MP3 REPORT – GROUP 20 (gnarang2, hnehra2)

We replicated Hadoop in this MP using the simple distributed file system from previous MP making slight changes to incorporate directories in the SDFS by changing characters. Our system had a master (name node) responsible for storing all the files' metadata, replicating files to ensure fault tolerance up to 4 simultaneous failures and scheduling MappleJuice tasks. Only one job was permitted at a time, however, clients could queue up multiple jobs which were scheduled according to the order in which they were requested. For any job, the client contacted the master sending information regarding the job including the executable associated, files to be parsed, and in case of Juice whether to delete the files or not.

For Mapple, the client would first PUT the executable and the files into the SDFS if they are not already present and then ask the master to schedule the required job. The master would take the information provided by the client and create a Mapple task with the given number of tasks contacting each data node and telling them what files to parse, the output to be produced through the executable. The data node upon receiving the job would first fetch the files if they are not storing it themselves and then perform the job by creating a separate process and running the executable. The data nodes were also supplied by the offsets to parse (decided by the number of tasks specified by the client) and then create separate intermediary outputs. Therefore, each data node first parsed through the offsets creating files for each key for each machine. The master periodically contacts the data nodes asking them regarding the progress of the job marking status updates accordingly in the task structure at the master for the specific task being run. After this job is completed, the master selects two servers to consolidate the keys present in different data nodes to create one file per key. In other words, each data node creates one file per key based upon the offsets and all of these files are consolidated in the consolidation task. Whichever data node completes the task first puts the file in the SDFS and the master informs the other data node to kill the process it was running. Any failure in between either the consolidation task or the pre consolidation task are treated appropriately as soon as the master detects the failure. Upon detection of failure the master reschedules the jobs supposed to be done at the failed data node at another data node and simply runs the consolidation task. All of this is managed by communication between the name node and the data nodes through TCP communication. Once the consolidation task is completed and the output is put into the SDFS the master informs all the data nodes storing intermediary output to delete the files and background replication ensures output is stored at 4 different servers. Juice works in a similar manner, however, the client cannot perform a PUT operation, or in other words, the juice task's input (besides the executable) needs to be already present in the SDFS, otherwise the operation would fail appropriately informing the client about the inability to perform the operation. Furthermore, after the juice task is completed the client is given the option to delete the Juice task's input.

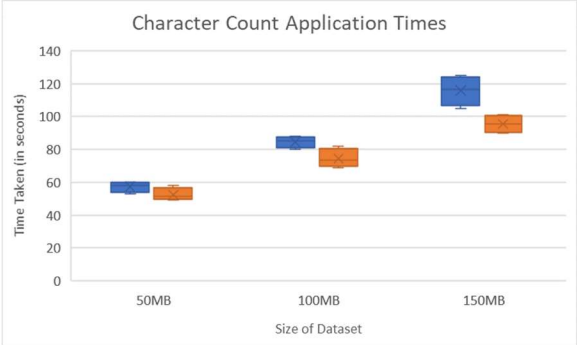
Scheduling at the master for each of the jobs is offered through range and hash partitioning. Range partitioning allows the task to be divided into subtasks based upon the file size and the number of tasks specified by the user in Mapple or Juice. For hash partitioning, Juice task is

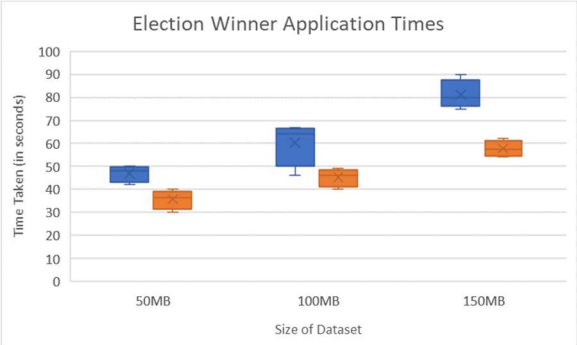
divided into sub tasks specified by the user, however, instead of having equal number of keys, each data node is made responsible for having approximately equal job in terms of job size.

Our system was equivalent to Hadoop in terms of time taken for smaller input files, however, for bigger input files due to our replication structure files were replicated fully first and then output was returned to the client, thus becoming slower overall. Resource allocation for our system was not uniform too, and therefore it contributed to our system being slower than Hadoop. Due to this, our system was unable to beat Hadoop in both the applications, by approximately 20-25 seconds.

Applications:

1) Voting Count (Dataset size: 50MB) – Given a file consisting of names of candidates, each name consisting of vote casted in that candidate's favor, we found out the winner or winners.
2) Character Count (Dataset size: 100MB) – Given a file consisting of words, we found the character appearing the max number of times. This application could also be modified to output the number of unique characters with the count of how many times they appear.





In the above diagrams, it can be seen that our system (blue) is performing more slower than Hadoop as the file size increases. The explanation behind the same is that is the difference between MappleJuice and MapReduce. In MapReduce the intermediary output is not fully replicated (in between Map and Reduce), which we are doing. Furthermore, we were able to receive these figures by changing our design (in which the MappleJuice is performed on SDFS files instead of local files). Scheduling could be improved as well because the master asks the progress of the tasks periodically instead of tasks informing the master in an asynchronous manner regarding completion of the task; we chose to do the same for simplicity purposes. Overall, we were able to achieve similar timings to MapReduce boasting an average of 57 seconds, 85 seconds, 115 seconds in Character Count application compared to Hadoop's 52 seconds, 74 seconds, and 90 seconds for file sizes 50, 100, 150 MBs and an average of 45 seconds, 62 seconds, 82 seconds in Election Winner application compared to Hadoop's 36 seconds, 45 seconds, and 58 seconds for file sizes 50, 100, 150 MBs