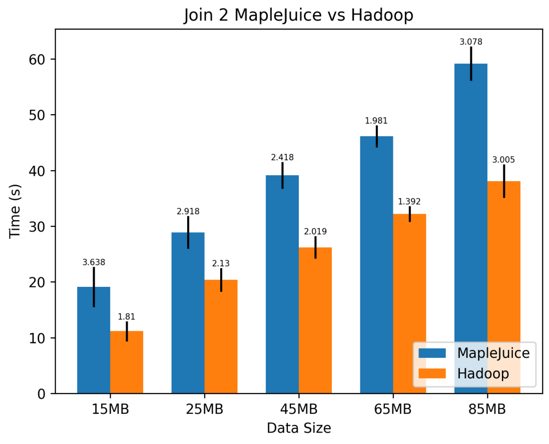
As the number of virtual machines increased, the time it took for our simple filter operation decreased. This is because by using more virtual machines our MapleJuice is more parallelized and the workload is being distributed among multiple machines. Hadoop was also significantly faster than our MapleJuice implementation. Again, this was expected because Hadoop is much more optimized than our MapleJuice implementation. Therefore our results were expected. This was a similar result to our first simple filter query. The main difference between the first and second plot was that all the times took longer because it takes a longer time to perform a complex sql query.

Join 1



As the dataset size increased, the time it took to perform the join operation increased for both Hadoop and for MapleJuice. This is expected because in order to perform queries on larger datasets it takes a longer time. In this scenario, Hadoop also beat our MapleJuice implementation. As stated before, this is because Hadoop is much more optimized including in terms of more parallelized caching, etc. Therefore, the results from this plot were expected.

Join 2



As the dataset size increased, the time it took to perform the join operation increased for both Hadoop and for MapleJuice. This is expected because in order to perform queries on larger datasets it takes a longer time. In this scenario, Hadoop also beat our MapleJuice implementation. As stated before, this is because Hadoop is much more optimized including in terms of more parallelized caching, etc. Therefore, the results from this plot were expected. This is similar to our result for Join 1. However it took a longer time to perform the more complex query.