Question 1

Part A

a) In this experiment, Spark is always better than Hive/MR and Hive/Tez. The difference is not always constant. Sometimes Spark is significantly way better than Hive/MR and Hive/Tez and sometimes its performance gain is negligible compared to Hive/Tez. Regardless, always Spark SQL is way better than Hive/MR. This is because Spark uses in-memory computation with minimized storage read/write and uses the notion of RDDs.

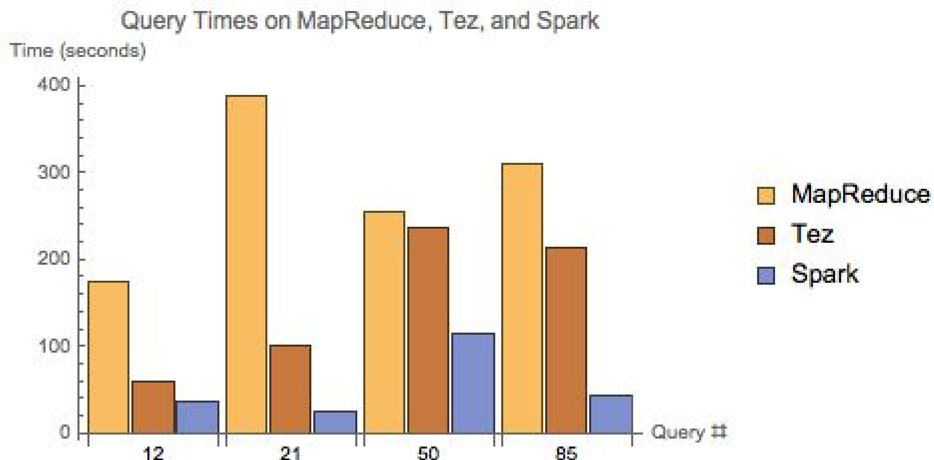

Figure 1. Query time comparison across Hive/MR, Hive/Tez and Spark.

b)

Query	Storage read (sum of Input across all stages)	Storage write (sum of Output across all stages)	Network read (Shuffle read across all stages)	Network write (Shuffle write across all stages)
12	7.127 GB	0 GB	1.072 GB	1.077 GB
21	1.139 GB	0 GB	1.179 GB	1.18 GB
50	20.448 GB	0 GB	3.76 GB	3.724 GB
85	7.797 GB	0 GB	1.667 GB	1.673 GB

Table 1. Comparison of queries 12, 21, 50 and 85 with respect to storage read/write and shuffle read/write.

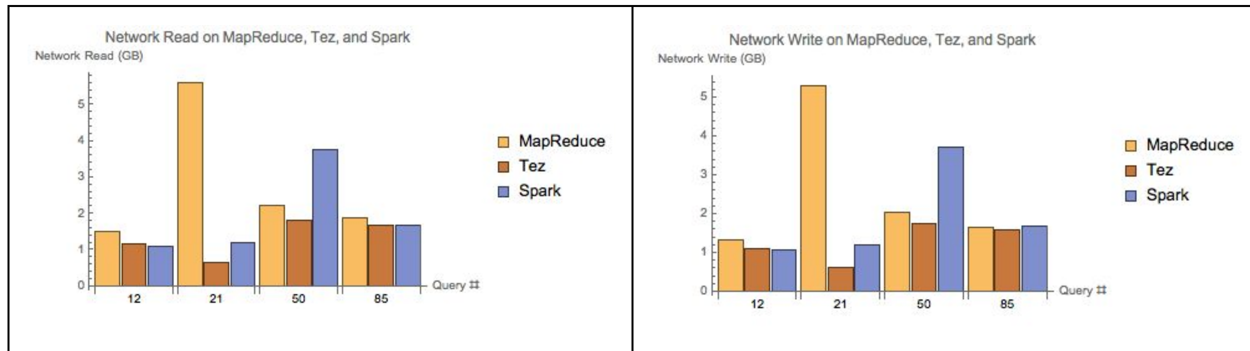

Figure 2. Network read and network write for queries 12, 21, 50 and 85 for Hive/MR, Hive/Tez and Spark.

Network read on Spark is synonymous with the sum of shuffle read values across all stages of a job. Similarly, network write on Spark is equivalent to the sum of shuffle write values across all stages of a job. As we can see in the graphs of network read/write on MapReduce, Tez, and Spark, the amount of data sent and received over the network in Spark is less than the other two frameworks.

The key idea behind Spark is to minimize disk and network I/O by using more RAM. Therefore, it makes sense that we see Spark using fewer resources compared to Hive/Tez and Hive/MR. Network read and write occurs during shuffle phases in Spark, which happen during specific operations (joins, repartitions, combine by key, reduce by key, and group by) between different stages. Spark does pipelining of operations like map, filter within a single node to minimize network I/O. In MapReduce, a shuffle phase occurs when data is transferred from a map task to a reduce task, so network I/O happens more frequently in MapReduce than in Spark.

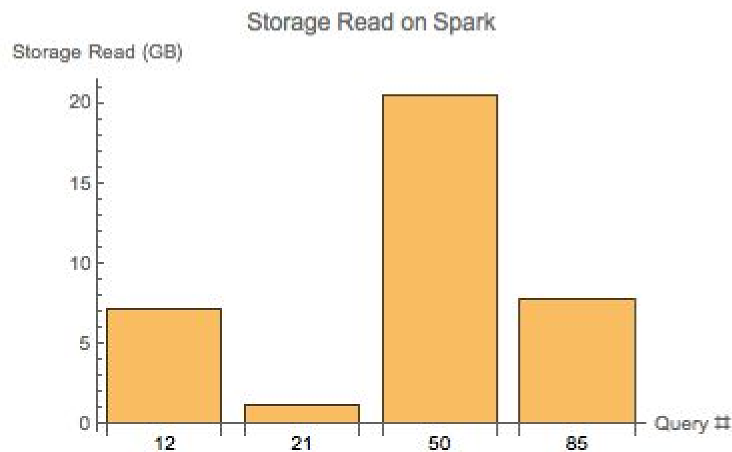

Figure 3. Storage read for queries 12, 21, 50 and 85 on Spark.

Storage read on Spark appears to be much higher than on MapReduce or Tez. For starters, our graphs for MapReduce and Tez use bytes as the units instead of gigabytes. This can partially be explained by the fact that the databases used for Spark were 50 GB and the ones used for Hive/Tez and Hive/MR were only 10 GB. Spark tries to minimize the storage read by performing

more in-memory computations. Hence, we expect the storage read for MR/Tez to be higher than Spark.

For each query, storage write is 0 GB. Data is only written to disk from Spark if .persist() is explicitly called on an RDD. Moreover, persist() must be called with an explicit storage level that involves a combination of disk storage and/or memory (e.g. DISK_ONLY or MEMORY_AND_DISK where the RDD won't fit in memory). Although Spark will call persist() automatically during shuffle operations, the default storage setting is memory only. By contrast, MapReduce and Tez both used disk storage for the queries measured in this question.

c)

Query	Framework	# tasks	# tasks that aggregate data	# tasks that read from HDFS	ratio of (# aggregate) / (# Read HDFS)
12	MR	23	10	13	0.769
12	Tez	10	7	3	2.33
12	**Spark**	**904**	**800**	**104**	**7.69**
21	MR	64	26	38	0.684
21	Tez	11	7	4	1.75
21	**Spark**	**856**	**800**	**56**	**14.28**
50	MR	44	19	25	0.76
50	Tez	14	9	5	1.8
50	**Spark**	**1206**	**1000**	**206**	**4.85**
85	MR	45	12	23	0.522
85	Tez	22	14	8	1.75
85	**Spark**	**1858**	**1600**	**258**	**6.20**

Table 2. Comparison of queries 12, 21, 50 and 85 across Hive/MR, Hive/Tez and Spark.

In Spark, query completion time decreases as the ratio of the number of tasks that aggregate data to the number of tasks that read from HDFS (last column in the table above) increases. Furthermore, this ratio corresponds with the amount of storage read from each query. Similarly, the number of tasks that read data from HDFS is also correlated with query duration. This

indicates that the slowest aspect of Spark is reading in data from storage. After that is done, work is done in memory and completes quickly.

Spark uses significantly more tasks than MapReduce or Tez on Hive. The ratio of aggregate to read tasks is also much higher for Spark for every query. This is visible on the plots of task distribution over query lifetime as well: Spark tends to use many more tasks at a time toward the end of a query lifetime, when more aggregation is being done. Tez and MapReduce had the number of active tasks spike in the beginning of the query (when tasks are reading data) and taper off at the end.

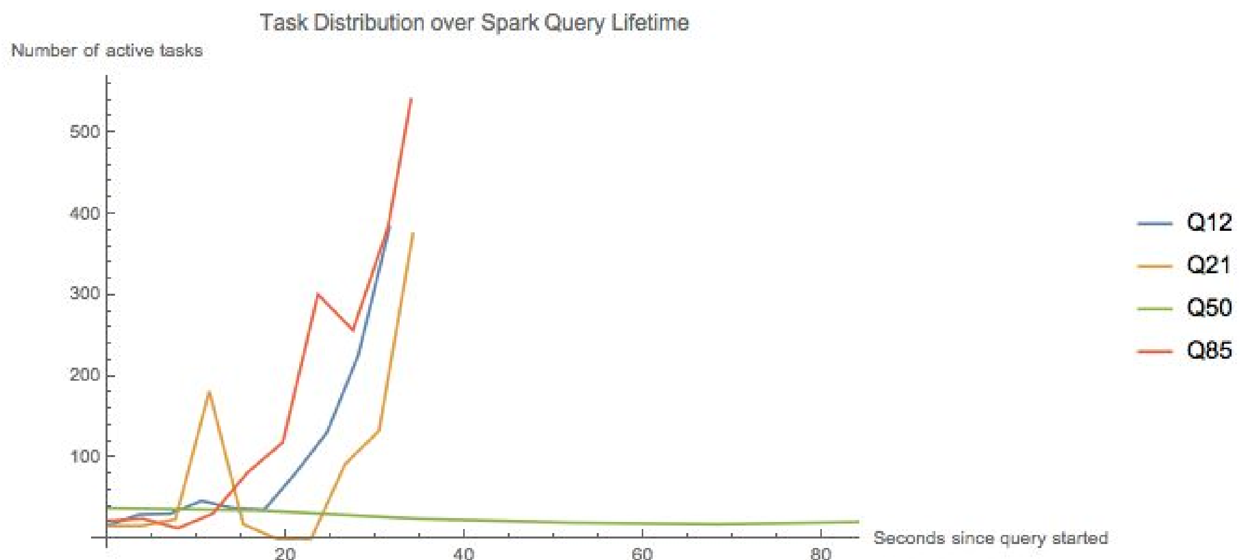

Figure 4. Comparison of queries 12, 21, 50 and 85 with regards to task distribution over Spark query lifetime.

d)

All the Spark DAGs have the same general structure. At first glance, it seems that there are not a lot of things done in parallel according to the DAG for stages, but in reality there are a lot of tasks done within each stage in parallel. Spark generates a DAG by pipelining as many narrow dependencies into a single stage, and then wide dependencies are handled by splitting the workload into multiple stages. This results in an optimal DAG, which positively impacts performance.

MapReduce generates its DAG by grouping Map/Reduce tasks and is not optimized. On the other hand, Tez does some optimization by reducing unnecessary reads and writes. Each framework has its own method of generating a DAG that is suited to how the framework operates, which explains why DAGs across queries within a single framework look similar but the same query has a very different DAG from one framework to another.

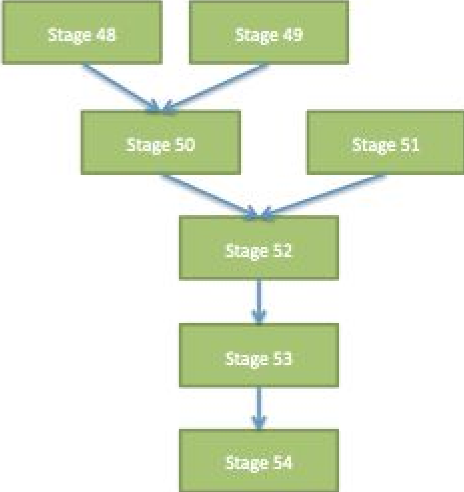

Figure 5. Spark DAG for query 12.

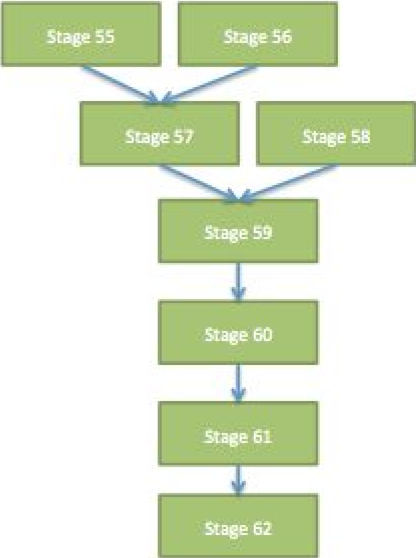

Figure 6. Spark DAG for query 21.

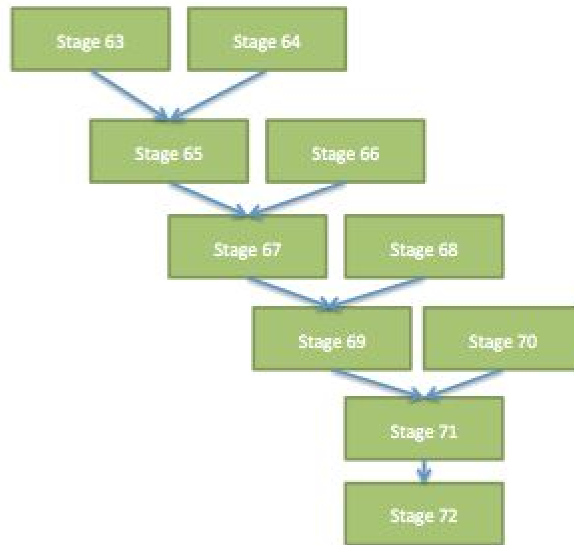

Figure 7. Spark DAG for query 50.

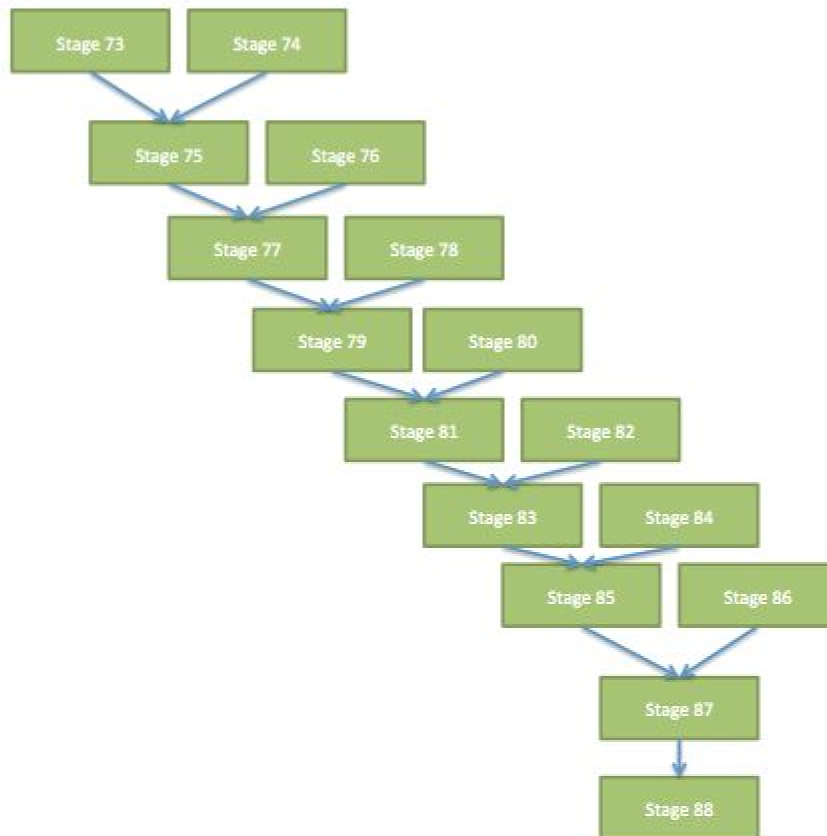

Figure 8. Spark DAG for query 85.

Part B

a)

`spark.sql.shuffle.partitions`	Query time	Storage read (sum of Input across all stages)	Storage write (sum of Output across all stages)	Network read (Shuffle read across all stages)	Network write (Shuffle write across all stages)
5	42 sec	1.1375 GB	0 GB	1.083 GB	1.085 GB
50	30 sec	1.1376 GB	0 GB	1.128 GB	1.13 GB
100	30 sec	1.1642 GB	0 GB	1.136 GB	1.138 GB
200	35 sec	1.1376 GB	0 GB	1.122 GB	1.126 GB

Table 3. Comparison of number of shuffle partitions with respect to query execution time and storage/network read/write.

Yes, when we change the number of partitions from 5 to 50, we see a great performance improvement. We don't see any change from 50 partitions to 100 partitions. However, when we double the number of partitions from 100 to 200, we observe performance degradation due to the number of shuffles being more than the number of unique keys, which may not be used. The reason that we don't have any storage write is due to fact that Spark does in-memory computations and we are not explicitly calling persist() to write to disk. The best value for this setting is 50 partitions.

b)

spark.storage.memoryFraction	Memory (RAM)	Query completion	Performance Speedup	RDD spilled?
0.02	1.512 GB	28 sec	1.035 (sped up a little)	No
0.05	3.78 GB	28 sec	1.035 (sped up a little)	No
0.1	7.56 GB	30 sec	0.966 (slowed down)	No
0.2	15.12 GB	29 sec	1 (no change)	No
0.4	30.24 GB	29 sec	1 (no change)	No
0.6 (baseline)	45.35 GB	29 sec	NA (baseline)	No
0.9	68.04 GB	30 sec	0.966 (slowed down)	No

Table 4. Studying effects of memory fraction and its relation to query completion time.

As we adjusted the memory fraction parameter, we did not see a noticeable change in performance for Query 21. Query time ranged from 28 seconds to 30 seconds. Under the default setting of 0.6, the query took 29 seconds. Furthermore, we never saw any data in the “Storage” tab on Spark’s web UI, so we never saw any RDDs spilled to disk.

Changing this parameter results in a change in the amount of memory that is used for caching. Even at the lowest setting we tried for this parameter (0.02), all of the data that is read can fit in the resulting memory reserved for cache (1.512 GB). Because we cleared the cache before running the query on each setting, the cache size was always sufficient and there was no impact on query performance. This also explains why we never saw the RDD spilled to disk.

Part C

Baseline time: 28
25% of baseline time: 7
75% of baseline time: 21

% of completion time	failure Type	Query completion time
25%	A	31
25%	B	50
75%	A	31
75%	B	51

Table 5. Comparison of query completion time when we have failures of type A and B at 25% and 75% of execution time

In case of failure type A, we noted that the failure did not make a significant difference to the query performance regardless of whether the failure was triggered at 25% or at 75%. Spark uses the lineage of RDD which if a failures happens it only re-executes the failed RDD and this feature makes Spark fault-tolerance to be efficient.

In case of failure type B, we killed a worker process on a desired VM. We chose to kill the worker process on the master VM. We observed that in general, all VM workers were used in each query for different tasks, so the choice of desired VM to kill a worker process on was arbitrary. We observed a significant performance degradation compared to the baseline and failure type A.

When a worker process is killed, the driver will reinitialize it. This adds overhead to query completion time. We do not see a change in the overhead based on whether the failure occurs at 25% or 75%. This is because when a worker process is killed, all its child processes are killed and leads to relaunching of the worker process by the driver.

Question 2

Part #	Query time	Network read	Network write	Storage read	Storage write	# Task
1	82.854 sec	1.079 GB	1.077 GB	7.127 GB	0 GB	904
2	205.178 sec	1.062 GB	1.062 GB	7.127 GB	0 GB	904
3	40.150 sec	1.054 GB	1.052 GB	2.714 GB	0 GB	904

Table 6. Comparison of two different failure types on the performance of query with baseline system.

Case 1: In this scenario, we ran the query without any persist() calls. The cache is empty and the query reads all the input tables from disk.

Case 2: When all of the input tables are persisted (fully in memory if they fit, otherwise there is spillover into disk). We used cacheTable() to save the input tables to memory. Here, the cacheTable() is a lazy computation that is done only when a query using the table is issued. Hence, the increased query completion time is attributed to the cacheTable() operation. The storage read values are equivalent to case 1 because we are reading in the entire input tables from disk.

Case 3: In this scenario, the output of second run is persisted. Here, there is no query overhead as with the cacheTable() function in Case 2. Furthermore, the persist() operation is not a lazy computation, so it does not impact the query completion time for Case 3. The time it took to persist the output in was 39 seconds. This case took the least amount of time to finish the query because it could benefit the most from cached data.

We see the same number of tasks in each scenario we tested. Caching the data does not appear to change the DAG structure; it just improves the time it takes for each scenario to complete.

Question 3

The output produced by our script is:
Product40

Product12
Product2
Product7
Product29

Note 1: The output is stored in a text file in HDFS located at:
hdfs:/HW2Question3/q3/part-00000

Note 2: The script generated for this query is located in the same tar as this report with filename: Question3.py