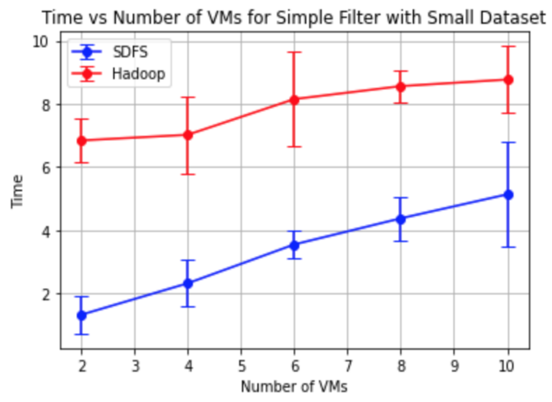
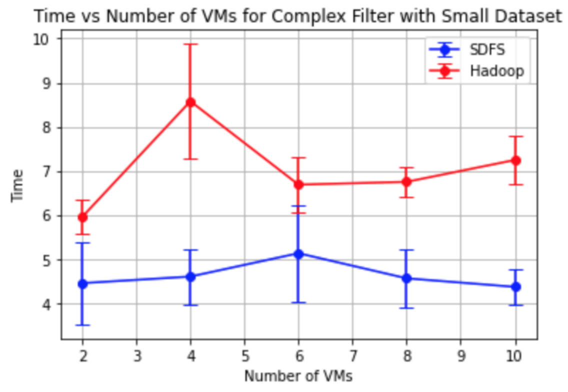
Sayan Sisodiya (sayans3)
Vraj Patel (vrajsp2)

Design: Our implementation of MapleJuice has a single leader and multiple worker nodes (the leader is also a worker node). The Maple and Juice commands require that the input files, as well as the relevant executables, are already stored in the SDFS. Maple commands are sent from any worker node to the leader. The leader then gets all of the files in the source directory locally, and chunks them into num_maples different files. The leader puts these files back into the SDFS, and then allocates the file chunks to all of the online machines (including itself) in round-robin order until all have been allocated. These allocated files are sent along with the other information, like the intermediate filename prefix, in MapleTasks. MapleTasks execute in parallel on worker nodes, so if a worker receives more than one MapleTask they can still happen in parallel. MapleTask gets all the files needed (the file to process and the executable) from the SDFS and then runs the executable on the file. The resulting output file is then put up to the SDFS. Once the leader sees that all the MapleTasks have finished, it deletes all the temporary files created on it, and all the ones it can delete on workers. This is where Juice starts; when a Juice command is sent from a client to the leader, the leader collects all files with the intermediate filename prefix, and partitions them into num_juices groups based on the partitioning scheme requested by the client (hash or range-based). Then it allocates these groups to all of the online workers, and sends them and other information (like the destination filename) along to the workers in JuiceTasks. JuiceTasks also execute in parallel on the workers. JuiceTask gets all the files it needs and runs the executable. The local resulting file is then put (appended/merged) to the destination filename in the SDFS. If either Maple or Juice tasks fail, the leader retries on the next online node. Only once all tasks have been completed successfully does it return to the client.

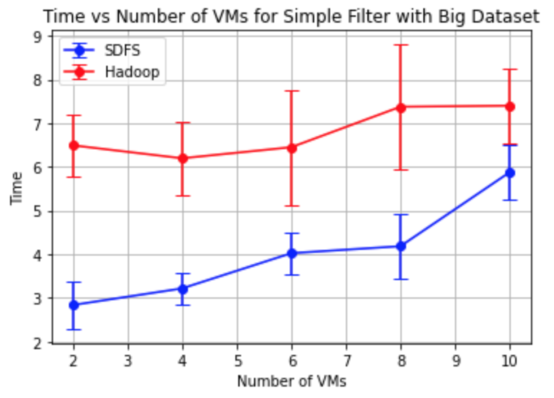
We wrote the SQL layer using only our Maple and Juice components. The SQL filter command uses a single Maple job only; we wrote a custom executable that runs grep locally on the machine and stores the results in a file. The grep results from all of the spawned MapleTasks (we fixed this number at 10 for filter and join) are appended/merged to create the final filter output. The SQL join command uses two Maple jobs and one Juice job, and it takes in two datasets and two column indices (the ones to join on) as input. Each of the Maple jobs reads from one of the input datasets. Each Maple job (split into 10 MapleTasks) isolates values from the requested column in the data, and it puts "<dataset #> <entire current row>" to the intermediate file that is named after the column value for the current line. Both of the two Maple jobs do this. This way, all of the intermediate files generated by the Maple phase contain only rows from both datasets that had matching column values (so they are joinable). The single Juice job (split into 10 JuiceTasks) then goes through each file and performs an inner join on rows from dataset 1 and dataset 2, appending/merging the output into the final join output.

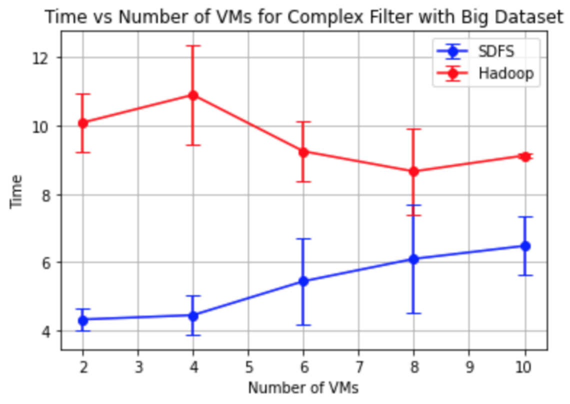
Sayan Sisodiya (sayans3)
Vraj Patel (vrajsp2)





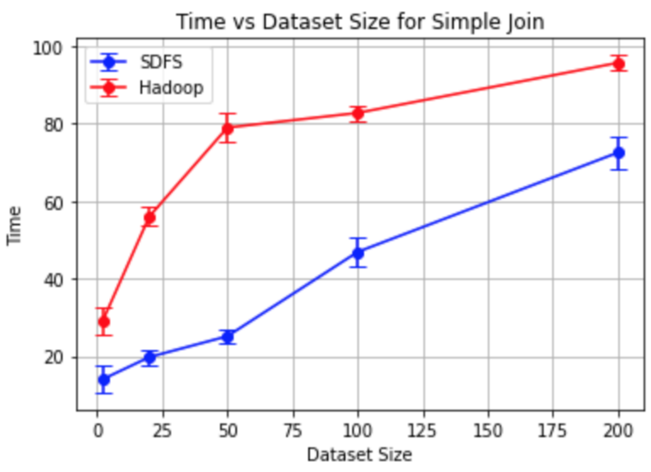
The plots above show filters on a small (20MB) dataset. I suspect that our time increased as the number of VMs increased because the files and executables used by the Map and Juice jobs were less likely to be on the machine assigned the relevant task as the number of VMs increased, so there would be more network calls to get those to the assigned machine. Our implementation handily beat Hadoop, which is likely due to the fact that our implementation had some optimizations that allowed machines to avoid repeatedly calling our MP3 “get” if the executable was already present locally. However, for the simple filter, it seems like our implementation does not scale as well as Hadoop; the slope of our SDFS line is larger than the Hadoop line as the number of VMs increases. For the complex filter, the overall times were consistent, at about the level of the slowest simple filter. This could be because the execution time of the filter executable starts to dominate, and the number of VMs becomes less impactful.

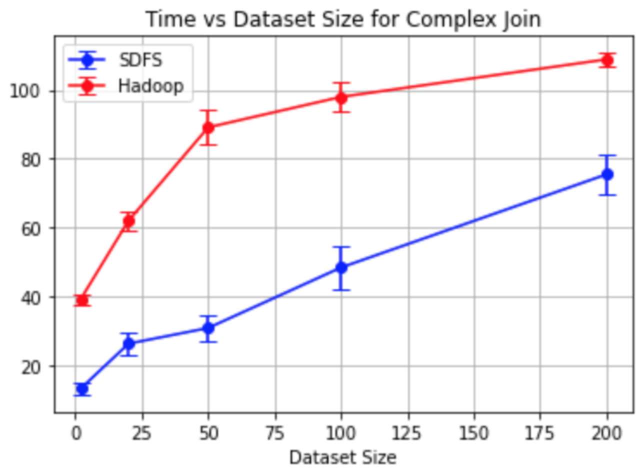




The trends from above are similar for the big (200MB) dataset. Our implementation still beats Hadoop, but it seems like our implementation does not scale as well as Hadoop. The approximate slope of the Hadoop trendline for the simple filter is roughly flat, while the SDFS (our) one seems to be growing almost exponentially. We suspect that they would cross once datasets and the number of VMs grew large enough. The inefficiencies could be due to increased network traffic while not increasing parallelism; for filter (and join) we fixed the number of tasks to 10, so there was little additional benefit to adding more VMs if we had no more tasks to give them. Our simple regex was a fixed word and our complex one was a day and date range.

Sayan Sisodiya (sayans3)
Vraj Patel (vrajsp2)





Our implementation also handily beat Hadoop for SQL joins on datasets of sizes 2 to 200MB. I still think this could be because Hadoop's implementation is more resilient and general; for example, HDFS chunks input files into 64MB chunks when they are added, which introduces overhead that is not present in our implementation. Additionally, I believe our implementation continues to slow at a roughly linear rate as the dataset size increases because our number of tasks was fixed at 10, so we had limited parallelism and each of the tasks grew to contain a sizable amount of the overall data. In contrast, Hadoop can increase the number of Map or Reduce Tasks dynamically, which likely explains why the time seemed to level off in an asymptotic manner as the dataset size increased. Our simple and complex joins different in "hit" rate in the relevant columns; our simple hit rate had an approximately 80% hit rate while our complex one was around 95%.