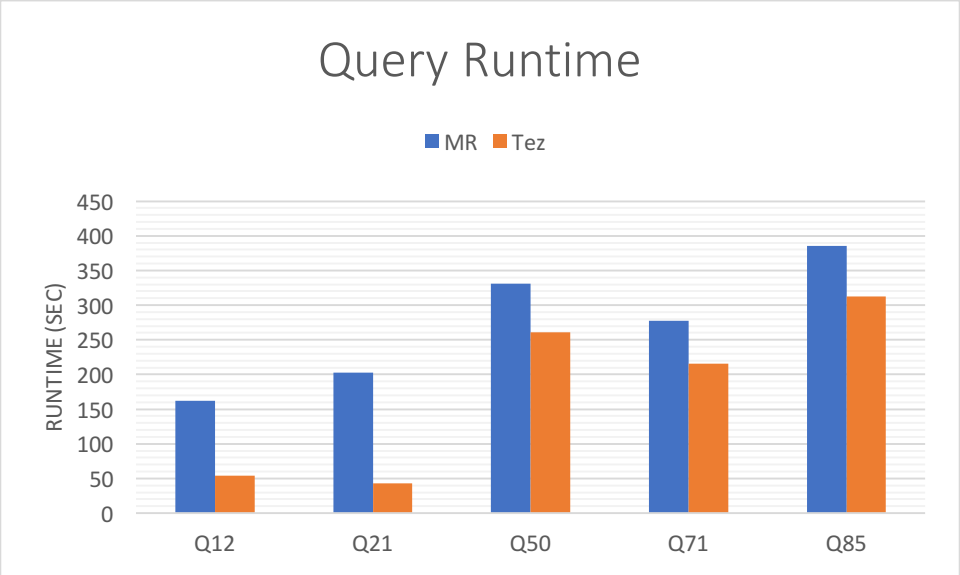
CS744- Big Data Systems Assignment 1

Part A: Experimenting with Performance

Question 1.a:

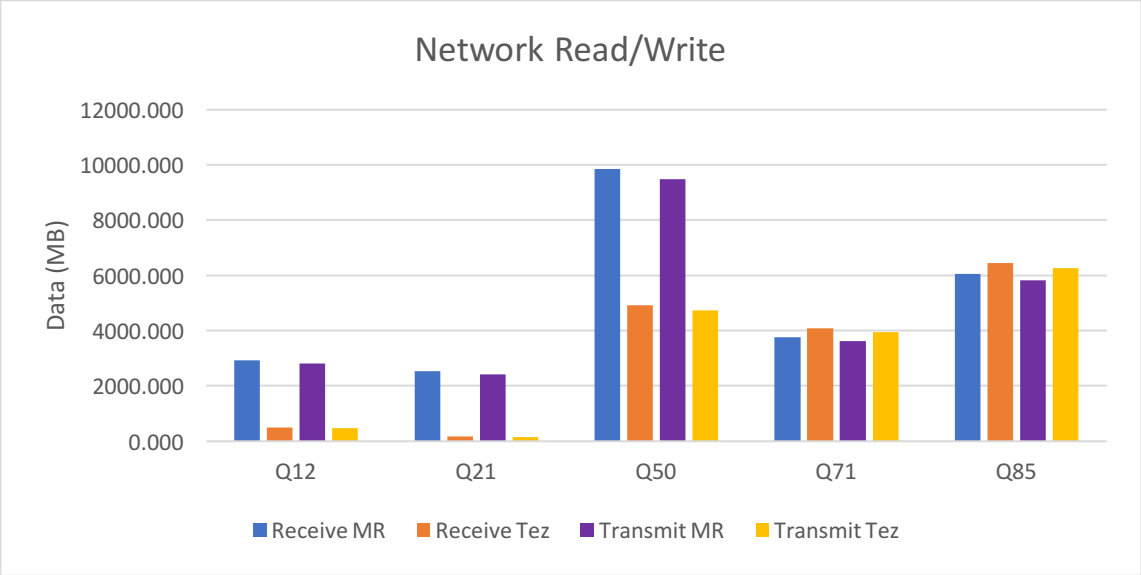


We observed that Hive/Tez always performs better than Hive/MR. The difference in their execution times as is evident from the graph ranges from 50 to 150 seconds. This difference is not constant because Tez outperforms MR due to lesser job splitting and HDFS access which varies with the query. Hence the difference in performance depends on the query and task structure.

Question 1.b:

Network Read/Write:

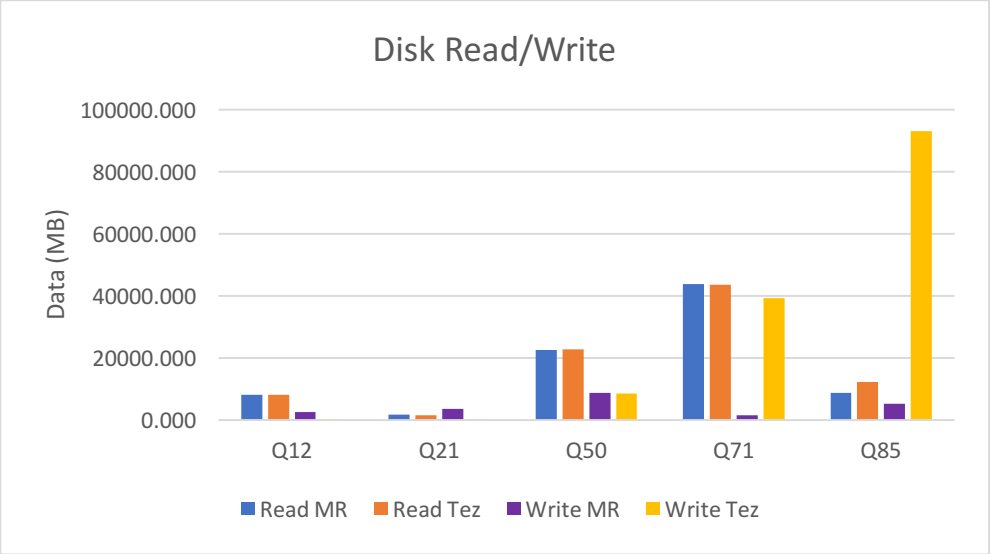
	Receive MR	Receive Tez	Transmit MR	Transmit Tez
Q12	2919.326	491.823	2803.587	472.554
Q21	2520.110	157.789	2422.097	152.074
Q50	9846.386	4914.664	9471.064	4736.557
Q71	3769.189	4091.044	3625.458	3932.666
Q85	6057.918	6457.164	5824.137	6255.535



Hive/Tez has lot lower traffic on network compared to Hive/MR for normal queries. For computationally expensive queries like Q71 and Q85, network traffic for Hive/Tez is slightly more than Hive/MR. The reason would be that due to many parallel branches in Hive/MR task, computationally complex queries gets better advantage of parallelism.

Disk Read/Write:

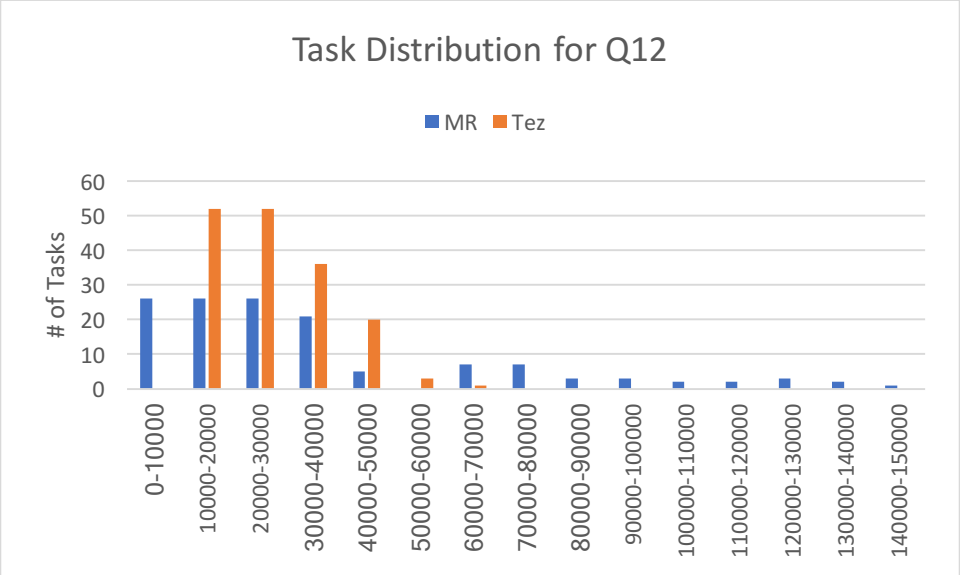
	Read MR	Read Tez	Write MR	Write Tez
Q12	8060.617	8085.811	2408.575	46.354
Q21	1559.560	1514.803	3517.546	19.751
Q50	22486.553	22628.650	8603.853	8389.030
Q71	43725.066	43602.008	1487.294	39144.700
Q85	8657.621	12095.812	5234.192	93079.700

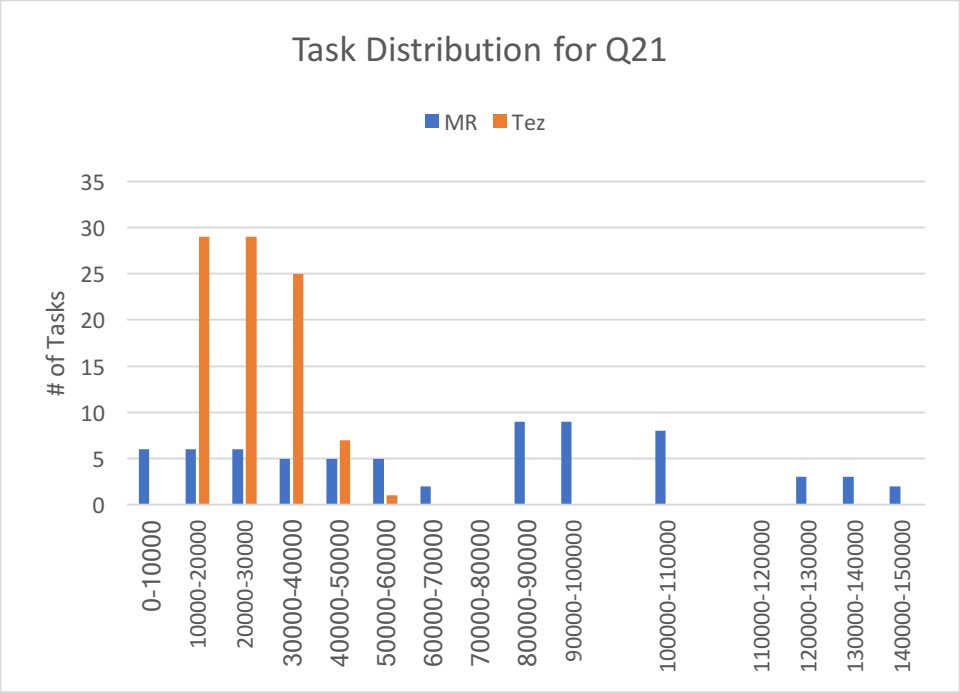


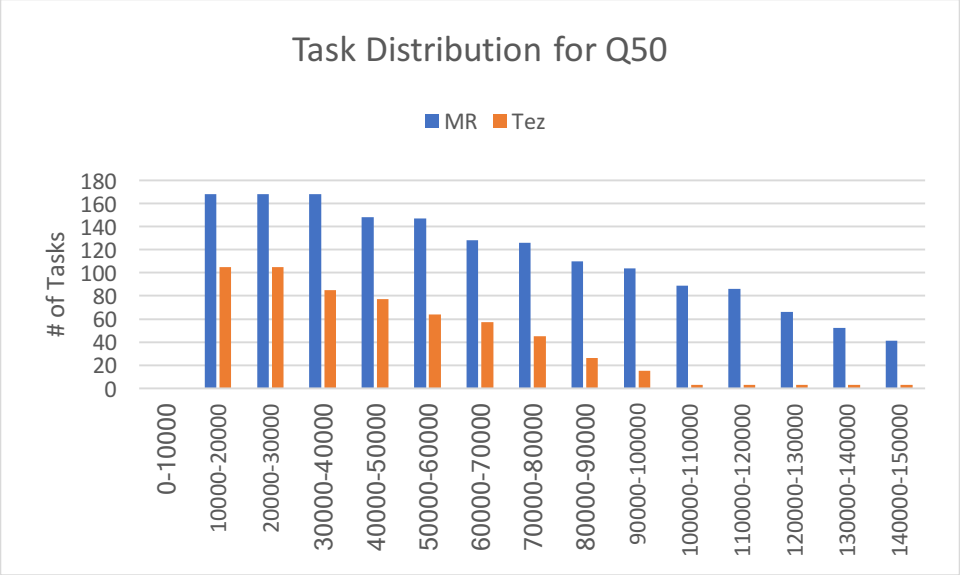
We observed that Disk reads in both Hive/MR nad Hive/Tez are almost equivalent. Disk writes in Hive/Tez are more than in Hive/MR and it increases logarithmically in Hive/Tez with increase in computational complexity of queries. This trend leads to reduction in the runtime advantage Tez has over MR.

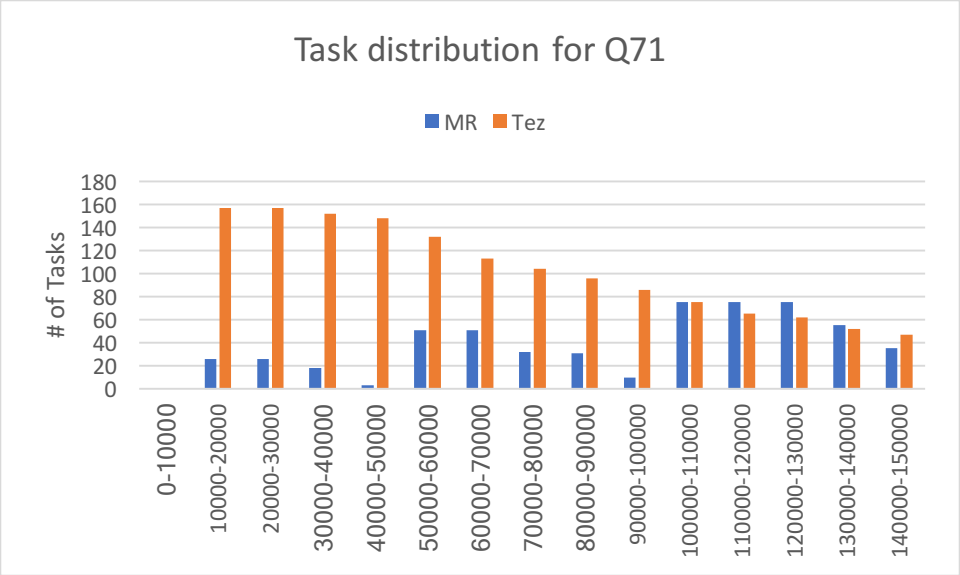
Question 1.c:

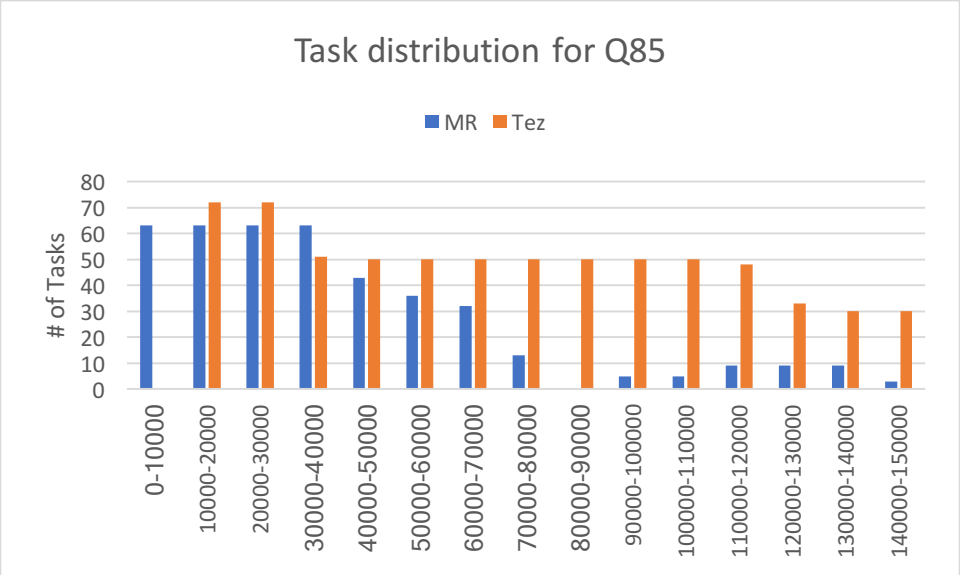
Query	MR		Tez	
	Total Tasks	Ratio	Total Tasks	Ratio
Q12	40.000	0.081	55.000	26.500
Q21	25.000	0.250	31.000	9.333
Q50	187.000	0.889	110.000	0.048
Q71	176.000	0.023	172.000	33.400
Q85	97.000	0.732	83.000	2.773




Task Distribution for Q21
MR
Tez
# of Tasks
35
30
25
20
15
10
5
0
0-10000
10000-20000
20000-30000
30000-40000
40000-50000
50000-60000
60000-70000
70000-80000
80000-90000
90000-100000
100000-110000
110000-120000
120000-130000
130000-140000
140000-150000


Task Distribution for Q50
MR
Tez
# of Tasks
180
160
140
120
100
80
60
40
20
0
0-10000
10000-20000
20000-30000
30000-40000
40000-50000
50000-60000
60000-70000
70000-80000
80000-90000
90000-100000
100000-110000
110000-120000
120000-130000
130000-140000
140000-150000



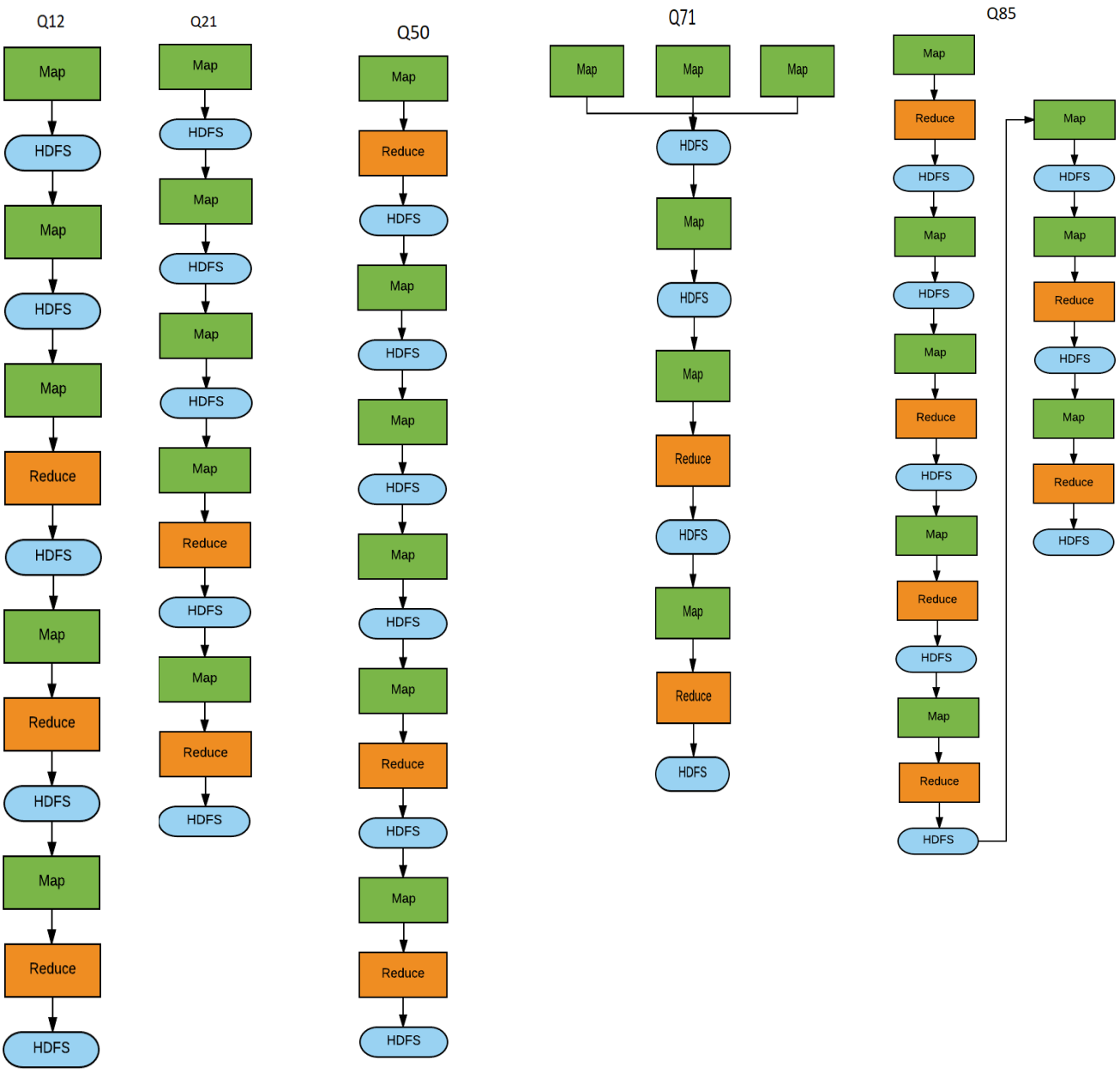


Observations:

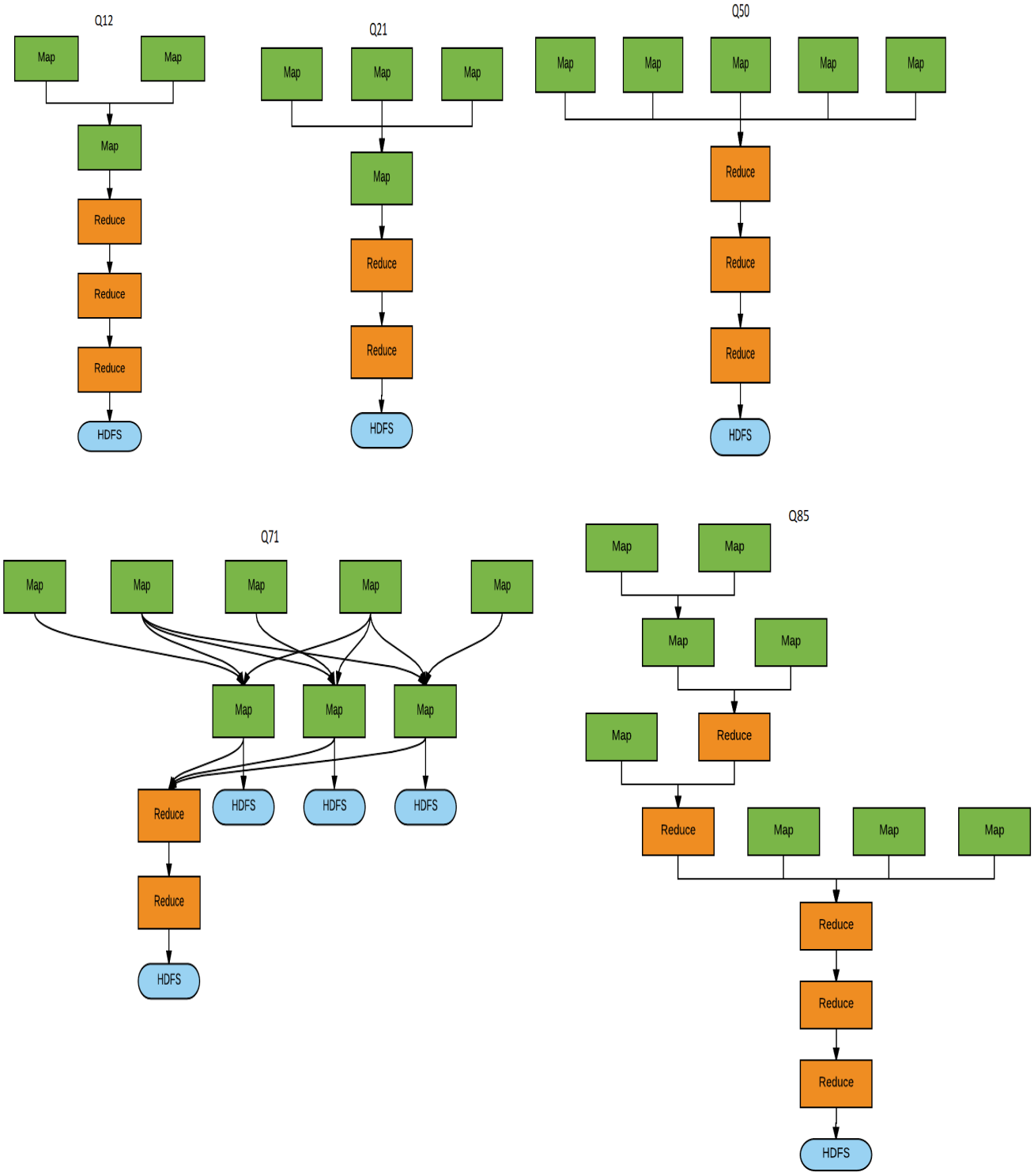
Tez reads all the data in start and hence it starts latter than MR. Task distribution is even in Tez as each stage reads output directly from previous stage and hence does not wait for other parallel processes to finish. Hence there are no humps in task distribution for Tez compare to MR.

Question 1.d:

Map Reduce:

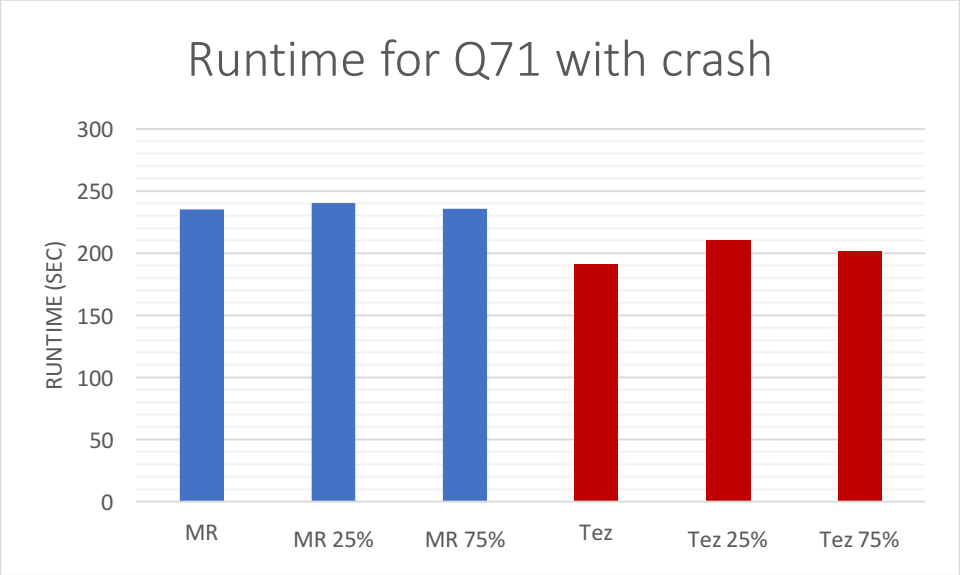


Hive/Tez:


Q12
Map
Map
Map
Reduce
Reduce
Reduce
HDFS
Q21
Map
Map
Map
Map
Reduce
Reduce
Reduce
HDFS
Q50
Map
Map
Map
Map
Map
Reduce
Reduce
Reduce
HDFS
Q71
Map
Map
Map
Map
Map
Map
Map
Map
Reduce
HDFS
HDFS
HDFS
Reduce
Reduce
HDFS
Q85
Map
Map
Map
Map
Map
Reduce
Reduce
Map
Map
Map
Reduce
Reduce
Reduce
HDFS

Question 2:

Query	Runtime (Sec)					
	MR	MR 25%	MR 75%	Tez	Tez 25%	Tez 75%
Q71	235.247	240.31	235.59	191.61	210.28	201.63



Killing a data node will cause loss of data and will kill the processes that are running on that node. Tez and MR will rerun the jobs associated with that data node. This process in turn would generate overhead and would delay the entire processing of the query. As is evident from the graph above kill of a data node cause increase in query runtime by several seconds.

PART C

Question 1: PageRank without any custom partitioning

The key to maximizing cluster resource utilization is increasing parallelism within a stage. Parallelism inside a stage can be increased by increasing the number of partitions of data that can be acted upon by executors. In the cluster that we have been assigned, there are **five** workers, each with **four** cores. Thus, there are 20 execution engines that can be run in parallel. So, we should have at least 20 partitions of the data and we should ensure that at each stage the number of tasks is >= 20 and being executed in parallel.

```
# Initialize the spark context.
sc = SparkContext(conf = conf)

# Load the input file containing URL pairs and remove all lines containing comments
lines = sc.textFile(fileName, numPartitions).filter(lambda line: "#" not in line)
```

Optimization 1: We experimented with the value of numPartitions to ensure that cluster resources are utilized to the full.

	numPartitions = 2	*numPartitions = 20*
Time Taken	12 mins	2.1 mins
Number of Stages	23	23
Number of Tasks	286	2860

This increased parallelism and reduced the execution time almost 6 times. However, upon close inspection we found that this does not ensure that cluster resources are fully utilized. If we look closely at the number of tasks / executor, we can see that the distribution is not uniform.

Executors

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write
4	10.254.0.85:41484	Active	0	0.0 B / 366.3 MB	0.0 B	4	0	0	702	702	6.5 m (10.8 s)	255.1 KB	519.2 MB	587.4 MB
3	10.254.0.82:58347	Active	0	0.0 B / 366.3 MB	0.0 B	4	0	0	742	742	5.4 m (9.5 s)	255.0 KB	480.8 MB	449.5 MB
2	10.254.0.84:33743	Active	0	0.0 B / 366.3 MB	0.0 B	4	0	0	711	711	5.7 m (8.7 s)	255.4 KB	494.1 MB	468.4 MB
1	10.254.0.86:44658	Active	0	0.0 B / 366.3 MB	0.0 B	4	0	0	8	8	50.5 s (2.7 s)	191.3 KB	18.6 MB	56.5 MB
0	10.254.0.83:60054	Active	0	0.0 B / 366.3 MB	0.0 B	4	0	0	697	697	6.2 m (10.9 s)	255.2 KB	496.3 MB	536.6 MB

Optimization 2: The above behavior happens due to Sparks inclination towards running code locally first and waiting for a default period before trying to attempt execution on a non-local node. To prevent this, we set “spark.locality.wait” to 0

Executors

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write
4	10.254.0.85:33005	Active	0	0.0 B / 366.3 MB	0.0 B	4	0	0	756	756	5.6 m (9.6 s)	255.2 KB	464.8 MB	447.4 MB
3	10.254.0.82:55399	Active	0	0.0 B / 366.3 MB	0.0 B	4	0	0	531	531	5.1 m (9.4 s)	255.0 KB	395.8 MB	410.3 MB
2	10.254.0.84:44567	Active	0	0.0 B / 366.3 MB	0.0 B	4	0	0	531	531	5.1 m (7.7 s)	255.1 KB	402.8 MB	414.6 MB
1	10.254.0.86:55415	Active	0	0.0 B / 366.3 MB	0.0 B	4	0	0	481	481	5.2 m (7.7 s)	191.3 KB	366.5 MB	406.0 MB
0	10.254.0.83:49262	Active	0	0.0 B / 366.3 MB	0.0 B	4	0	0	561	561	5.1 m (9.1 s)	255.4 KB	406.9 MB	420.1 MB

The figure above shows how setting wait time to 0 helps distribute the task more fairly across the resources available. The running time also improved, as detailed below:

Num. Partitions	*Spark.locality.wait*	*Running Time*
20	Default	2.1 mins
40	Default	3.3 mins
20	0	1.7 mins
40	0	1.6 mins

In summary, to maximize resource utilization, we

1) increased the number of partitions to be >= 20 and
2) set ‘spark.locality.wait’ time to be 0.

Question 2: Page Rank with Custom Partitioning

Optimization 1:

In part 1, we were already partitioning the URLs using the default hash partitioning. But the ranks were not partitioned using the same hash function. This led to shuffle for joins in each iteration. One optimization we did was to ensure that ranks are partitioned in the same manner as URLs. This can be achieved using "mapValues" method instead of the "map" method which we were using earlier. This change reduced the number of tasks and the total number of stages since joins could now be done in parallel inside each partition without any shuffle.

```
# Initialize ranks for all URLs
ranks = links.map(lambda urls: (urls[0], 1.0))
```

→

```
# Initialize ranks for all URLs
ranks = links.mapValues(lambda urls: 1.0)
```

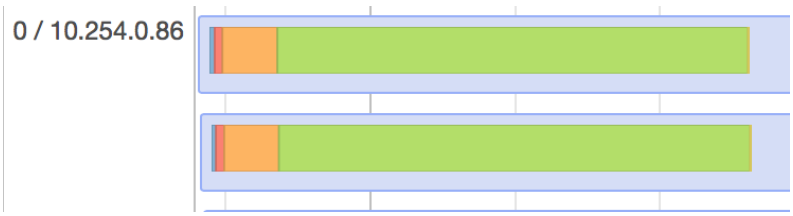
The performance gain is detailed below

	20 partitions of URLs and ranks partitioned independently	*20 partitions of URLs and ranks partitioned per URLs*
Time Taken	1.7 mins	1.2 mins
Number of stages	23	13
Number of tasks	2860	260

Discussion on other attempts:

We were pondering over having a custom partitioner for URLs instead of the default hash partitioned. The basic idea was to find connected components in the graph and to put each component in one partition. So, we inspected the graph looking for fully connected components. We used "Snap" library to inspect the graph. We found one component of size 654782 ~ 95% of nodes and many small components that comprised the rest 5%. Since one component is so big it does not make sense to have it in one partition since it will make the data highly skewed.

We also noticed that shuffle read time is small compared to the computation time, so custom partitioning does not seem to provide much benefit.



Question 3: Persist RDD

The RDD which gets re-used is the one containing the adjacency matrix of node to neighbors. Thus, it makes sense to persist it. The size of the RDD is small enough to be cached in memory

```
# Map [url1, url2], [url1, url3] to [url1, [url2, url3]]
links = links.distinct().groupByKey().persist(StorageLevel.MEMORY_AND_DISK)
```

Data Distribution on 6 Executors

Host	Memory Usage
10.254.0.84:33070	16.3 MB (350.0 MB Remaining)
10.254.0.86:54249	0.0 B (366.3 MB Remaining)
10.254.0.86:60486	16.6 MB (349.7 MB Remaining)
10.254.0.85:57046	16.6 MB (349.7 MB Remaining)
10.254.0.83:34091	16.5 MB (349.7 MB Remaining)
10.254.0.82:36990	16.5 MB (349.8 MB Remaining)

The data gets cached uniformly across the five machines.

	20 partitions without caching	*20 partitions with caching*
Time Taken	1.2 mins	1 min
Number of stages	13	13
Number of tasks	260	260

1) Caching has no impact on the number of stages or tasks (as expected) but helps bring down the amount of data shuffle and execution time because the execution engine does not have to recalculate the adjacency matrix RDD for each iteration.
2) We tried caching “ranks” after each computation but that had no impact on the runtime.

Summarizing Q1 to Q3

	No custom partitioning	*Custom Partitioning of ranks*	*Persistence*
Time Taken	1.7 mins	1.2 mins	1 min
Network Read	2.3 GB	1.91 GB	813 MB
Network Write	2.34 GB	1.95 GB	813.55 MB
Storage Read	337 MB / 5.35 MBps	322 MB / 5.79 MBps	323 MB / 5.65 MBps
Storage Write	1683 MB / 2.79 MBps	481 MB / 2.97 MBps	451 MB / 3.06 MBps
Number of Stages	23	13	13
Number of Tasks	2860	260	260

The network utilization goes down when partitioning ranks and links using the same hashing and it reduces drastically when caching links.

Question 4: Increase number of partitions

These experiments were performed with spark.locality.wait = 0 and ranks = links.mapValues()

	20 partitions	*200 partitions*	*300 partitions*	*1000 partitions*
Time Taken	1.2 mins	1.4 mins	1.4 mins	3.6 mins
Number of Stages	13	13	13	13
Number of Tasks	260	2600	3900	13000

Increasing the number of partitions does not always help because the shuffle time starts dominating the computation time inside each task. We can see with 1000 partitions that the time taken gets tripled. A close look shows how the shuffle time is impacting total performance. Also, the number of cores is limited to 20 so we cannot achieve high parallelism even if we partition the data into many chunks.

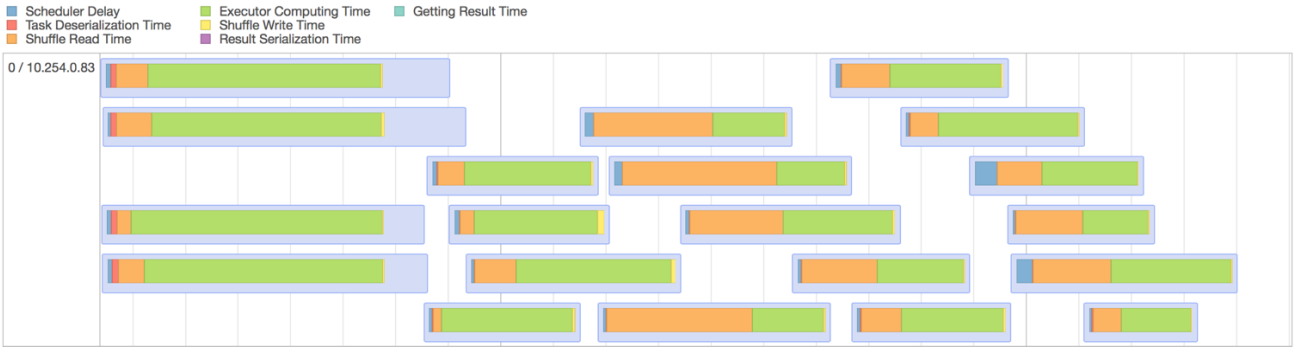


Figure 1 Gantt Plot for time taken by a task

Question 5: Lineage Graph

The lineage graph for Q3 (with persist) is as follows-

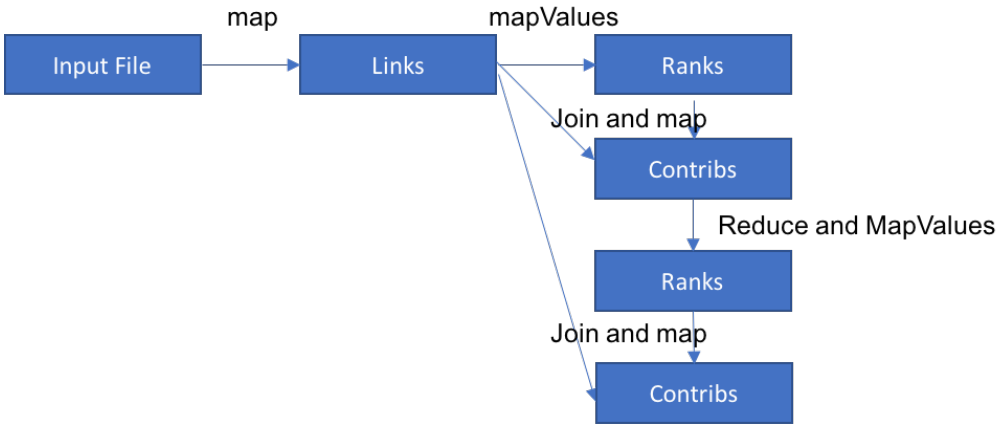


Figure 2 Lineage graph for Q3

The lineage graph for Q1/Q2 is as follows-

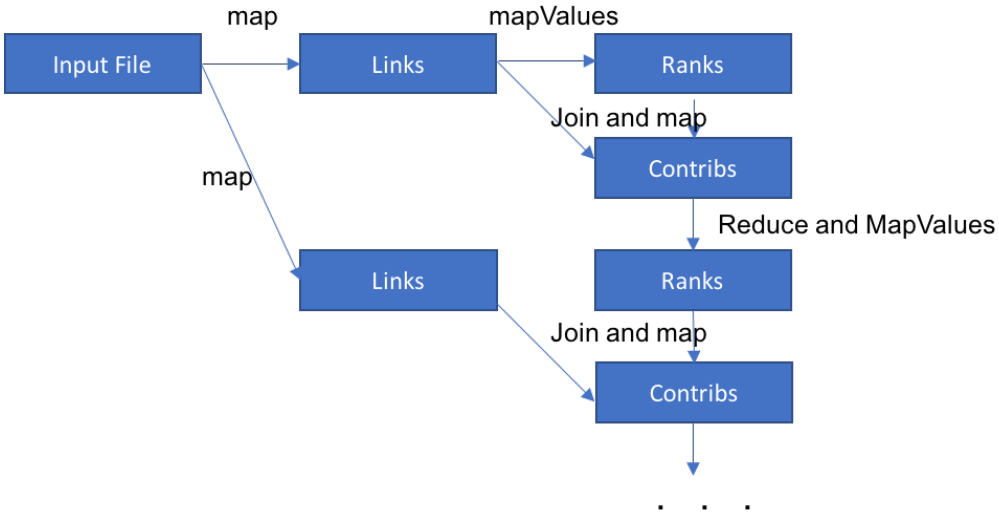
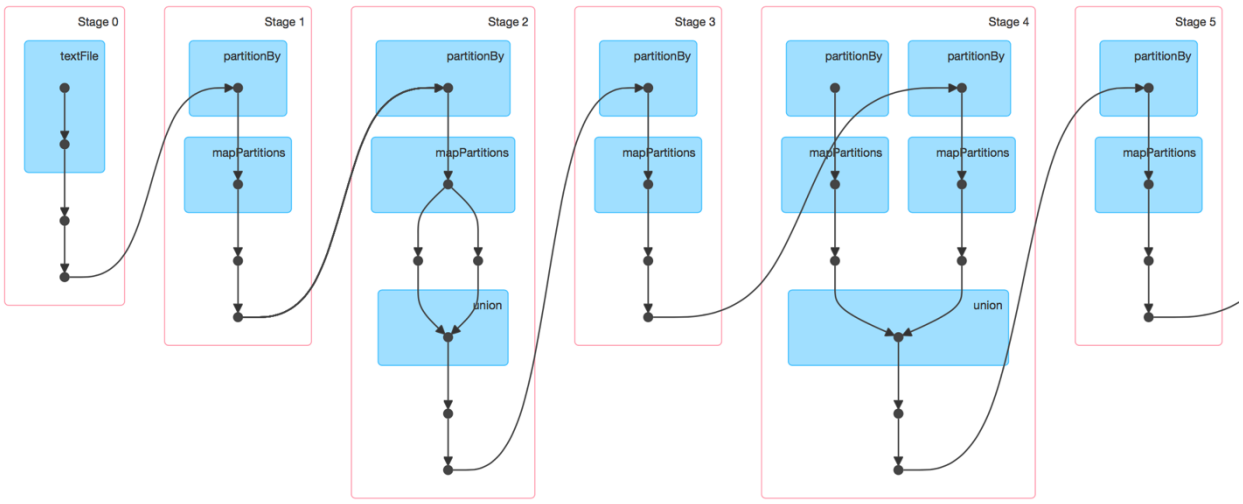


Figure 3 Lineage graph for Q1/Q2

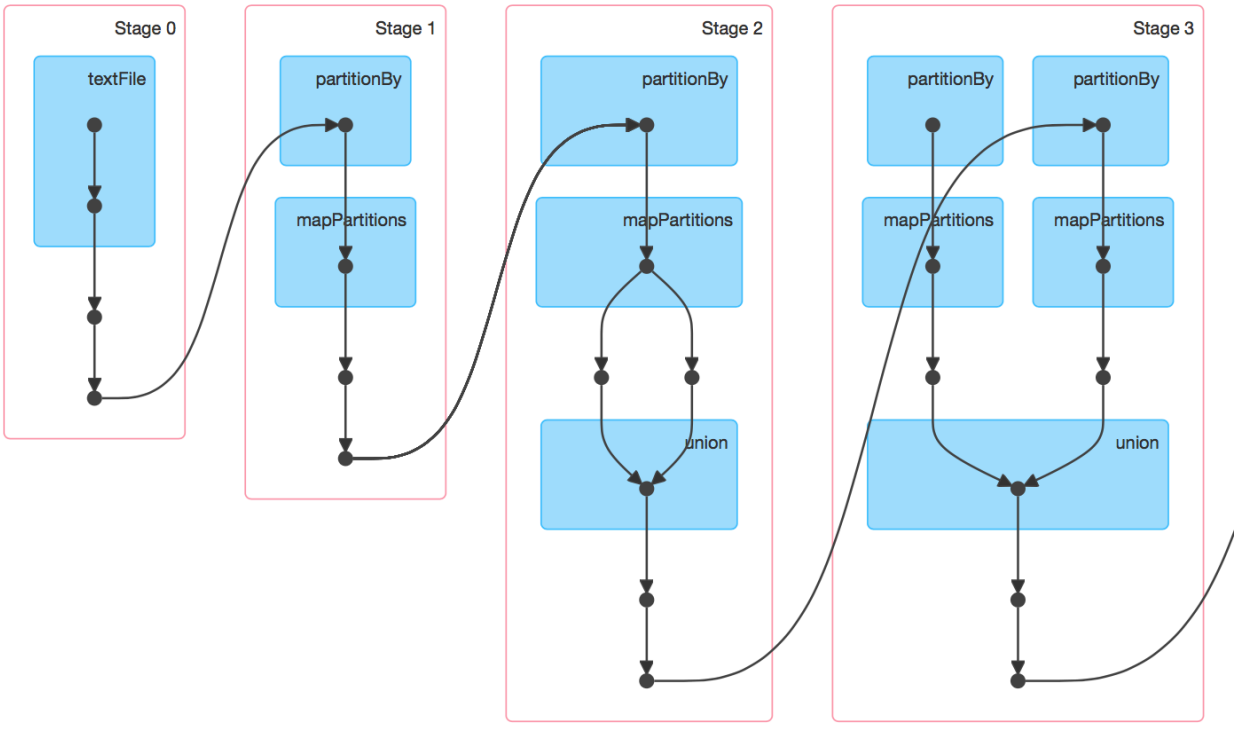
The difference between the lineage graphs of Q3 and the previous parts is that the links RDD need not be recreated in each iteration for Q3.

Question 6: DAG of the application developed

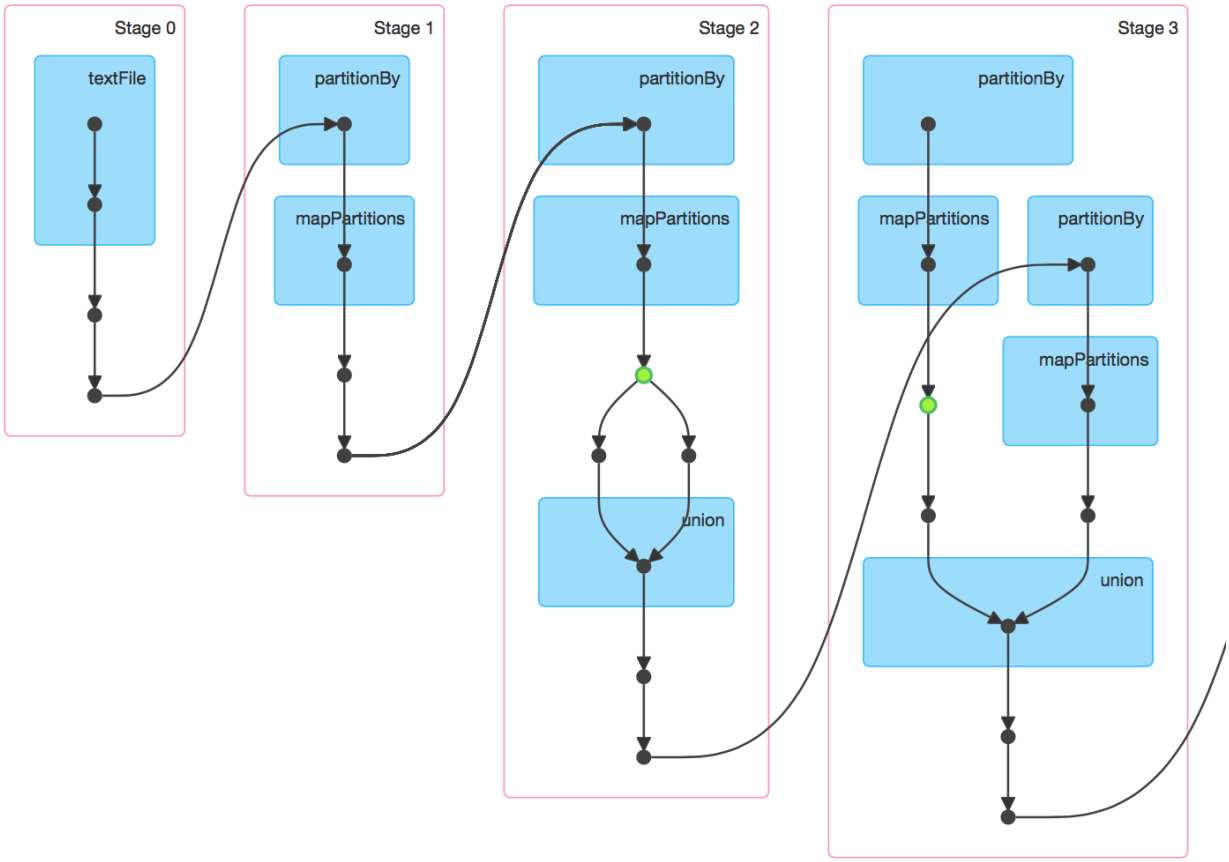
1) DAG for no custom partitioning



2) DAG when ranks and URLs are partitioned in the same manner



3) DAG with persistence



The basic difference between 1 and 2 is that in 1, join needs to be explicitly calculated by shuffling data across partitions whereas in 2, join can be calculated in parallel inside each cluster since ranks and links are partitioned in the same manner. This saves a lot of time. The difference between 2 and 3 is the green dots which mean that the corresponding RDD (for links) is cached and can be used directly so we save time on re-computation.

Question 7: Effect of clearing cache and worker failing

	Normal Execution	Worker fails at 25%	Worker fails at 75%	Cache cleared at 25%	Cache cleared at 75%
Q1	1.7 mins	2.2 mins	2.4 mins	1.7 mins	1.7 mins
Q3	1 min	1.4 mins	1.6 mins	1 mins	1.1 mins

1) If a worker fails late in the computation, we need to redo a lot of computation whose results were stored in that worker. This triggers a lot of re-computations. If a worker fails earlier, we need to do fewer re-computations but we also have one less resource for future computations. So, there is a trade-off involved and the actual impact of late v/s early failing would depend on the application at hand. In the context of PageRank, we found that late failing had a slightly worse impact on the execution time than early failing.

2) We could not find any significant impact of cache clearing under both the scenarios. Maybe the re-computation is fast enough to not make any difference.