CS 838: Assignment 1

Group 3: Subasree Venkatsubhramaniyen, Mona Jalal, Margaret Pearce

Question 1

Part a

We ran queries 12, 21, 50, 71, and 85 under both the MR and Tez frameworks with Hive and tracked the start/end times. For each query, Tez outperformed MR. In some cases the difference was drastic (e.g. Query 12, Figure 1) and in others it was subtle (e.g. Query 50, Figure 1). Tez removes some of the overhead associated with standard MR by allowing more processing steps to be performed in memory before writing to disk, hence it avoids the intermediate step of writing to disk when possible and increases the overall processing speed.

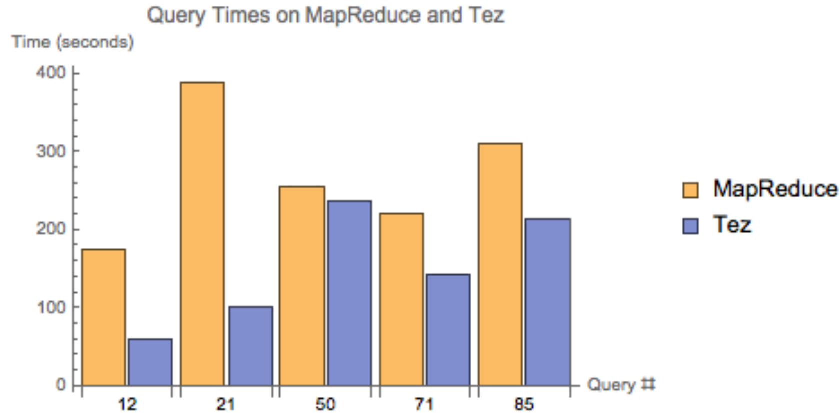


Figure 1 Completion time for running queries 12, 21, 50, 71 and 85 using Hive/MR and Hive/Tez.

Part b

Network read and write is less in Tez comparing MR because Tez does not write to HDFS for each reducer. Tez performs local reads, so it does not need to access HDFS each time. As an example, looking specifically at query 21 (Figure #), network read is high for MR. Comparing the network read values to the DAGs for MR (see Figure #), we see that one of the early stages does 22 mappers and 24 reducers. The 22 mappers need to read from HDFS, and the output from the 24 reducers needs to be read by successive mappers. By contrast, we can see in the DAG for Tez that there are four tasks without parents, which are the only tasks that read from HDFS. The remaining 7 tasks do not need to read from HDFS because Tez is optimized to do local reads. Similarly, we see much higher network write values for query 21 in MR compared to Tez.

Query 71 is the only example we saw where Tez performed worse in terms of network read and write compared to MR. The DAG for Tez is much more complicated in this example. We believe Tez is optimizing to minimize query completion time at the expense of increased network load. By observing the DAG for Query 71 (Tez), we can find that it uses more number of “BROADCAST” data movement operations, which we suspect is due to the join operation in the query. This means that the subsequent tasks need to read all the data from the previous task rather than from the desired partitions.

Except for query 21 and 50, storage read is higher with Tez with query 85 being significantly higher. In terms of storage writes, Tez is much higher for query 12 and 21 compared to MR. Aside from query 50, Tez is slightly higher than MR for storage writes. In short, network read/write seems to be higher for MR and storage read/write higher for Tez.

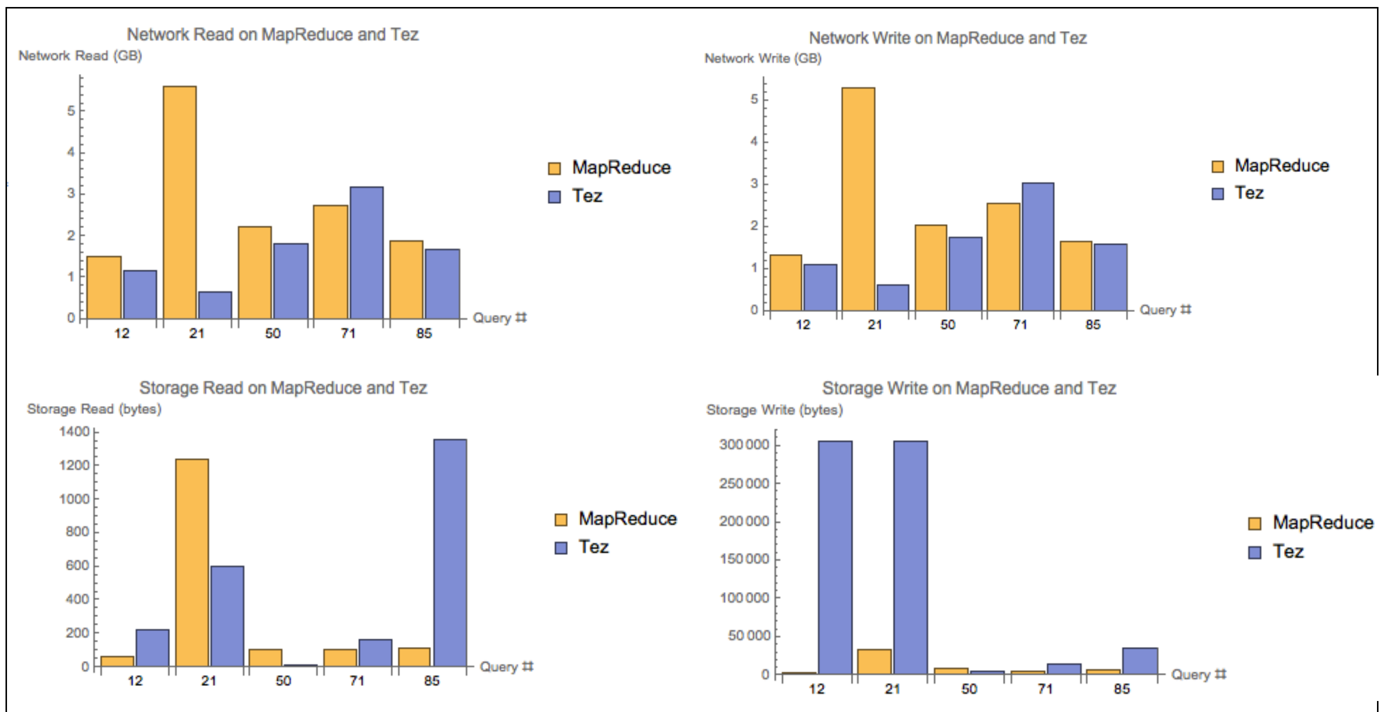


Table 1 Comparison of network reads, networks writes, storage read and storage read for MR and Tez jobs.

Part c

Query #	Type of Job	#Tasks	#Tasks that read from HDFS	#Tasks that aggregate data	Ratio
Q12	MR	23	13	10	0.769
Q21	MR	64	38	26	0.684
Q50	MR	44	25	19	0.76
Q71	MR	46	43	3	0.07
Q85	MR	45	23	12	0.522
Q12	Tez	10	3	7	2.333
Q21	Tez	11	4	7	1.75
Q50	Tez	14	5	9	1.8
Q71	Tez	22	8	14	1.75
Q85	Tez	22	8	14	1.75

Table 2 Comparing Hive/MR and Hive/Tez tasks.

We believe that the metrics shown in Table 2 have an impact on performance. Tez has fewer number of total tasks compared to MR for the same query, resulting in less resource contention and speeding up queries. Additionally, Tez has more number of tasks that perform aggregation than MR. This implies that fewer tasks are required to read from disk, and as a result Tez performs faster than MR.

Furthermore, looking at Tez DAGs, we observe that there are reducers following reducers rather than the traditional MR pattern of reducers followed by mappers. This implies that the HDFS writes by intermediate reducers are minimized, which makes Tez queries much faster.

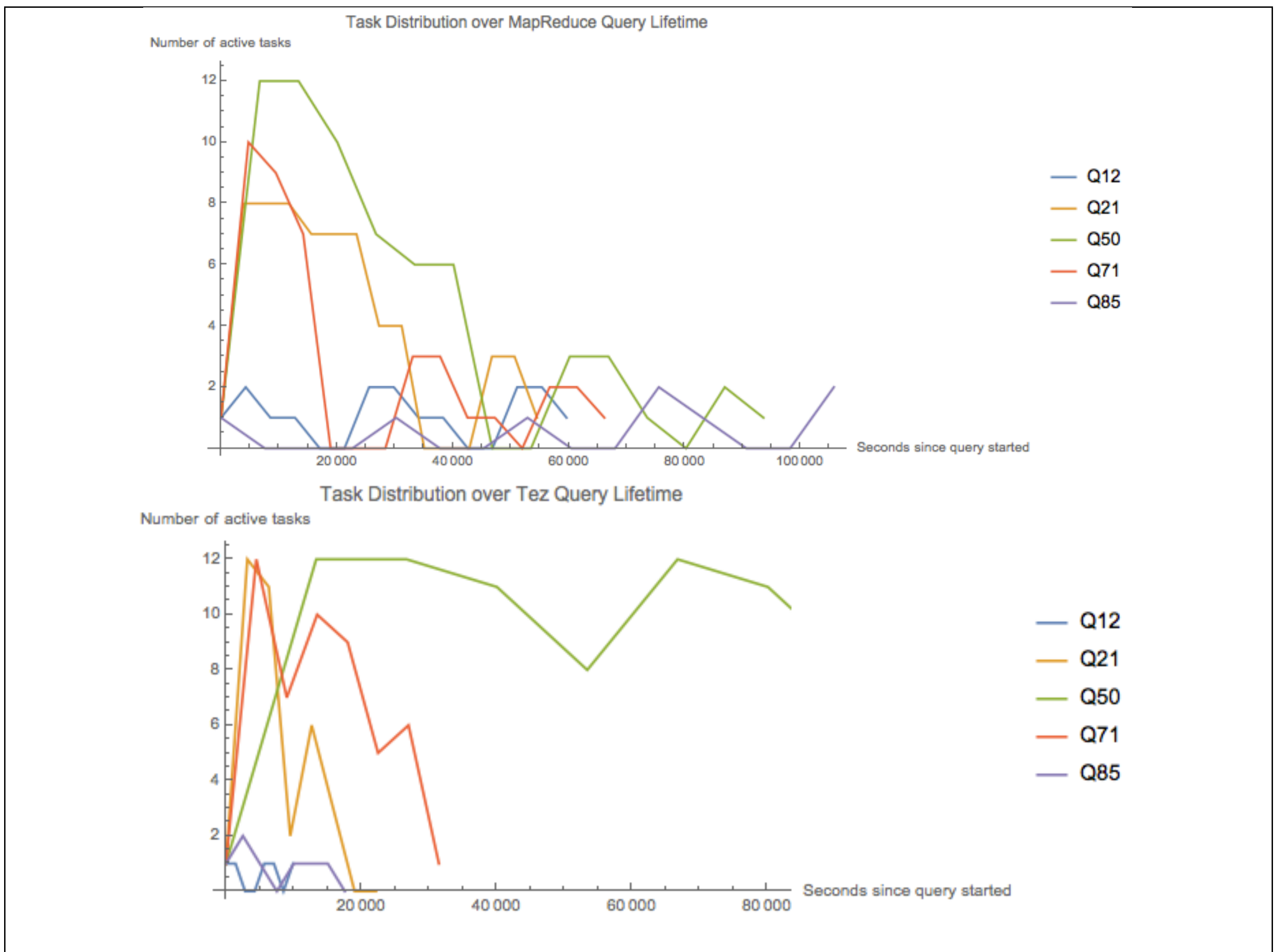


Figure 2 Task distribution for MR and Tez query lifetime.

To see how many tasks are running at a given time, we wrote a script to parse the set of .jhist files for each query. Specifically, we extracted the start and finish time for each task. We used this data to find the earliest start time among all tasks as well as the latest finish time. Next, we picked a fixed number of intervals and split up the query lifetime into time intervals. We looked through each task to determine whether or not it was active in a given interval and summed up the total number of active tasks in an interval. The resulting query distribution graphs are given in Figure 2.

Looking at the query lifetime graphs, overall, there is an increase in the number of tasks at the beginning and later a decrease toward the end of the query. There are certain time intervals for each query when there are no MR tasks running. However, there is an increase in the number of running tasks after this interval. By looking at the output files, we can see that there are periods of time when stages are being filtered out dynamically. When one stage ends in MR, the next stage doesn't start right away. There is some overhead required to initialize a task. For example, by examining the .jhist files, we see different containers are being initialized before each task (containers are not reused within MR). Though the shuffle phase can get started before all mappers finish, the actual reduce tasks wait for the output from the mappers. Hence, this also accounts for no active tasks at a given time interval.

Part d

To create DAGs for Hive/MR, we examined the stages in the .out file generated by each query and attempted to match up HDFS data read/write amounts between stages. Next, we examined the .jhist files for each query and drew the full non-optimized query plan as indicated in the adjacency tags. The adjacency tags represent dependencies between all stages. However, not all stages are executed when the query runs. We extracted

the executed stages in the output file from the full DAG to generate the fully optimized and executed DAG. Results from this process are depicted in Figure 3.

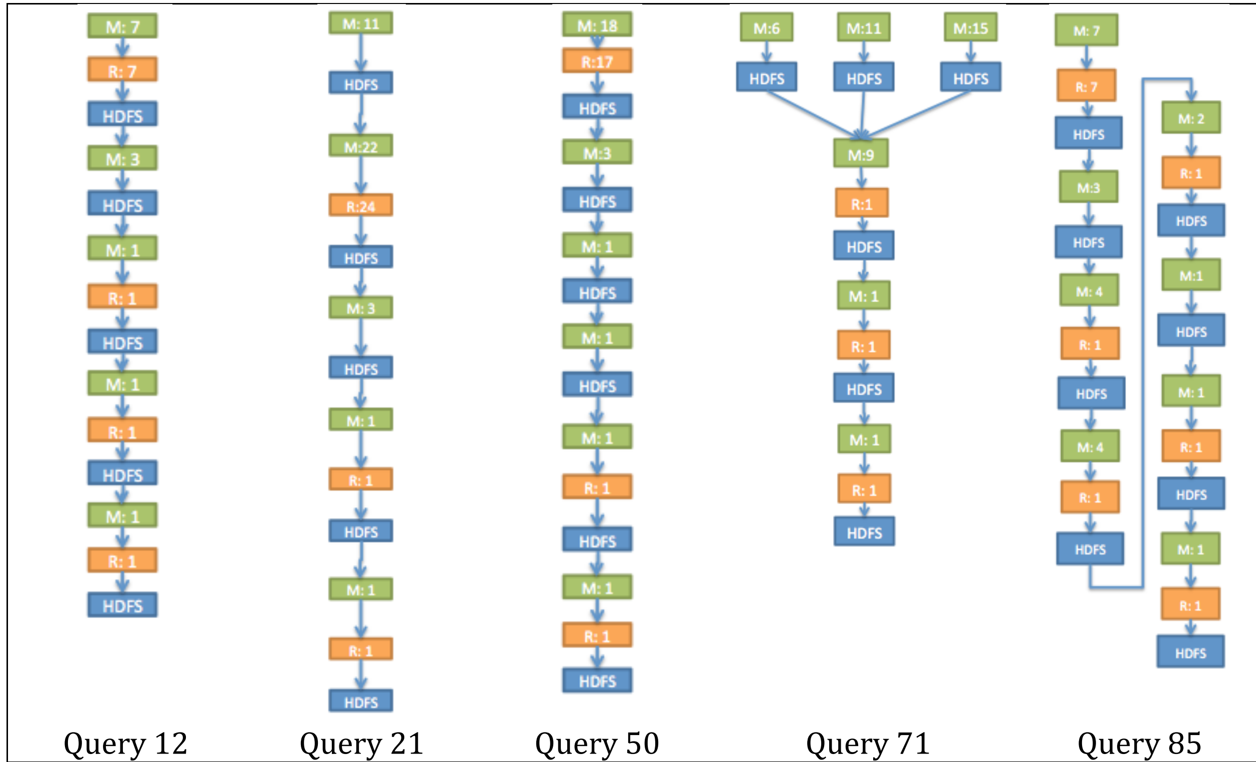


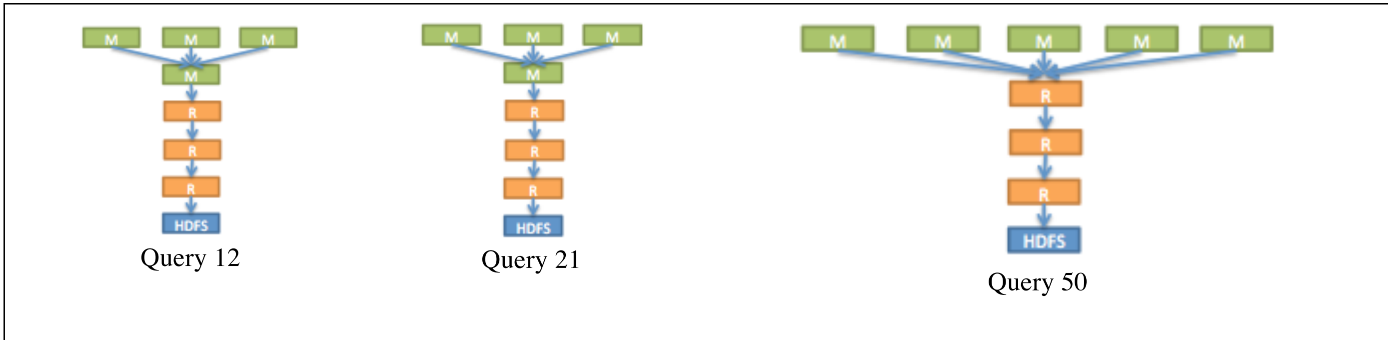
Figure 3 DAGs for MR on Queries 12, 21, 50, 71, and 85.

In order to create the DAG for Tez jobs, we first had to find which slave VM contained the application master. In order to do so, after running the Tez job we browsed to the YARN status URL and found the application_#_# for the current running job. Next, we looked in all the slave VM folders matching this application number to find the one containing the application master (indicated by container *001). From here, we grabbed the DAG file in dot format and converted it to PNG format using graphviz command:

"dot -Tpng input.dot > output.png"

Because Tez had only one dot file for each query, the process was more straightforward than creating the DAG file manually for MR jobs.

The DAGs for MR and Tez atop Hive are different in structure. The DAGs for Tez are significantly more complicated in structure because they are optimized for each query. We noticed in the Tez DAGs there are reducers following reducers, which is not observed in the MR DAGs. As previously highlighted in this report, HDFS writes by intermediate reducers are minimized, which improves query performance in Tez. MR tended to schedule sequential stages that required intermediate writes on HDFS, which added overhead.



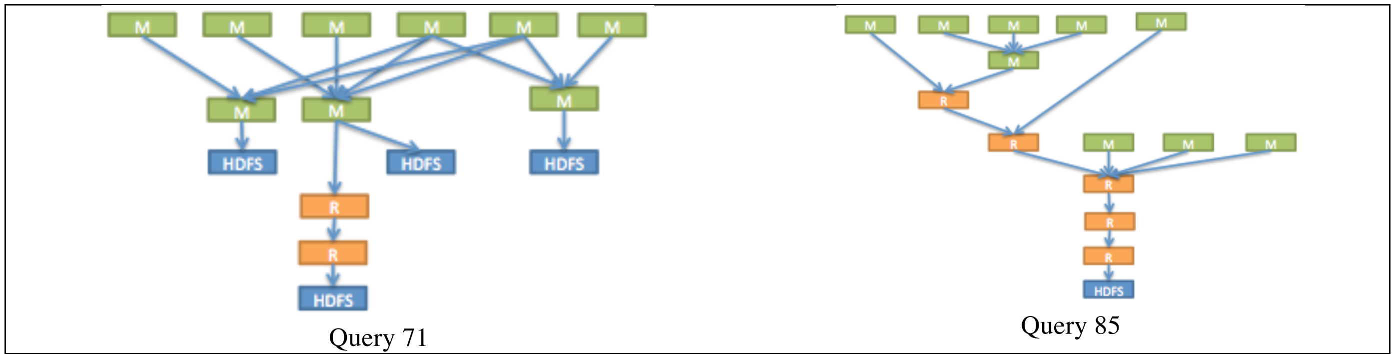


Figure 4 DAGs for Tez on Queries 12, 21, 50, 71, and 85.

Question 2

Part a

Table 3 Adjusting reducers

# of reducers	Average query time
1	707.5
5	417.5
10	377
20	387

Table 4 Adjusting # of parallel copies

# of parallel copies	Average query time
5 (default)	377
10	415
15	370
20	371

Table 5 Adjusting % completed maps

% of completed maps (before reduce starts)	Average query time
0.05	359
0.25	360
0.5	351
0.75	353
1 (default)	370

We changed the number of reducers using "mapred.reduce.tasks" to 1, 5, 10 and 20 as a script parameter. The best performance was realized by having 10 reducers (Table 3). By increasing the number of reducers from 1 to 10, we saw improvement in the performance, and then by changing from 10 to 20 we observed performance degradation. This could be due to two reasons. One explanation is that 20 reducers were too many for the distinct key sets of map output, where some of the 20 reducers were initialized with no datasets being hashed to them. This adds to the time required to initialize reduce tasks which are never used. Another reason could be the network overhead caused by 20 reducers during the shuffle phase.

Setting the number of reducers to 10, we varied the number of parallel copies by changing the mapreduce.reduce.shuffle.parallelcopies parameters to 5, 10, 15, and 20. Having 15 parallel copies achieved the best performance (Table 4). While having some number of parallel copies can be useful, after a specific threshold, communication overhead and resource contention will outweigh the efficiency gains and we will see performance degradation. Increase in number of parallel copies increases the number of network connections for every reducer to read from every mapper output. If the job creates huge intermediate outputs, then a smaller number of parallel copies might work better.

Having set number of reducers to 10, and number of parallel copies to 15 for MR tasks, we started changing the *"mapreduce.job.reduce.slowstart.completedmaps"* parameters to 0.05, 0.25, 0.5, and 0.75. This parameter indicates at which percentage of map tasks completeness reducers can start. We saw the best performance when we started the reducers after 50% of map tasks completed (Table 5). Starting reducers early can help because it will start the shuffling phase and distribute the data from mappers to reducers, which is beneficial when network bandwidth is a bottleneck. Setting the value to 1 might delay the query execution, as the shuffling phase does not start until all the maps are completed. However, starting reducers too early comes with a tradeoff: if a reducer starts much ahead of the completion of the map tasks, it will need to wait for map tasks to complete. In the meantime, it is using up resources without performing any work that could have been allocated to a pending map task.

Part b

Table 6 Setting reduce enabled

Reuse enabled?	Average query time
True	83
False (default)	100

Table 7 Adjusting # of parallel copies

# of parallel copies	Average query time
5 (default)	83
10	80
15	78
20	77

Setting "tez.am.container.reuse.enabled" to true enabled us to reuse containers without requesting new ones from ResourceManager (RM). The average query lifetime is small, and hence the individual tasks take smaller time to complete. However, some amount of time is spent in initializing the task by requesting resources from NM. Enabling container reuse reduces the task initialization time as AM assigns a pending task to a node when one of its tasks get completed. In short, setting the flag avoids the overhead of reallocating container resources, which ended up in better performance result (Table 6).

Leaving the previous parameter set to true, we changed the tez.runtime.shuffle.parallel.copies parameter to 5, 10, 15, and 20. The best performance result was achieved by using 20 parallel copies. In this case we saw consistent improvement while increasing the number of parallel copies, however the improvement was very small. Tez does not require as much network transfer compared to MR: in some cases, not much data is transferred over the network across MapReduce phases because the outputs of every reducer are not written to HDFS. Also, Tez has an optimized DAG in place with one stage. This explains why trying to minimize network transfer time through parallel copies doesn't have a significant impact on query completion time for Tez.

Part c

Table 8 MapReduce queries with 2(a) settings

Query #	With 2(a) flags?	Completion time
12	No	174
12	Yes	170
50	No	254
50	Yes	222

Table 9 Tez queries with 2(ab) settings

Query #	With 2(ab) flags?	Completion Time
12	No	59
12	Yes	48
50	No	237
50	Yes	167

Having 10 reducers, 15 parallel copies, and 50% map tasks completeness for starting the reducers in Hive/MR jobs, we got the best performance for queries 12 and 50 as well (Table 8). Similarly, setting container reuse in Hive/Tez jobs to true and having 20 parallel copies gave better performance for queries 12 and 50 (Table 9). The network was a factor in all of the queries, so tuning factors that are impacted by network bandwidth are likely to behave similarly across different queries. We are making similar tradeoffs on starting reduce tasks too early and hogging the network bandwidth versus starting too late and not distributing transfer tasks over time/ creating a bottleneck. Improvement amounts may vary based on query-specific factors, such as the amount of data that is returned from each query, how many mappers and reducers are used, how often data is written to HDFS, and so forth.

Question 3

We executed query 71 (MR and Tez) several times on the same day as well as the same timeframe to find the average query lifetime. Next, we computed the values corresponding to 25% and 75% of the query lifetime to identify the time periods at which the NM had to be killed. We initially decided to use the .jhist files to identify the slave VM that has an active MR task during 25% or 75% of the query lifetime. However, we realized that those files get written only after the submitted job gets completed. Hence, we had to make use of .out files to identify the slave VM.

Figure 5 Query 71 total job time baseline system, killing the NM after 25% of total job time, and killing the NM after 75% of the total job time.

Below are the steps we followed to kill the NM using an automated script:

1. Wait for the required number of seconds (as calculated from 25% (or) 75% query lifetime).
2. Identify the VM that has the container *001 which is the ApplicationMaster (AM).
3. Identify the latest started job that has not completed from the .out file.
4. Extract the task ID from .out file and obtain the slave VM from the application history server URL for that task.
5. Check and kill the slave VM if it does not host the AM else repeat the previous step to get another VM that has an active task using the following command:

./yarn-daemon.sh stop nodemanager

6. After the query completes, we run script to start the NM that we killed so as to not affect the subsequent jobs.

After the NM is killed, the RM will detect its failure through heartbeat timeouts. RM will report the failure to all running AM, which are responsible for redoing work done by containers on the failed node.

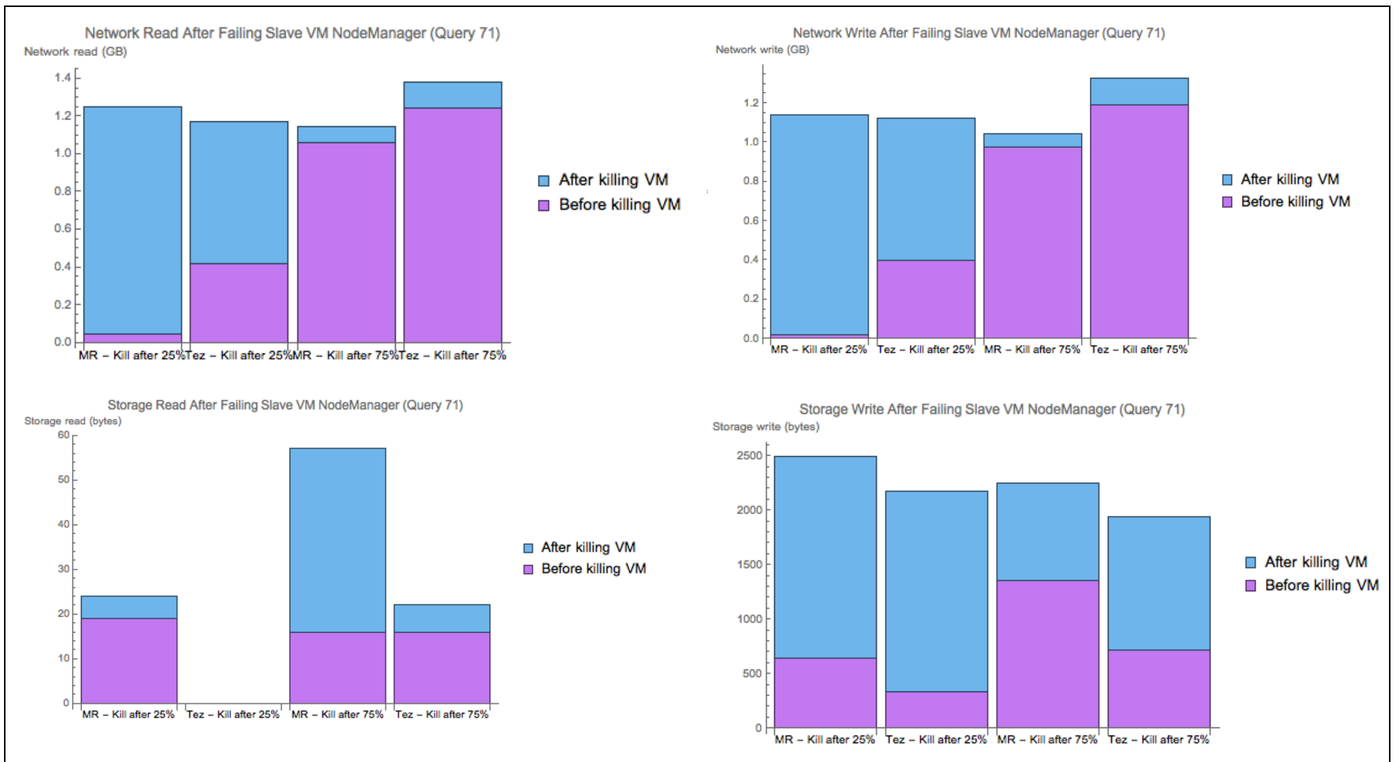


Figure 6 Network read, network write, storage read, and storage write for query 71 after failing the NM in a slave VM.

Depending on how much work needs to be redone, this could significantly increase the query time (see Figure 5). For MR, we saw that killing the NM after 25% of the query lifetime resulted in the first stage (consisting of all mappers and no reducers) being drawn out from a few seconds to over one minute. This implies that MR needed to re-read everything in this stage from HDFS, increasing query completion time

and network read amounts. Killing the NM after 75% resulted in the second stage (consisting of mappers and reducers) being impacted during the end of map phase and beginning of reduce phase. Hence, the map tasks and the shuffle phase for all the reducers to read the map output from the rescheduled task needed to be rescheduled. The in-progress reduce tasks on this node needed to be rescheduled since NM is inaccessible. This resulted in increased storage read and query completion due to rescheduling overhead.

In Tez, we suspect that mostly reducers are running at the 75% point where the NM is killed. Referencing the DAGs in Figure 4, we expect that we are left with tasks that do not read from HDFS, but instead perform local reads. Although restarting these tasks will take more time than if NM were not killed, it will take less than killing NM at 25% of the query lifetime. At 25%, there is a greater chance of needed to re-read from HDFS for rescheduled tasks. This explanation is also consistent with the network and storage results in Figure 6.

Other NMs will be assigned the incomplete work on the node with the failed NM, which may require NMs to copy any dependencies from the work on the failed node (data files, executables, tarballs) to local storage. This contributes to the increase in storage read and write after killing the NM. AM will also need to request additional containers from RM to perform the work.

Appendix

Here is the link to our Google Drive for the project which contains all the graphs, tables and scripts:
https://drive.google.com/folderview?id=0B2somm1FlLXjcTVPNmQzRDNKbm8&usp=sharing