# CS 838: Assignment 2
# Group 27: Deepanker Aggarwal, Saket Saurabh

## Part A - Apache Spark

**Spark cluster configuration:**

- In our Spark cluster, we have a single master and 5 worker nodes. Each node has 4 CPU cores and approximately 20 GB of memory.
- Each Spark application runs with 1 GB of driver memory and spawns 5 executors in total across the cluster, where each executor utilizes 4 CPU cores and 1 GB of memory on each of the worker machines.
- Each task within a Spark executor runs on 1 CPU. Thus, a maximum of 20 tasks can run in parallel in our Spark cluster at any given point of time.

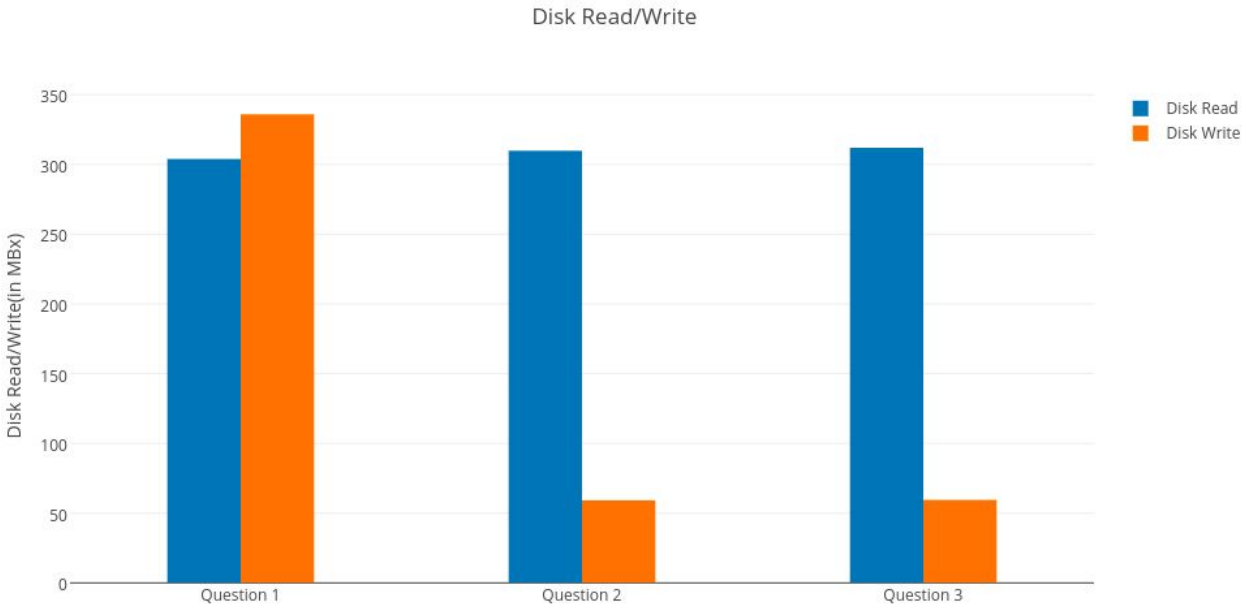
**Application Execution Time:**

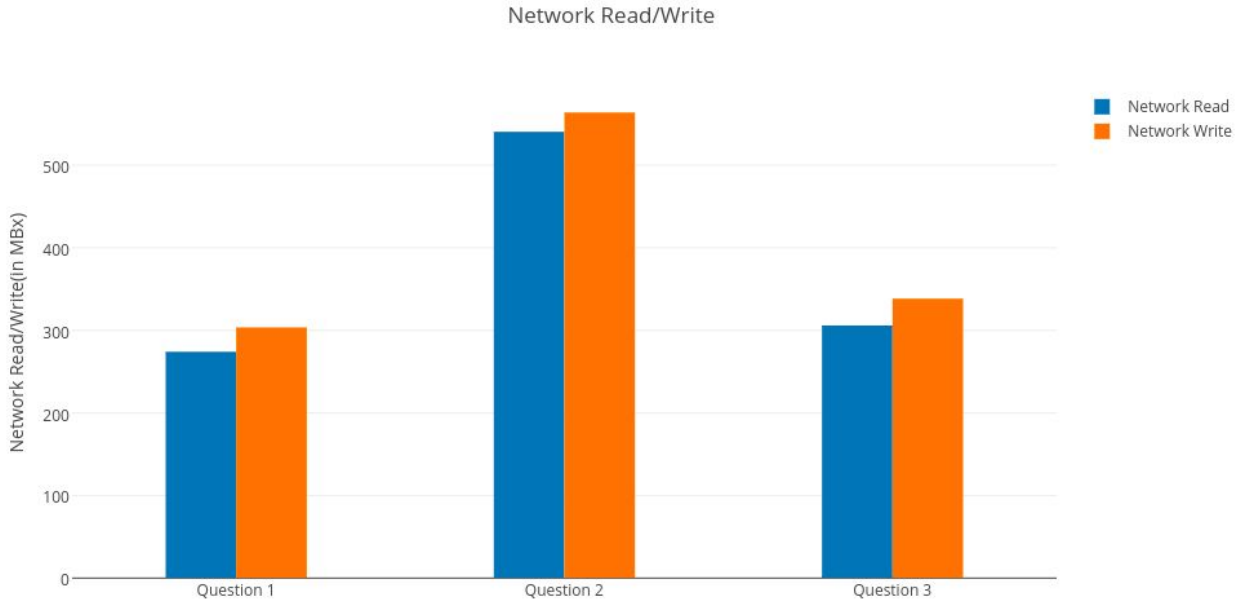
- For the three Spark applications developed in Part-A, we report below the application execution time for each of them. The following application completion time was derived by taking the mean of the total duration (uptime) for multiple runs of the application as reported in the Spark Application History Server UI.
- In our experiments, the number of partitions used for the RDDs is 40.

| | Application Completion Time (in seconds) |
|---|---|
| `CS-838-Assignment2-PartA-Question1` | 58 |
| `CS-838-Assignment2-PartA-Question2` | 47 |
| `CS-838-Assignment2-PartA-Question3` | 37 |

**Amount of disk/network read/write:**

We report below the amount of disk/network read/write for each of the three Spark application developed in Part-A.



**Number of tasks per application:**

We report below the number of tasks per execution for each of the three Spark application developed in Part-A. The number of partitions used for the links, ranks and contribs RDD is 40.

| | Number of tasks |
|---|---|
| `CS-838-Assignment2-PartA-Question1` | 480 |
| `CS-838-Assignment2-PartA-Question2` | 520* |
| `CS-838-Assignment2-PartA-Question3` | 520* |

*The number of tasks for Question-2 and Question-3 is slightly more because each of these application runs an additional small sampling job for the custom range partitioner initialization.

**Explanation for various observed metrics above:**

- *Application Execution Time:*
  - The application execution time improves with each version. This is expected because `CS-838-Assignment2-PartA-Question2` adds support for custom partitioning (as explained below) that improves data locality and data movement across various stages of execution. The `CS-838-Assignment2-PartA-Question3` builds upon the previous Question 2 and adds RDD persistence that even further improves the application execution time (as explained below).
- *Amount of disk read/write:*
  - The amount of disk reads is same for all the three applications, as all the applications read and process an equal amount of input data.
  - The application `CS-838-Assignment2-PartA-Question1` performs a significantly large number of disk writes as compared to the other two applications. This is because the application in Question 1 employs a poor data partitioning strategy that leads to a large number of page evictions to disk swap space when the RDDs cannot fit into allocated 1 GB memory of respective executors.
- *Amount of network read/write:*
  - To improve data locality, the application `CS-838-Assignment2-PartA-Question2` performs data repartitioning of intermediate RDDs a number of times over the course of application execution. And this is why the network read/writes are significantly higher than the application `CS-838-Assignment2-PartA-Question1`
  - However, the same relatively high network read/write trend is not observed for the application `CS-838-Assignment2-PartA-Question3` because this application saves some significant network I/O by caching the 'links' RDD as in-memory objects. But still, the overall network I/O of application in Question 3 remains higher than the application in Question 1.
- *Total number of tasks:*
  - Since the number of partitions determine the number of tasks executed on the cluster, all applications approximately run the same number of tasks, as the number of partitions is same for all the three applications.
  - The number of tasks for Question-2 and Question-3 is slightly more because each of these application runs an additional small sampling job for the custom range partitioner initialization.

## Question 1

To ensure that the cluster is fully utilized, we oversubscribe the cluster resources by creating more number of partitions than the total number of available cores across all executors in the cluster. Since the number of partitions directly translates to number of tasks, by having an appropriate number of partitions we can ensure that the cluster is fully utilized.

The Spark Tuning Guide [link] recommends to have 2-3x tasks per CPU core in the cluster for best performance. We evaluated the application completion times for various number of partitions- 20, 40, 60 & 80 and found that the best performance is obtained for 40. We suspect that setting the number of partitions as 40 gives us the best performance because it achieves the best trade-off between the gains due to cluster resource oversubscription and the loses due to scheduling overhead.

## Question 2

As presented above, the job completion time for the `CS-838-Assignment2-PartA-Question2` application is lower than the `CS-838-Assignment2-PartA-Question1` application. Therefore, certainly the application performance improves by introducing custom partitioning.

We utilized custom partitioning in the following two ways to achieve better performance:

- We used a range partitioner instead of the default hash partitioner to partition the RDDs. Upon manual inspection of the input dataset, we discovered that a given source link number has more neighbors in same value range. For example, the source url 1 has its neighbors as [2, 5, 7, 8, 9, 11, 17, 254913, 438238]. Hence, using a range partitioner reduces the amount of data that has to be shuffled during a join operation between links and ranks RDDs, as the two join keys are likely to be on the same partition. This has also been verified by looking at the cumulative Shuffle Read and Shuffle Write values at Spark Application UI for range partitioning versuses hash partitioning.
- We explicitly specified the range partitioner after the transformations that would make the RDDs lose their custom partitioning information. The ‘map’ and the ‘flatmap’ transformations used in our Spark program would make the RDDs lose the partitioning information of their parent RDDs [link]. Therefore, for example, every time we generate a new contribs RDD for each iteration after a ‘flatmap’ transformation, we re-partition the resulting ranks RDD using the custom range partitioner. This improves the join performance when the ranks RDD is joined with the links RDD in the subsequent iteration.
- Thus, in our case the performance improves by almost 18.97% for Question 2 as compared to Question 1.

## Question 3

As shown above, the job completion time for the `CS-838-Assignment2-PartA-Question3` program is even lower than the job completion time for `CS-838-Assignment2-PartA-Question2` & `CS-838-Assignment2-PartA-Question1` Therefore, certainly there is a benefit of persisting RDDs as in-memory objects.

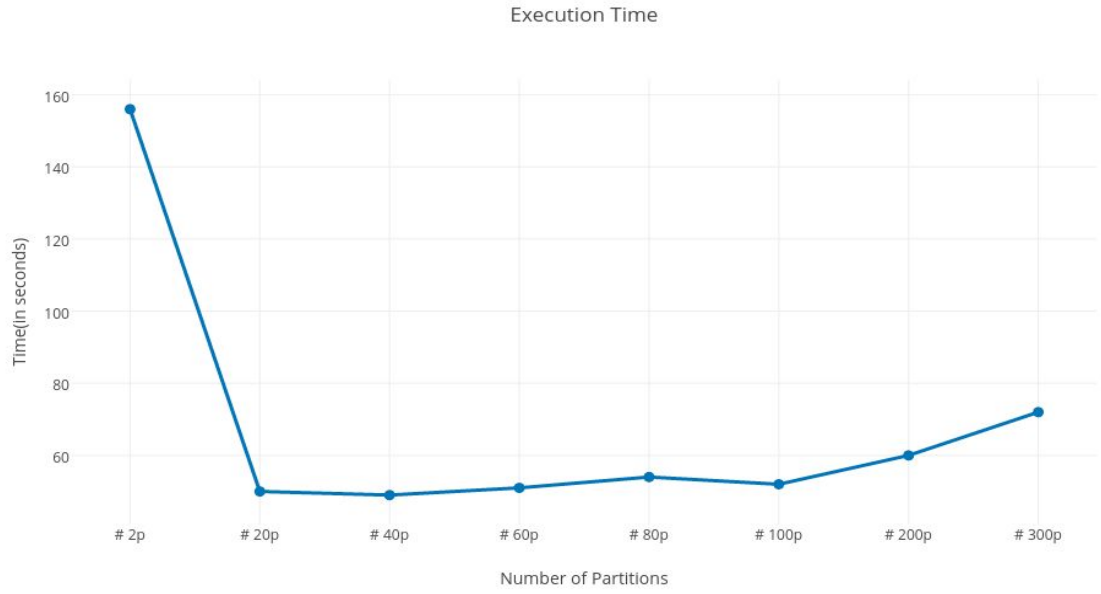
In our implementation, we decided to only persist the links RDD, as the links RDD is repeatedly used in multiple iterations for joining with the ranks RDD. When the links RDD is persisted as an in-memory object, the overall program execution improves because the subsequent iterations do not have to recompute the links RDD. Thus, in our case the performance improves by almost 21.27% for Question 3 as compared to Question 2.

## Question 4

Increasing the number of RDD partitions initially helps to improve the application performance, however, an inappropriate increase may also lead to performance degradation.

Increasing the number of RDD partitions gives the benefit of increased parallelism that can fully utilize the cluster resources. However, this increased parallelism comes at the cost of scheduling overhead. Hence, increasing the number of partitions after a given threshold causes a performance degradation, as scheduling overheads become more pronounced.

Following is a graph that shows the performance for `CS-838-Assignment2-PartA-Question2` application for various number of partitions:




As evident from the graph, the best performance is seen when the number of partitions is 40.

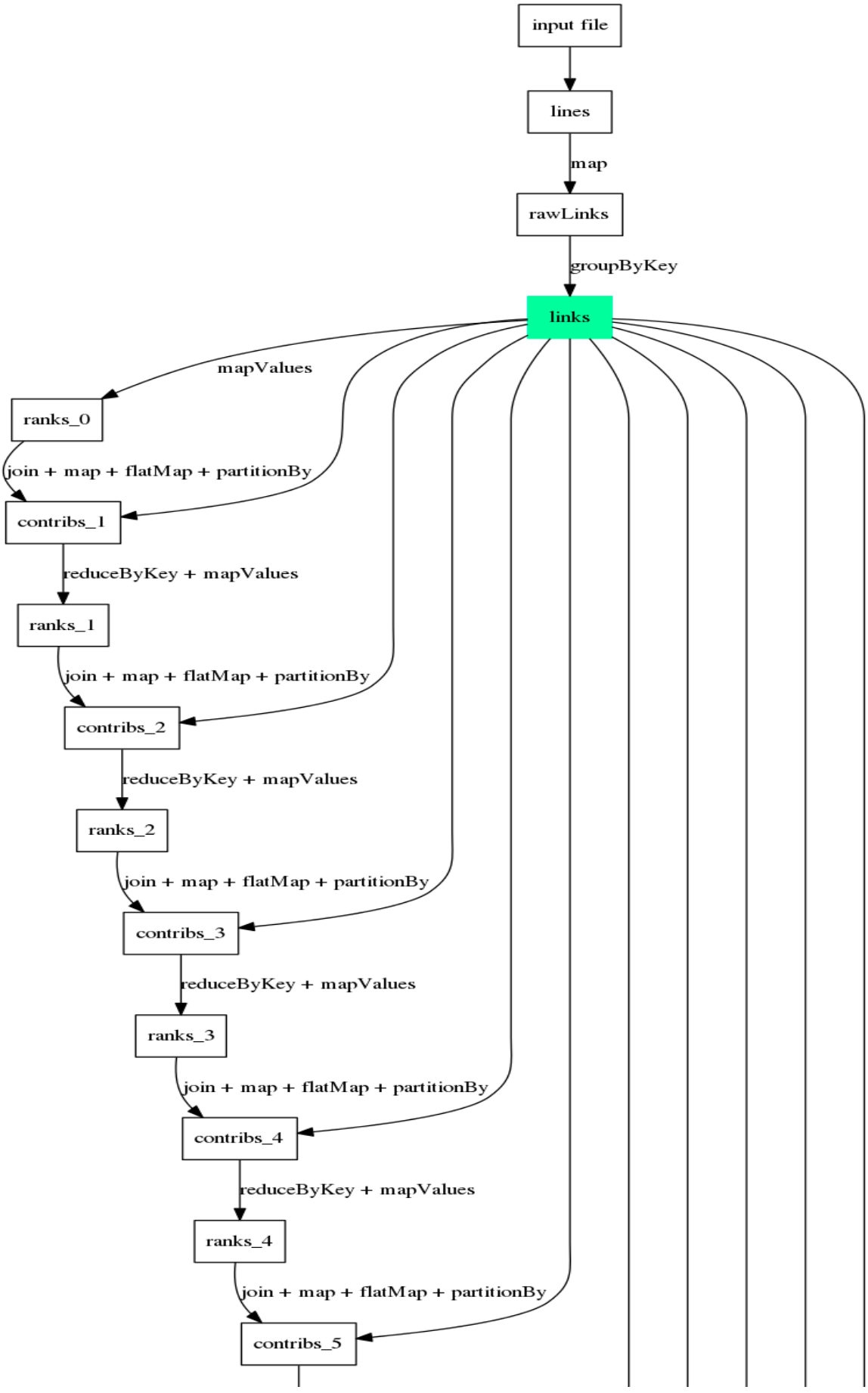
When the number of partitions is 2, we suffer from poor performance due to very low parallelism, whereas for 200 partitions or more the scheduling overheads trumps any gain from increased parallelism. Hence, 200 can be quoted as a number where performance starts to have a huge negative impact with the number of partitions.

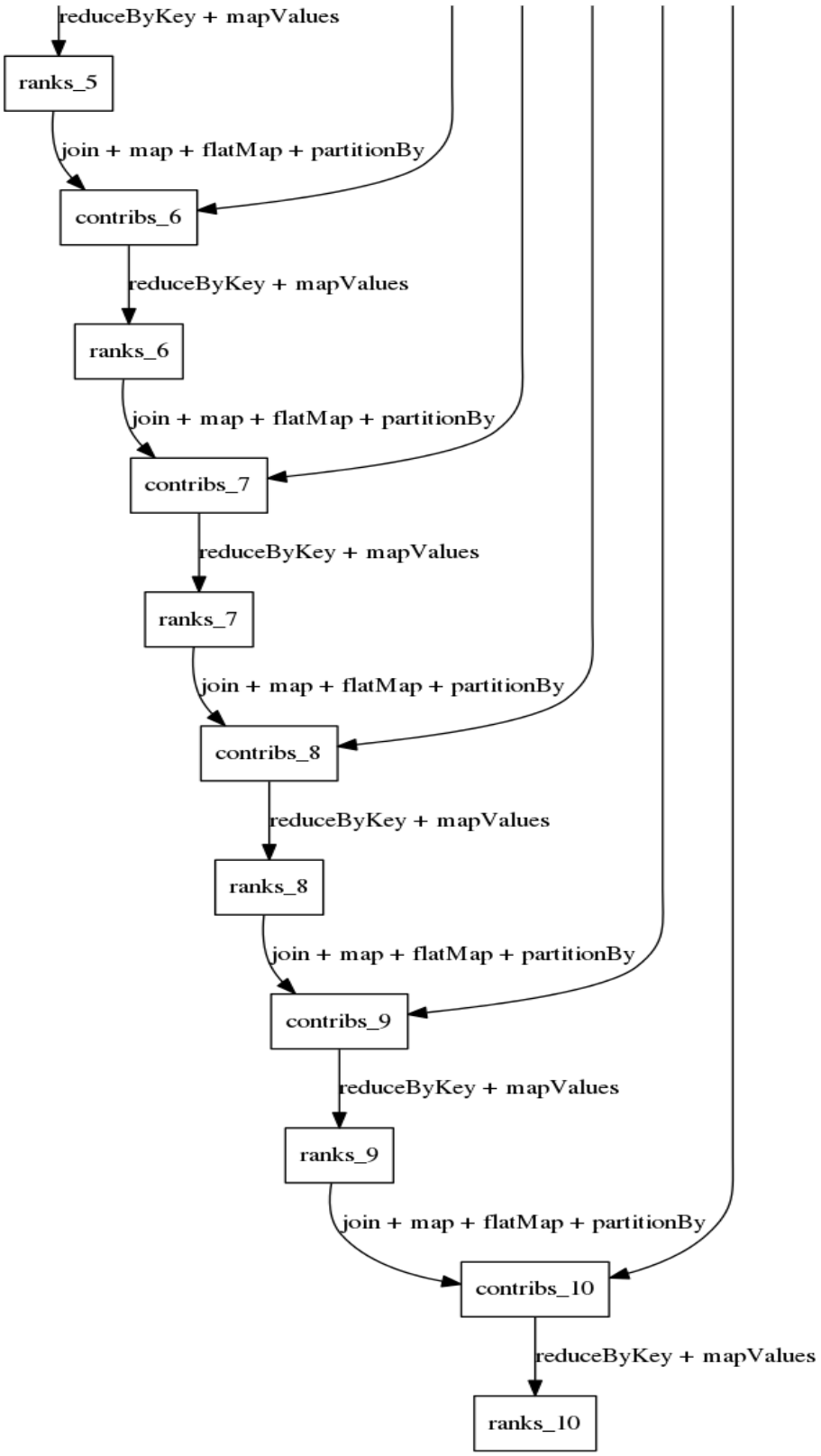
## Question 5

As required, the following shows the lineage graph for the `CS-838-Assignment2-PartA-Question3` application:

The lineage graph observed is the same for all the three applications, except for a few minor differences. For `CS-838-Assignment2-PartA-Question1` the lineage does not have the 'rawLinks' RDD and the 'partitionBy' transformation is also missing (which are introduced due to custom partitioning).

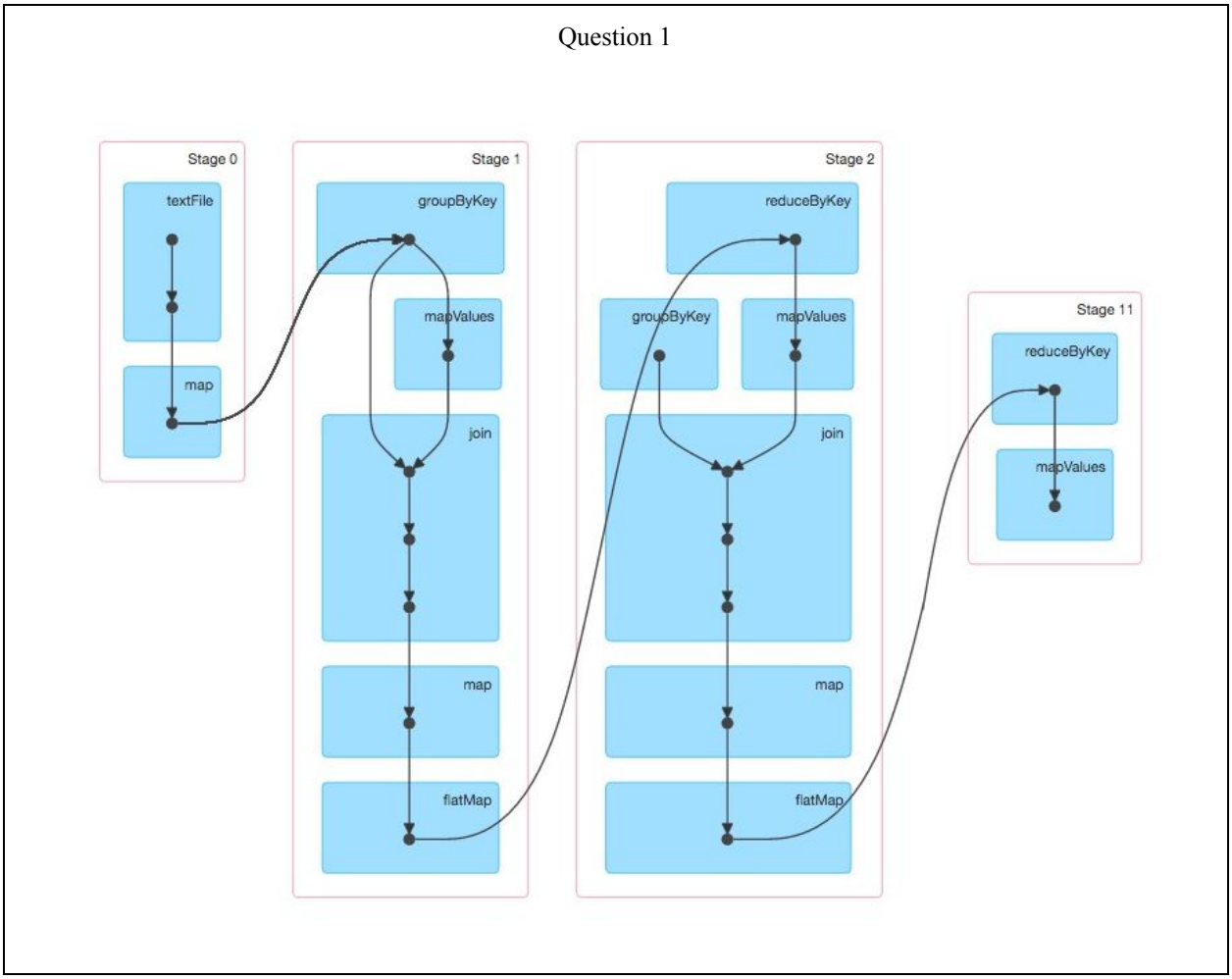
As the 'partitionBy' transformation does not change the dependency between RDDs, but just the type of the dependency (narrow/wide), the lineage observed is the same. Similarly, the only difference between the lineages of `CS-838-Assignment2-PartA-Question2` and `CS-838-Assignment2-PartA-Question3` is that the links RDD is persisted in the case of latter. As caching only affects the recovery of RDDs and helps in the decision making when pages are to be swapped out from the memory, we observe no differences between the lineages in this case too.


input file
lines
map
rawLinks
groupByKey
links
mapValues
ranks_0
join + map + flatMap + partitionBy
contribs_1
reduceByKey + mapValues
ranks_1
join + map + flatMap + partitionBy
contribs_2
reduceByKey + mapValues
ranks_2
join + map + flatMap + partitionBy
contribs_3
reduceByKey + mapValues
ranks_3
join + map + flatMap + partitionBy
contribs_4
reduceByKey + mapValues
ranks_4
join + map + flatMap + partitionBy
contribs_5

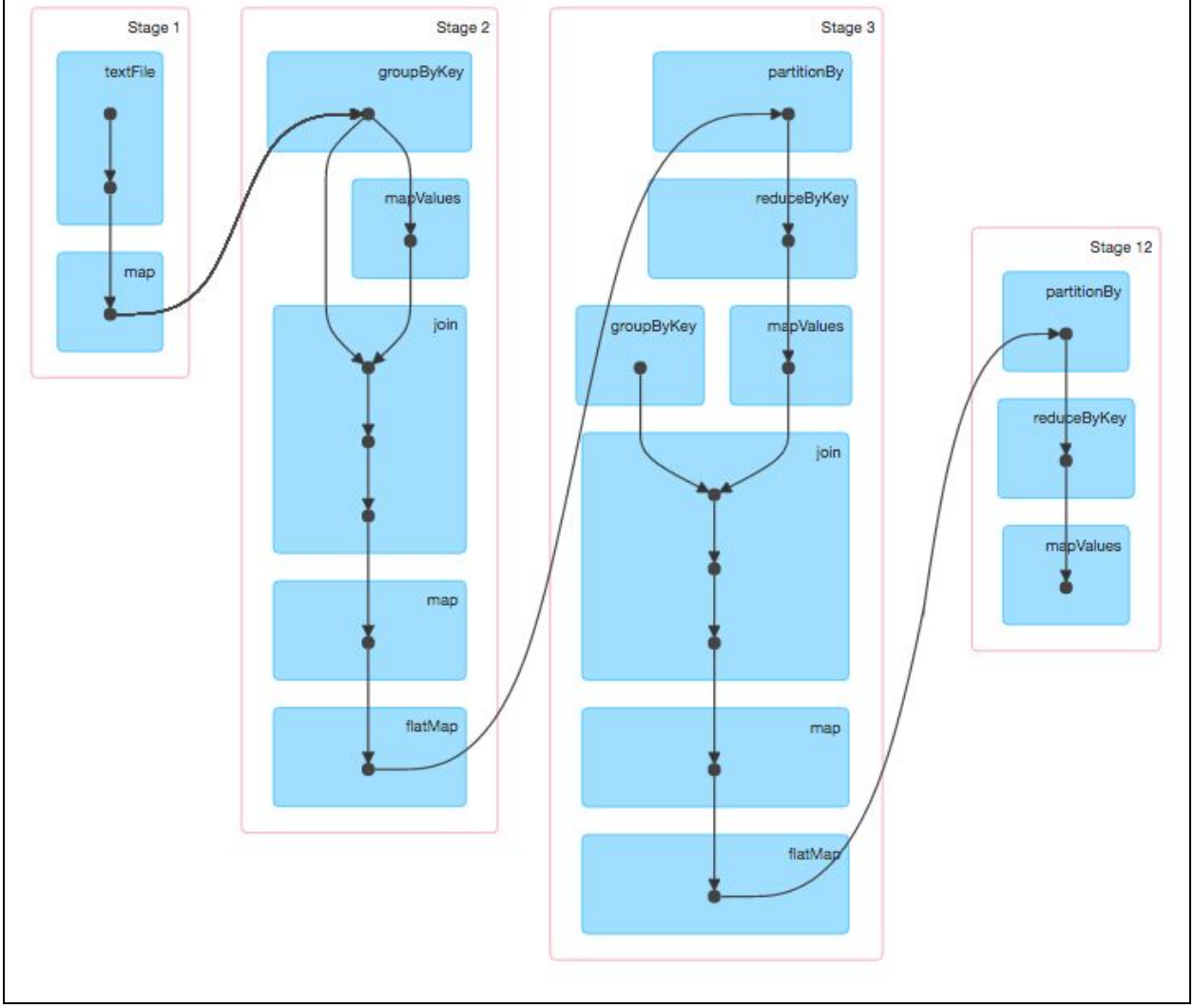

reduceByKey + mapValues
ranks_5
join + map + flatMap + partitionBy
contribs_6
reduceByKey + mapValues
ranks_6
join + map + flatMap + partitionBy
contribs_7
reduceByKey + mapValues
ranks_7
join + map + flatMap + partitionBy
contribs_8
reduceByKey + mapValues
ranks_8
join + map + flatMap + partitionBy
contribs_9
reduceByKey + mapValues
ranks_9
join + map + flatMap + partitionBy
contribs_10
reduceByKey + mapValues
ranks_10

## Question 6

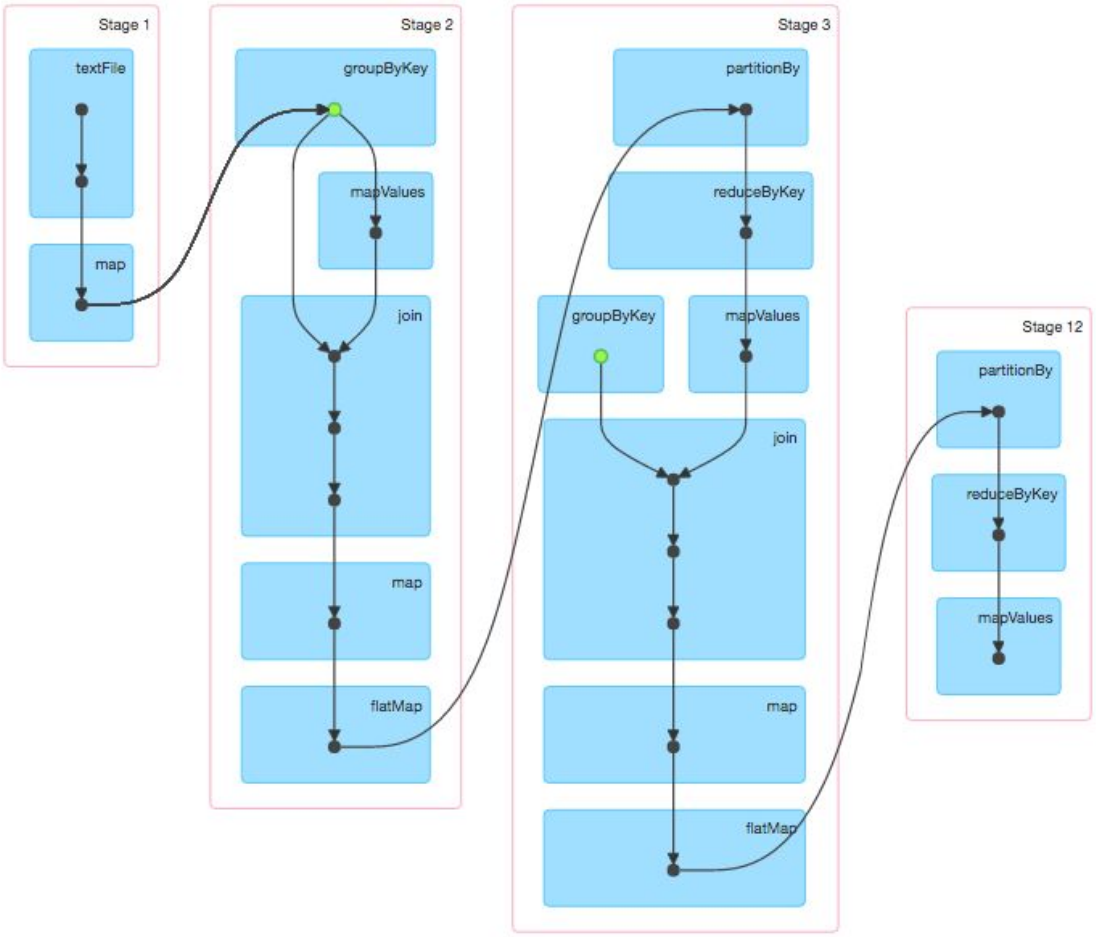
As required, the following shows the DAG for all the three applications:



Question 2


Stage 1
textFile
map
Stage 2
groupByKey
mapValues
join
map
flatMap
Stage 3
partitionBy
reduceByKey
groupByKey
mapValues
join
map
flatMap
Stage 12
partitionBy
reduceByKey
mapValues

Question 3




DAG is almost the same for all the three applications.The noticeable differences are as follows:-

1) The partitionBy transformation is missing in case of `CS-838-Assignment2-PartA-Question1` application, which is there in the case of `CS-838-Assignment2-PartA-Question2` and `CS-838-Assignment2-PartA-Question3` We do a range partitioning using 'partitionBy' in the latter two applications. This reduces the execution time as the links and ranks RDD partitions to be joined are now co-located on same node and network overhead is less.
2) The links RDD is persisted (the green dot in the Stage 2 and the Stage 3 of DAG) in the application DAG for `CS-838-Assignment2-PartA-Question3`.By persisting the links RDD, we ensure that the links RDD is always readily available in memory to be used in subsequent iterations. If the links RDD were not persisted, then it could be evicted to disk if the executor ever runs out of memory. Hence, this improves the application performance in case of Question 3 over Question 2.

## Question 7

Performance of `CS-838-Assignment2-PartA-Question1` application in presence of failures:

*[Execution time for non-failure scenario = 58 s]*

| | Clear cache on a worker | Kill Worker |
|---|---|---|
| Failure at 25% | 63 s | 60 s |
| Failure at 75% | 62 s | 66 s |

Performance of `CS-838-Assignment2-PartA-Question3` application in presence of failures:

*[Execution time for non-failure scenario = 37 s]*

| | Clear cache on a worker | Kill Worker |
|---|---|---|
| Failure at 25% | 42 s | 55 s |
| Failure at 75% | 40 s | 58 s |

*Explanation:*

- For `CS-838-Assignment2-PartA-Question1`killing the worker node at 25% has limited effect on the performance, as the lost work is small enough to be computed quickly by running backup tasks on different executors. However, as more amount of useful work is lost when the worker is killed at 75%, hence it finishes quite later as seen from the execution times.

- For application `CS-838-Assignment2-PartA-Question1` dropping the cache has a significant impact on the execution time. Since the application employs a poor partitioning strategy, the disk swap space is heavily used to page out RDDs which do not fit in the memory (as also evident from the disk write plots above). Hence, clearing the disk page cache leads to a negative I/O overhead of bringing those pages back in the memory from disk, which in turn increases the execution time.

- For `CS-838-Assignment2-PartA-Question3`killing the worker node has a huge negative impact on the performance of this application. From the Spark Application UI, we figured out that about 15s of time was spent in recovering the results of the failed stage. Similarly it spends around 17s to recover the results at 75%. The difference is because the in case of 75% it has do more work than in case of 25%.

- For `CS-838-Assignment2-PartA-Question3` dropping cache doesn't affect the performance as much in this application because this application does not rely much on the usage of the disk swap space (as also evident from the disk write plots above). This effect is due to custom partitioning, which greatly improves the data locality of the RDDs.