CS425, Distributed Systems: Fall 2023
Machine Programming 4 – MapleJuice+SQL

Group No: 58 **Names (netId):** Risheek Rakshit S K(rrs7), Siddharth Lal (sl203)
Repository link: *https://gitlab.engr.illinois.edu/rrs7/cs425-mp4.git*

The MapleJuice is built using Golang and we have built on top of our MP-2 and MP-3, i.e. the gossip based membership list implementation and the Simple Distributed File System. The first machine we start acts our introducer and the leader node for the MapleJuice, and hence it must not crash. We upload all our input files and executables on the SDFS, and then run the commands for Maple and Juice phases respectively. A user can type the maple/juice command on any machine to start the jobs. The user can even type in an SQL query (FILTER/JOIN), which will be parsed and then broken into Maple+Juice jobs. Apart from gossip, which uses UDP, we are using gRPC for every network call (SDFS, MapleJuice).

Design:

1)Setup: There will be one Leader and remaining worker nodes in the network. The leader's responsibilities are to schedule maple/juice tasks on respective workers. The workers in turn run the tasks and return the files to the leader.

2)Leader and Scheduling: The leader will (via gRPC) receive a request from any node that a maple/juice command needs to be run. For the maple, as per the num_maples passed, it will do a range-based partitioning of the input data in such a way that each maple gets equal number of lines. The maples are scheduled in a round-robin fashion, on the active worker nodes fetched from the membership list. The partitioning schema, along with the names of input files and maple executable names are sent to the worker (via gRPC). The worker will return the results; files in the form of intermediary_workerId_range_key.csv with key,value pairs, which are combined on the leader to produce the final intermediary_key.csv and is uploaded on SDFS. A similar procedure is followed in Juice, wherein different keys are allotted to different workers in a round-robin fashion. The keys, along with the juice executable is sent to the workers via gRPC. The worker will return the results in the form of intermediary_key.csv, which are combined on the leader to produce the final output.csv, which is uploaded on the SDFS. Due to this criticality, the leader cannot die as per our design. The leader also keeps track of assigned tasks via a global map present in it.

3)Worker: Workers run the executables via Exec command in golang. On receiving the maple/juice task, the worker fetches the intermediary files and the executable from the SDFS via the sdfs.get implementation done in MP3, and runs the executable. On completion of the task, it sends the result file to the leader via gRPC. In addition to this, the workers would send any incoming maple/juice commands written by the user via the command-line via gRPC to the leader.

4) SQL Layer : For the SQL layer, our parser will break the query into maple and juice commands, which are then queued with the leader (a Queue of maple/juice requests is maintained with the leader), and run one-by-one. We have given the liberty of entering the number of maple/juice tasks to the user.

5) Failure Handling: When a worker failure occurs, all the pending tasks on the worker are rescheduled by the leader, when the leader detects its failure via the gossip based failure detector implemented in MP2. The rescheduling is again done task-by-task on the active nodes as per the membership list, as per the global map implemented, which changes whenever a task is completed. Please note that completed tasks of the worker would not get rescheduled.

Experiments

For our experiments with comparison, we attempted to run a Hadoop cluster on the VMs. We were able to run a cluster of 5 datanodes, and 1 namenode, but we were unable to run golang executables of maple/juice tasks for the SQL queries on top of it successfully. Regardless, since our MapleJuice is working, we are presenting the numbers in the same setup just for the MapleJuice task execution of Filter and Join queries.

Filter Queries:

We performed the following query on a dataset [1], keeping num_maples and num_juices as constant, and varying the number of nodes in the system from 5 to 10. We take 3 measurements for each setting by varying the node from which query is executed.

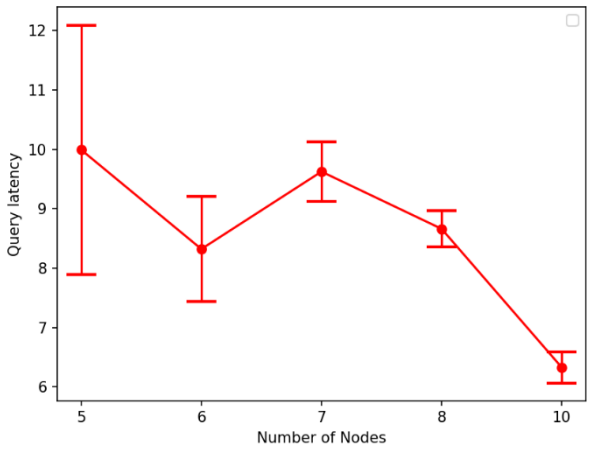

Figure 1

1) (Simple Query): *SELECT ALL FROM DataSet WHERE OBJECTID regexp ^9*

For this query, we observe that as we scale the network up, the query latency decreases. This can be attributed to the fact that tasks per worker are reduced (since we keep num_maples/num_juices as same), hence reducing the time taken by each worker to solve their allotted tasks. In a way, more worker nodes means more parallelization, hence decreasing query latency time. Figure 1 shows the plot in Query latency (in seconds) vs #VMs for the simple query.

2) (Complex Query): *SELECT ALL FROM DataSet WHERE Intersecti regexp \ /(?:B|O{2})*

For this query, we observe that, in comparison to the simple query, the complex query did take a miniscule amount of time more (in order of milliseconds). The similar pattern of latency decreasing as network is scaled up is seen for this query as well. Figure 2 shows the plot in Query latency (in seconds) vs #VMs for the complex query.

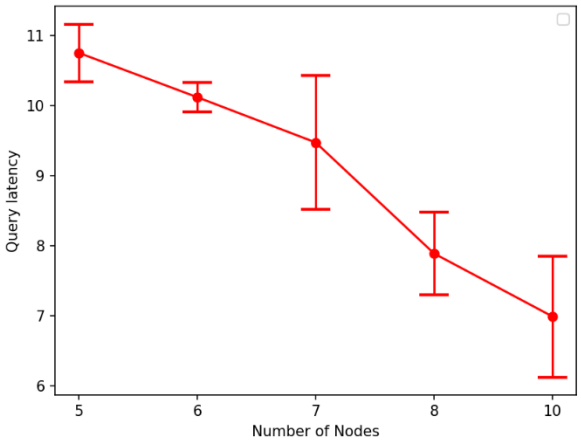

Figure 2

Join Queries:

We performed the following query on a datasets [2] , keeping num_maples and num_juices as constant, and number of nodes to 10. We trim the dataset on 5 settings, for sizes, in terms of lines of data: (120,240,360,480,600) ad take the latency measurements. We take 3 measurements for each setting by varying the node from which query is executed. Join queries are split into 2 maples (on the 2 datasets) and 1 juice jobs.

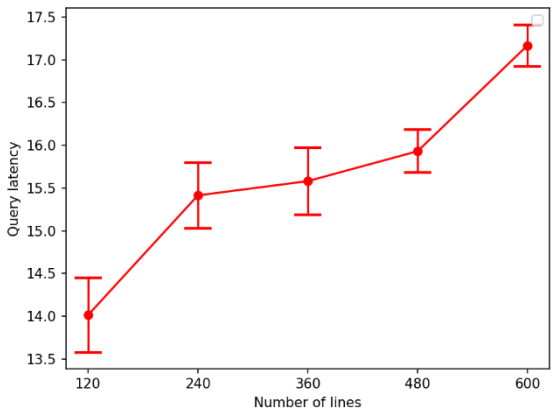

Figure 3

1) (Simple Query): *SELECT ALL from dataSet1 dataSet2 WHERE units = unit*

For the simple join case, we can observe that as number of lines increase, there is an increase in the query latency, on expected terms. Figure 3 denotes the query latency v/s number of input lines in the dataset(s). Note that 120 means lines are 120-130, and so on..

2) (Complex Query): *SELECT ALL from dataSet1 dataSet2 WHERE units = unit AND value > "20000"*

For the complex join case, we can observe that as number of lines increase, there is an increase in the query latency, on expected terms, similar to the simple join case. Figure 4 denotes the query latency v/s number of input lines in the dataset(s). Note that the latency is not that significantly higher as compared to the simple query case, due to the fact that there are still only 2 maple and 1 juice sub-jobs in the query.

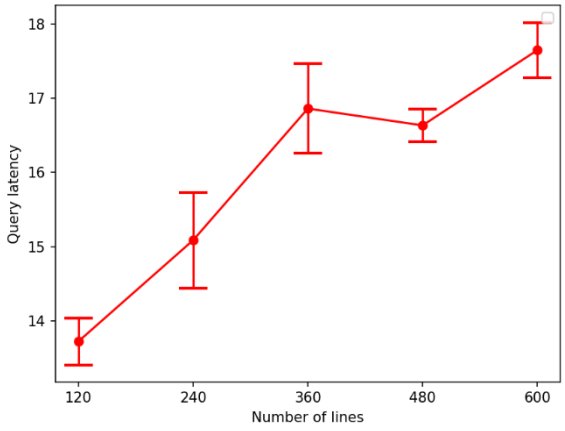

Figure 4

Dataset sources:

[1]:https://gis-cityofchampaign.opendata.arcgis.com/datasets/cityofchampaign::traffic-signal-intersections/explore

[2]: https://www.stats.govt.nz/large-datasets/csv-files-for-download/