Part A:

Performance Metrics

Ques Num	Completion time	Network read BW (MB)	Network write BW (MB)	Storage read/ write BW	Number of tasks
Q1	1m 54s	1,696	1,914	0/0	1,716
Q2	1m 54s	1,696	1,914	0/0	1,716
Q3	1m 24s	795	1,931	0/0	2,288

Table 1

Question 1:

We are running the spark application on a cluster with 5 VMs, where we have one master VM and 4 slave VMs. We are not using master VM to run slave processes. Each VM has an executor configured to use 4 cores and 1GB of memory.

To ensure that cluster resources are being efficiently, we looked into the application log statistics using the history server. Here we see that:

a) All executors are getting equal number of tasks to run. Also, the task run time on each executor is nearly same. This implies that work was distributed uniformly.
b) Network read/write bandwidth and memory usage is also nearly same for all executors.

Above observations led us to conclude that resources are being used efficiently for default run without specifying partitioning of RDD and parallelism.

Furthermore, we varied the no. of partitions to 16, 32, 64, 100, 128, 200, 256, 300 and observed following runtimes:

Number of Partitions	RunTime
2	13m
16	1m 54s
32	3m
64	2m 48s
100	4m
128	3m 12s
200	3m 54s
256	3m 30s
300	3m 24s

Table 2

From the table, we can see that the best performance comes when number of partitions are 16. One point to note is that there total 16 cores in our cluster. Hence, in this case, the work distribution is equal.

However, for other cases with high run times (e.g. 64, 128 number of partitions), work is not getting equally distributed with some VMs running 40 times more tasks than other other VMs. Hence the performance is bad.

Question 2:

In Spark, to optimize the communication in an operation like join, we can do custom partitioning of RDDs. For example, in the PageRank algorithm, the RDD partitioning of links and ranks should be kept same to ensure that the join operation between the links and the ranks requires no remote communication (as each URL's rank will be on the same machine as its link list).
In the spark programming model, the map operation by default does Hash Partitioning of the data set and preserves the partitioning order of Parent RDD.
Thus, in the following code snippet from our application code, both links and ranks RDD would have same partitioning

```
# Loads all URLs from input file and initialize their neighbors.
links = lines.map(lambda urls: parseNeighbors(urls) ).distinct().groupByKey().cache()

# Loads all URLs with other URL(s) link to from input file and initialize ranks of them to one.
ranks = links.map(lambda url_neighbors: (url_neighbors[0], 1.0))
```

This will ensure that join in Contrib calculation has no remote communication. This got substantiated by the locality of "NODE_LOCAL/PROCESS_LOCAL" observed at all join stages of Job execution. Hence, the default partitioner works well for our case.

Furthermore, during the Contrib calculation, essentially the keys are going to change as explained below:

E.g. key	value
1	[2, 3, 5]

After contrib calculation, new RDD will look as follows:

Key	value
2	contribution from 1
3	contribution from 1
5	contribution from 1

However, the following reduceByKey on contribs RDD uses the default practitioner again to bring back the ranks and links to same partitioned form. This leads to optimized communication at every iteration of join operation.

Question 3:

In the PageRank algorithm, only links RDD is reused across iterations. Ranks RDD gets recalculated on every iteration. Therefore, we persisted the links RDD. This led to 26% improvement in the overall completion time.

Question 4:

From Table 2, we observe that performance is very poor when partition count is 2. This is because there are only 2 tasks per stage and only 2 VMs are getting used. That said, increasing the number of partitions beyond the number of cores in the cluster will not necessarily improve the performance because the parallelism is eventually limited by the cluster size (total number of cores in all VMs).

Question 5:

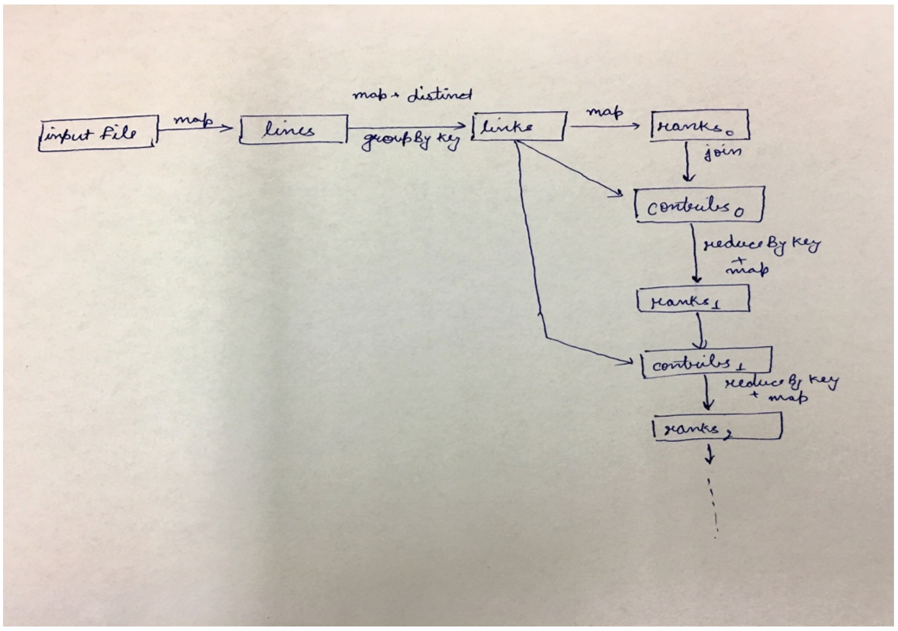


Figure 1: Lineage graph for PageRank algorithm

The lineage graph observed will be same across applications. This is because lineage just captures the transformations on the data so that we can replicate a partition. We do not change the transformations applied in questions 1, 2 and 3. Persist and custom partitioning are for performance optimizations.

Question 6:

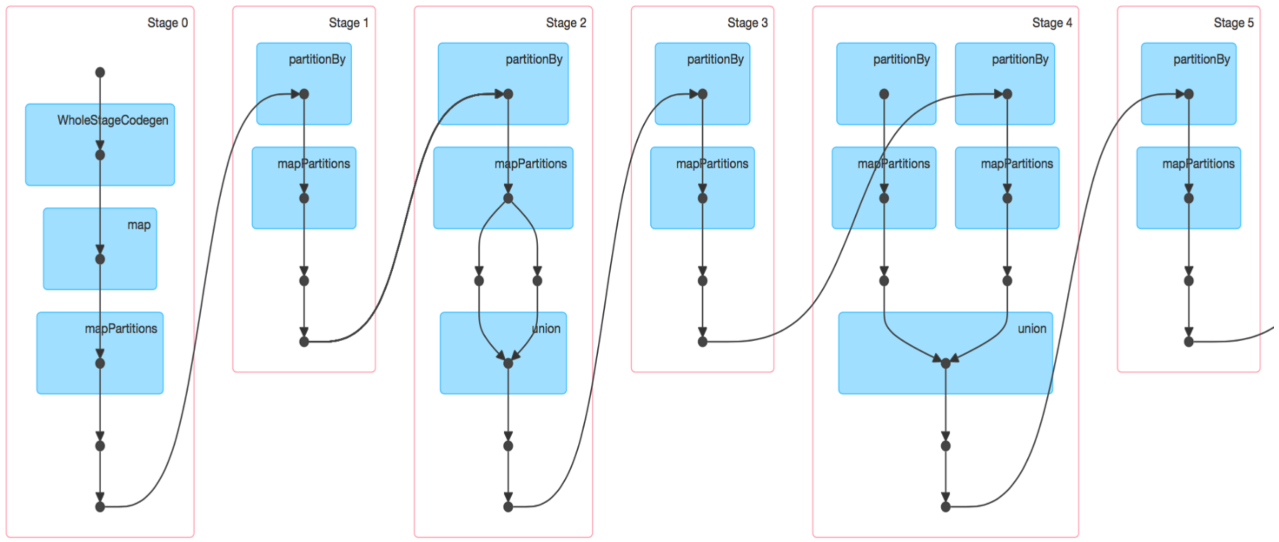


Figure 2: Stage-Level DAG for Question 1

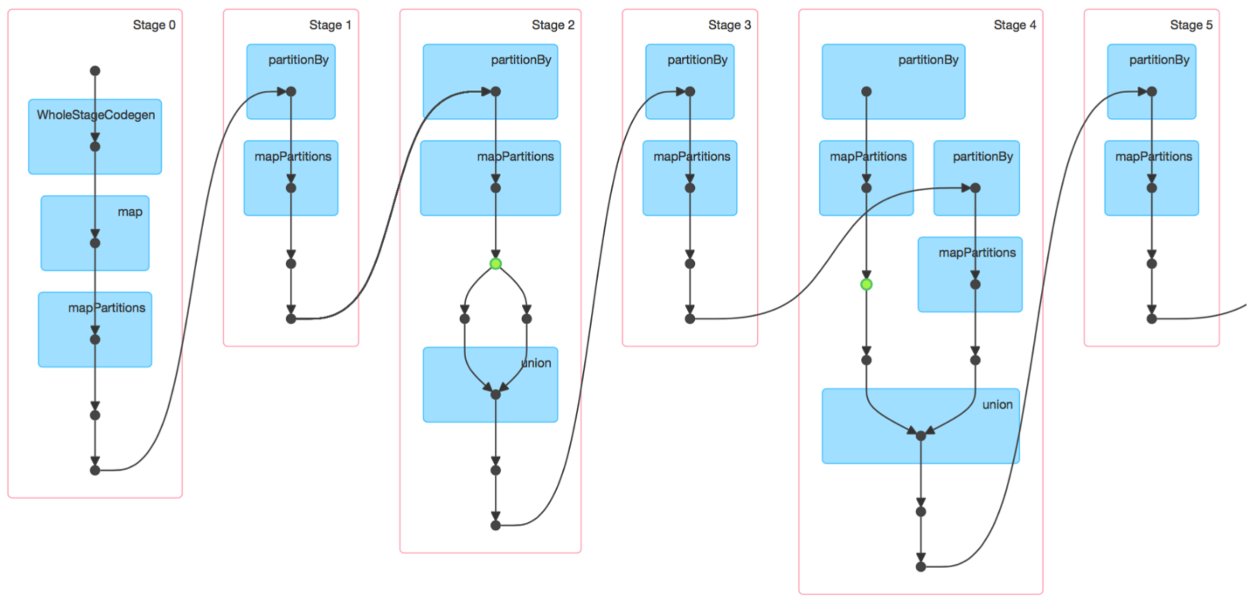
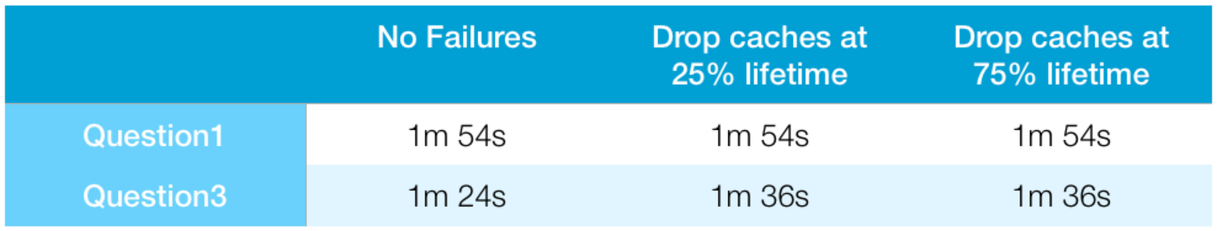


Figure 3: Stage-Level DAG for Question 3

The stage level DAG structure is almost same (i.e. the number of stages and edges) for applications developed in Question 1 and 3 visually. The only difference being now in Stages 2, 4, 6 etc, the join operation will pick persisted links RDD. In spark, each stage in DAG contains as many pipelined transformations with narrow dependencies as possible. The boundaries of the stages are the shuffle operations required for wide dependencies, or any computed partitions that can short circuit the computation of a parent RDD. Since in our algorithm, persist is not reducing wide dependencies, there is no change in DAG structure.
However, in general DAG has a crucial impact on performance. This is because, more number of stages implies more wide dependencies, leading to higher runtimes.

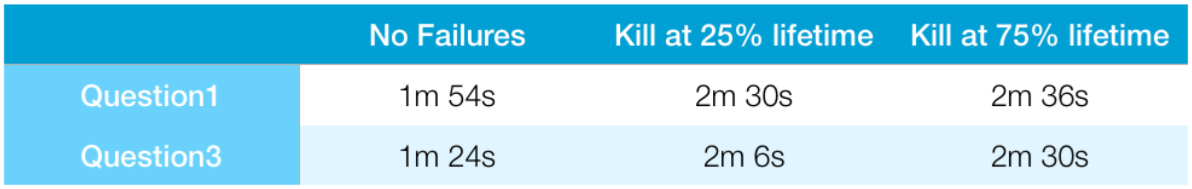
Question 7:



	No Failures	Drop caches at 25% lifetime	Drop caches at 75% lifetime
Question1	1m 54s	1m 54s	1m 54s
Question3	1m 24s	1m 36s	1m 36s

Table 3: Performance analysis with caches dropped in one worker VM

a) When persist is disabled (Question 1), as expected dropping caches during the run has no visible impact on performance.
b) When persist is enabled (Question 3), dropping caches during the run increases the run time. No significant difference seen in dropping caches at 25% lifetime vs 75% lifetime.



	No Failures	Kill at 25% lifetime	Kill at 75% lifetime
Question1	1m 54s	2m 30s	2m 36s
Question3	1m 24s	2m 6s	2m 30s

Table 4: Performance analysis with worker VM killed during the run

a) Killing a worker VM degrades the performance in both the applications (with and without persist). This is because the application now has to complete with just 3 workers instead of 4.

b) Killing the application at 75% increases the runtime more than killing it at 25%. We think this could be because now more work has to be re-done since the RDDs present in the 4th VM now has to recomputed by the other three VMs. However, if you kill at 25% less work has to be re-done because work will get redistributed early on among the three VMs as the killed VM would not have made much progress by then.

Overall, our observation is that we persist for performance improvements, not for failure resilience.