# Fall 2016 CS838: Assignment #2

Group 26 : Dongning Wang, Guiming Zhang

## Part A - Apache Spark

**Question 1**. Write a Scala/Python/Java Spark application that implements the PageRank algorithm **without any** custom partitioning (RDDs are not partitioned the same way) or RDD persistence. Your application should utilize the cluster resources to it's full capacity. Explain how did you ensure that the cluster resources are used efficiently. (*Hint: Try looking at how the number of partitions of a RDD play a role in the application performance*)

**Answer:**
We try to utilize the cluster resources to it's full capacity through ***two means***.

***First***, we **set the number of executors as five**. This decision is made based on the following rationale. According to the specification that we should "set executor memory and cores to 1GB and 4 respectively", and based on the fact that our cluster has five VMs each with 4 virtual CPU cores and 20GB memory (the Master VM serves also as a Worker), it makes sense to set the number of executors as five such that all 5 executors * 4 CPU cores per executor = 20 CPU cores are allocated to executors (memory is not of concern because there is more than enough of memory). This makes it possible to utilize the cluster's full capability when executing Spark applications using these executors on the cluster.

***Second***, we investigated **the impact of the number of partitions of the *links RDD* on the application performance** (without any custom partitioning or RDD persistence). We did this by setting the ***numPartitions*** parameter in the ***groupByKey()*** operation on the links RDD. The number of partitions we tested are 2, 5, 10, 20, 50, 100, 200, and 300. Figures 1.1 through 1.4 below show such impact in terms of performance metrics including completion time, amount of data read or written on disk, amount of data received or transmitted through network, and the total number of tasks.

Generally, mount of data read or written on disk, amount of data received or transmitted through network, and the total number of tasks increases when the number of partitions increases. Number of tasks grows linearly with increasing number of partitions. There are lots of data shuffling with larger number of partitions. However, the impact on completion time does not have such a monotonic tread. Actually, the completion time is very long when there were too few partitions (e.g., 2). Then the completion time drops dramatically when the number of partitions increases. But beyond, 50 or so, the number of partitions begins to have a negative impact on completion time. **When the number of partitions equals 20, the application completed fastest (in about 119 s)**.

We looked at how many executors were used in the executions with different number of partitions in the Spark History Server UI. Only four of the five executors were assigned tasks in the executions with 50 or fewer partitions. All five executors were assigned tasks when

there were 100 or more partitions. Nevertheless, when number of partitions equals 20, the applications runs fastest even though only four of the five executors were assigned tasks. **This is probably due to that the benefit of using few executor but with less data shuffling overweights that of using more executors in executing tasks but with more data shuffling.**

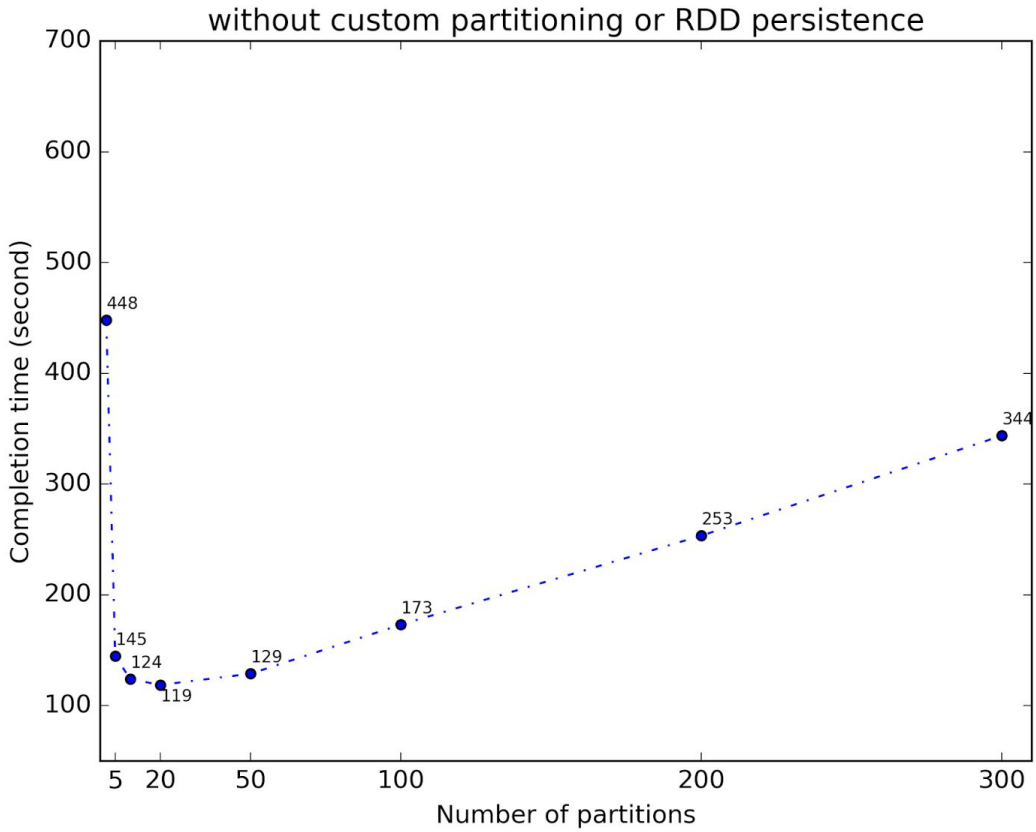



**Figure 1.1**. Impact of number of partitions on completion time.

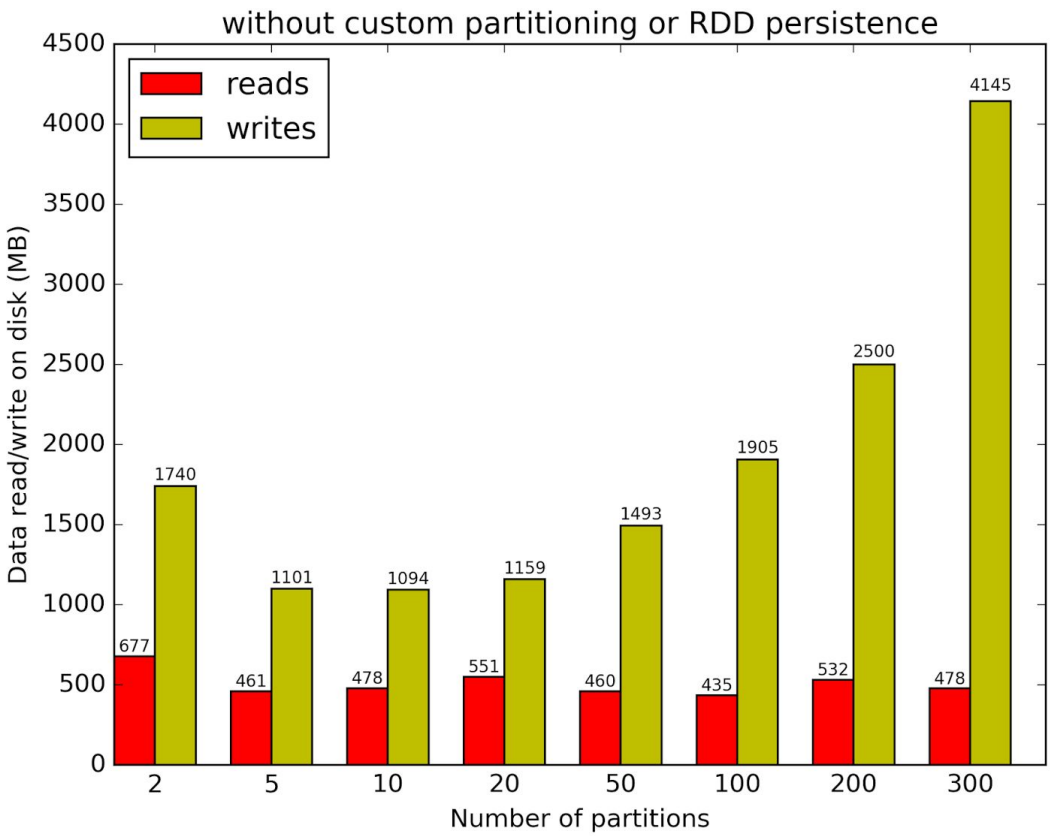



**Figure 1.2**. Impact of number of partitions on the amount of data read/written to disk.

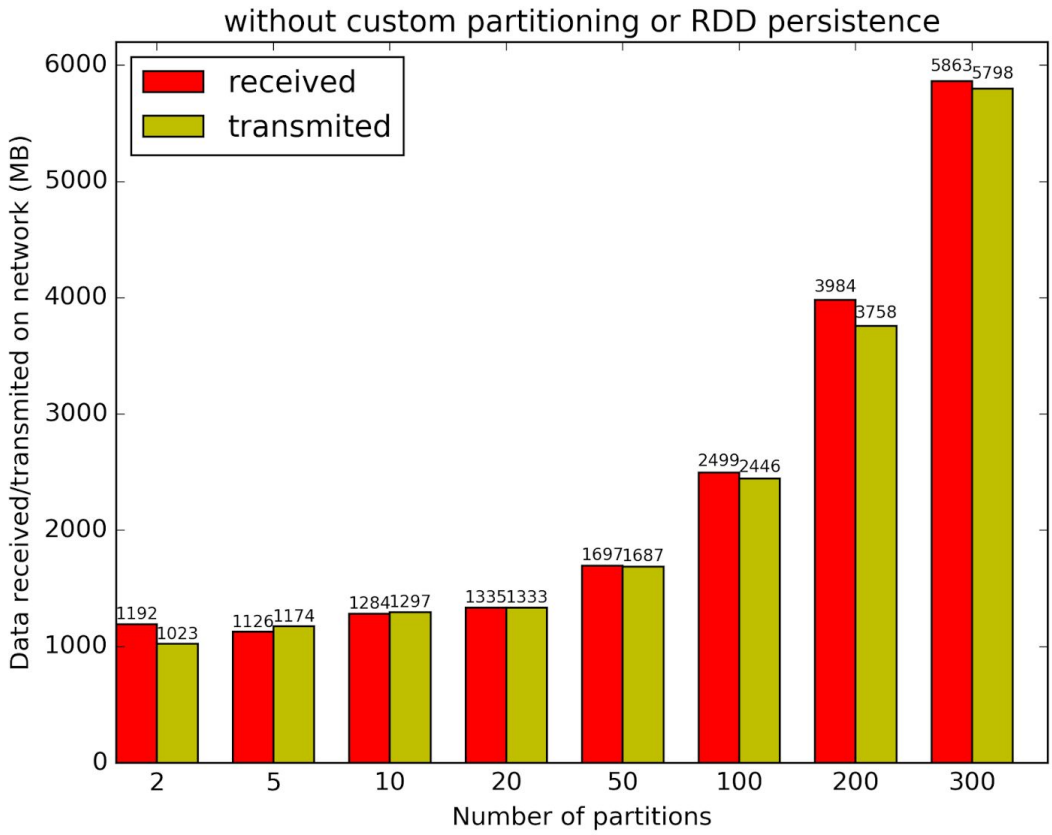


**Figure 1.3**. Impact of number of partitions on the amount of data received/transmitted through network.

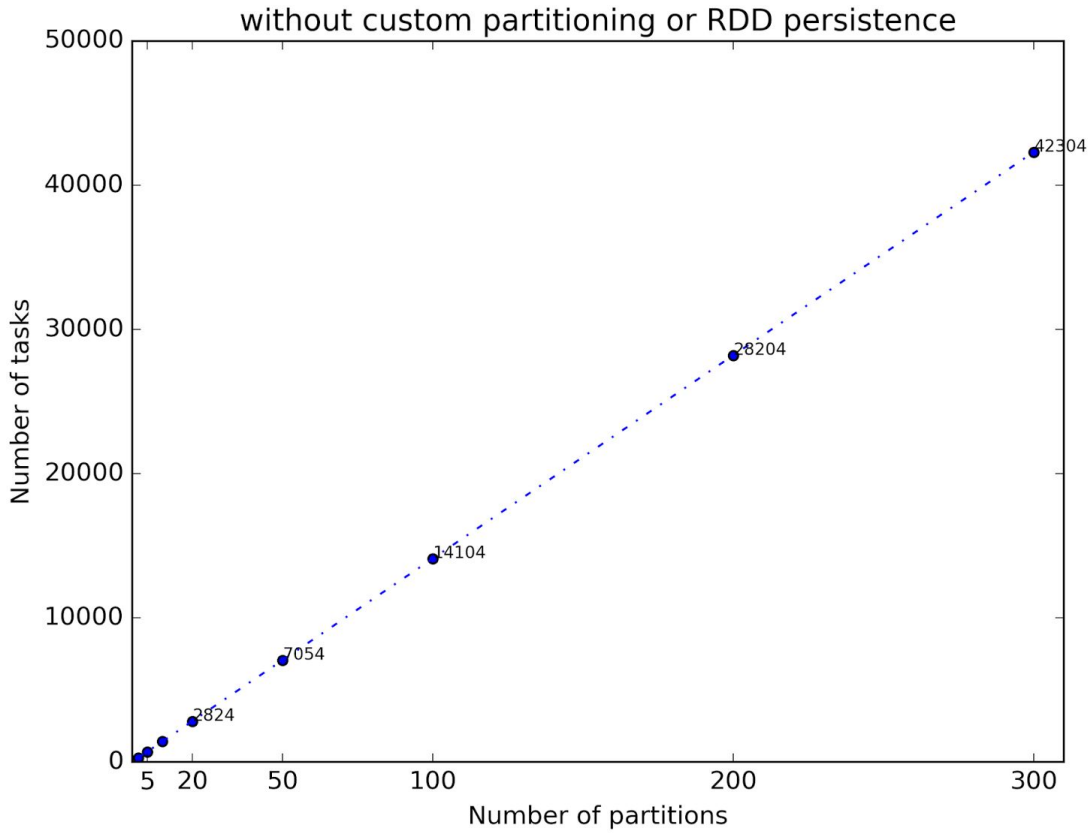



**Figure 1.3**. Impact of number of partitions on the number of tasks.

**Question 2**. Modify the Spark application developed in Question 1 to implement the PageRank algorithm **with** appropriate custom partitioning. Is there any benefit of custom partitioning? Explain. (*Hint: Do not assume that all operations on a RDD preserve the partitioning*)

**Answer:**
We implemented custom partitioning by **setting the parameter *preservesPartitioning* to *True* in the operations on RDDs** (e.g., map, flatMap,mapValues) (According the Spark documentation, map and flatMap operation by default do not preserve partitioning; mapValues operation by default preserves partitioning).

We investigated **the impact of the number of partitions of the *links RDD* on the application performance** (with the above custom partitioning but no RDD persistence). Figures 2.1 through 2.4 below show such impact in terms of performance metrics including completion time, amount of data read or written on disk, amount of data received or transmitted through network, and the total number of tasks.

Generally, **the amount of data read or written on disk, and amount of data received or transmitted through network is greatly reduced compared to that in Question 1**

(without custom partitioning), and remains about the same regardless of number of partitions. The total number of tasks is much smaller than that in Question 1 (without custom partitioning) although it still increases when the number of partitions increases. Number of tasks grows linearly with increasing number of partitions. Completion time overall follows the same tread as in Question 1 but is much shorter. The completion time is very long when there were too few partitions (e.g., 2). The completion time drops dramatically when the number of partitions increases. But beyond 100 or so, the number of partitions begins to have a negative impact on completion time. **When the number of partitions equals 50, the application completed fastest (in about 64 s)**.

In term of number of executors assigned tasks in the execution, when there were 5 or more partitions, all five executors got assigned tasks, with the only exception that when there were 200 partitions, only one executor was executing tasks (on the Master node). This explains why the amount of data transferred through network is almost zero.

**Overall, introducing custom partitioning significantly speeds up the application execution compared to that in Question 1 (without custom partitioning)**. The reason is that using custom partitioning (by preserving partitioning), all RDDs involved have the same partitioning scheme (partitioned on the url key), the same number of partitions and the same physical locations of the partitions. Thus data shuffling is greatly reduced as most operations are in narrow dependency instead of wide dependency because the partitions of the RDDs involved in an operation will be on the same node.

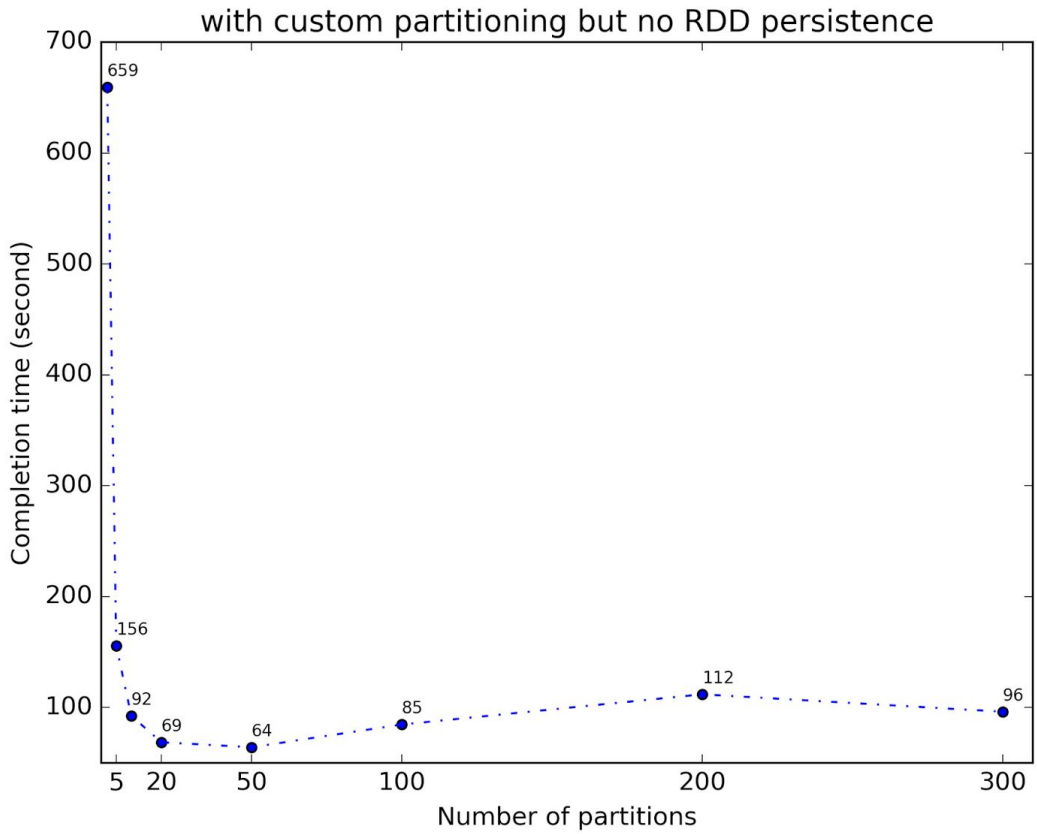



**Figure 2.1**. Impact of number of partitions on completion time.

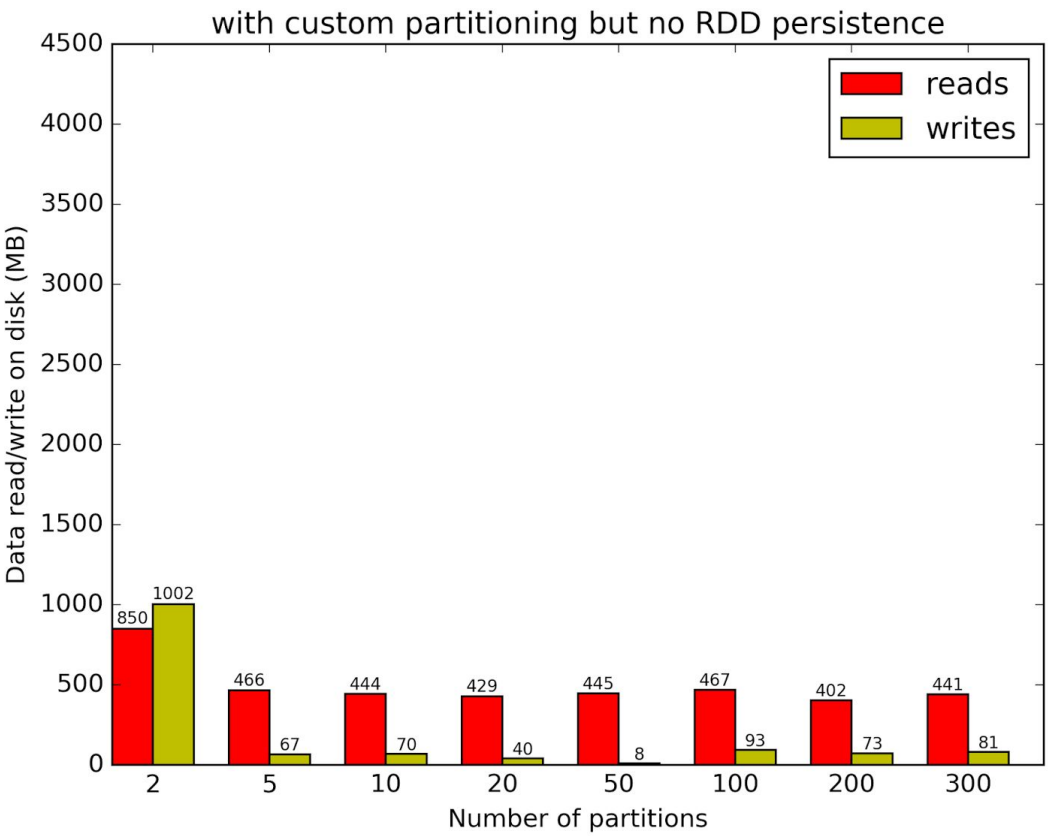



**Figure 2.2**. Impact of number of partitions on the amount of data read/written to disk.

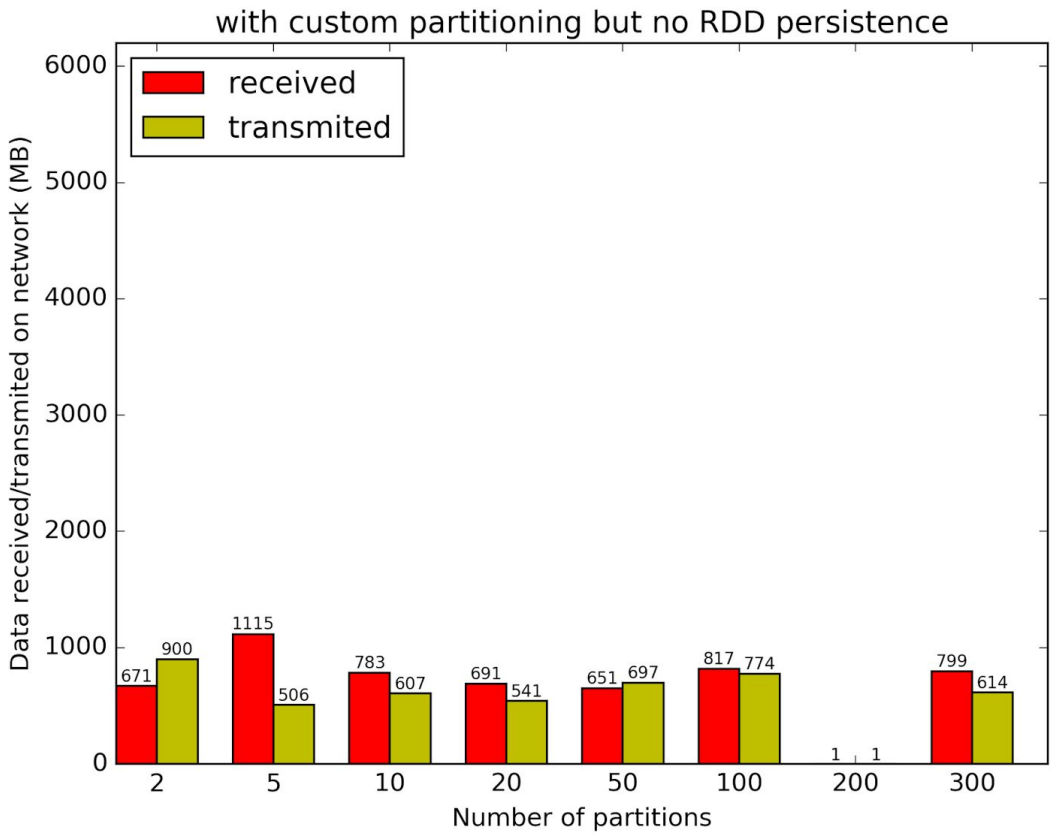


**Figure 2.3**. Impact of number of partitions on the amount of data received/transmitted through network.

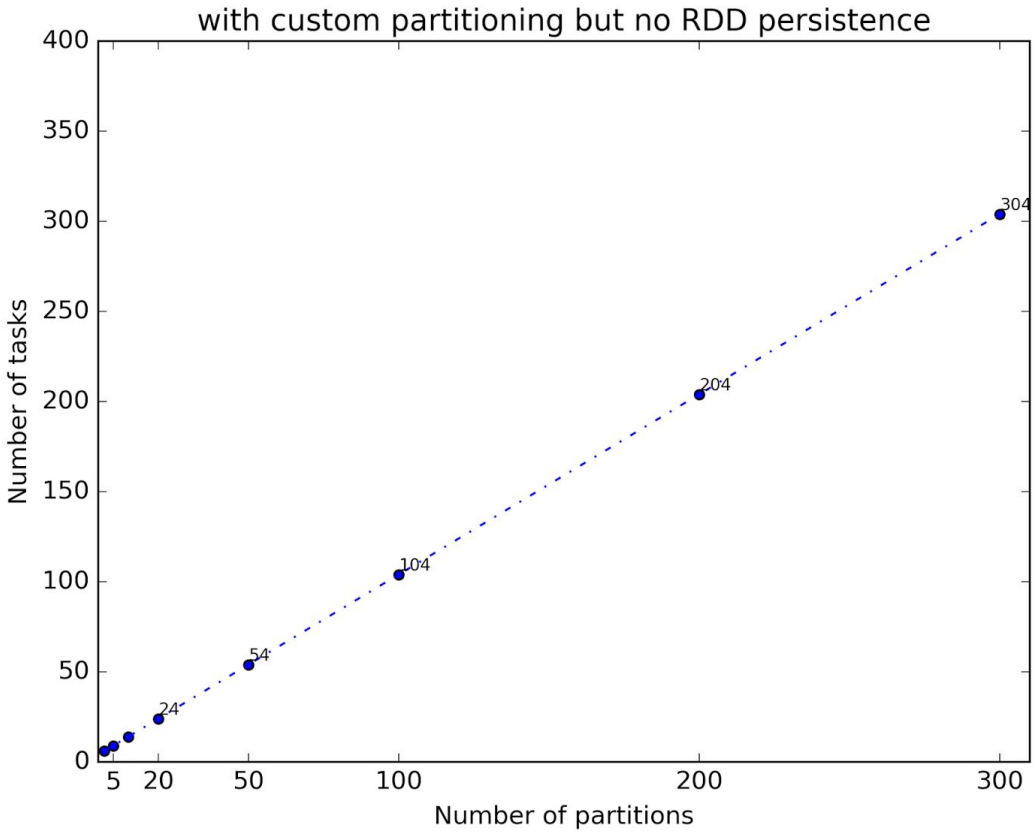



**Figure 2.3**. Impact of number of partitions on the number of tasks.

**Question 3**. Extend the Spark application developed in Question 2 to leverage the flexibility to persist the appropriate RDD as in-memory objects. Is there any benefit of persisting RDDs as in-memory objects in the context of your application? Explain.

**Answer:**
We extended the application developed in Question 2 to leverage the flexibility to persist the appropriate RDD as in-memory objects by ***calling the cache() action on the links RDD***.

We investigated **the impact of the number of partitions of the *links RDD* on the application performance** (with partitioning preserved and links RDD persisted in memory). Figures 3.1 through 3.4 below show such impact in terms of performance metrics including completion time, amount of data read or written on disk, amount of data received or transmitted through network, and the total number of tasks.

The amount of data read or written on disk is similar to that in Question 2 and remains about the same regardless of number of partitions. **The amount of data received or transmitted through network is greatly reduced compared to that in Question 2.** The total number of tasks is the same as in Question 2 and it still increases when the number of partitions increases. **Completion time overall follows the same tread as in Question 2 but is**

**slightly faster**. The completion time is very long when there were too few partitions (e.g., 2). The completion time drops dramatically when the number of partitions increases. But beyond 100 or so, the number of partitions begins to have a negative impact on completion time. **When the number of partitions equals 50, the application completed fastest (in about 62 s)**.

In term of number of executors assigned tasks in the execution, when there were 100 or more partitions, executors got assigned highly unbalanced number of tasks, with only one executor getting assigned a very large portion of the total number of tasks while other executors got assigned only a few tasks.

**Overall, introducing RDD persistence slightly speeds up the application execution compared to that in Question 2**. The reason is that, besides the benefits of custom partitioning discussed in Question 2, persisting the links RDD in memory save the work of repeatedly reading links RDD from HDFS. With custom partitions on RDD, data shuffling in the Spark application is reduced. But the HDFS has its own partitioning of the data stored in HDFS. Thus without persisting the link RDD in memory, the Spark application might need to repeatedly read data that are stored potentially across nodes. But with RDD persistence, only one such read is needed. This explains the slight improvement on application performance and the reduction of network bandwidth (Figure 3.3 compared to Figure 2.3).

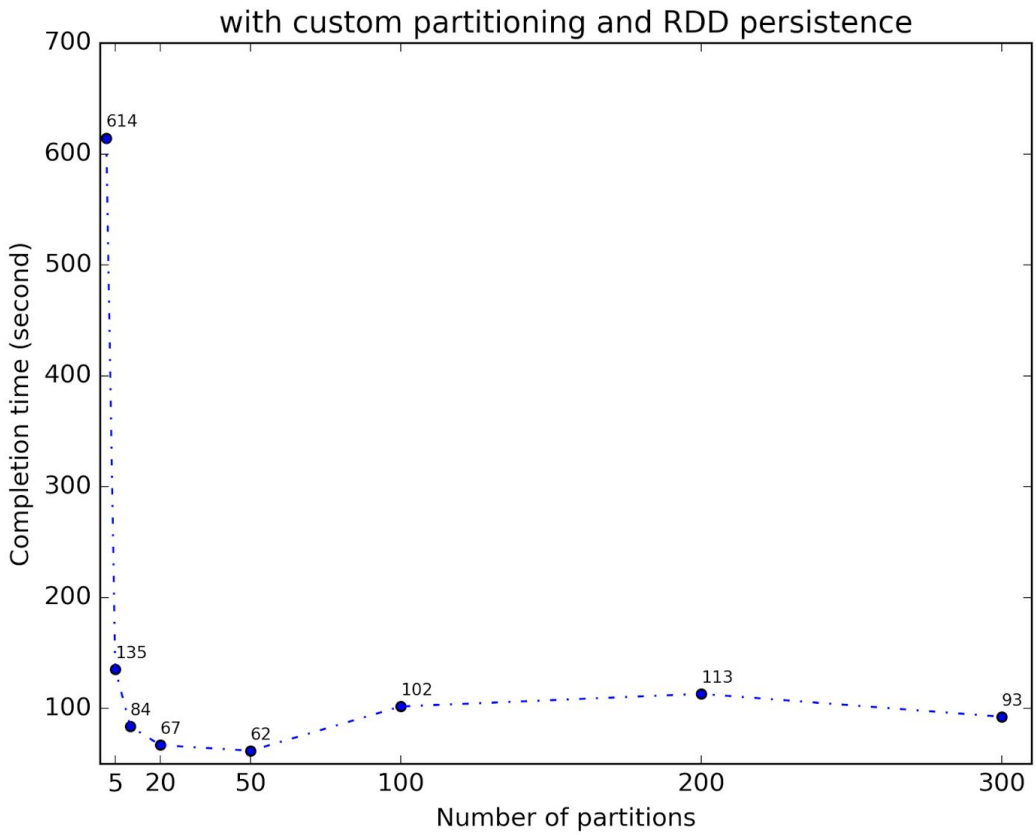



**Figure 3.1**. Impact of number of partitions on completion time.

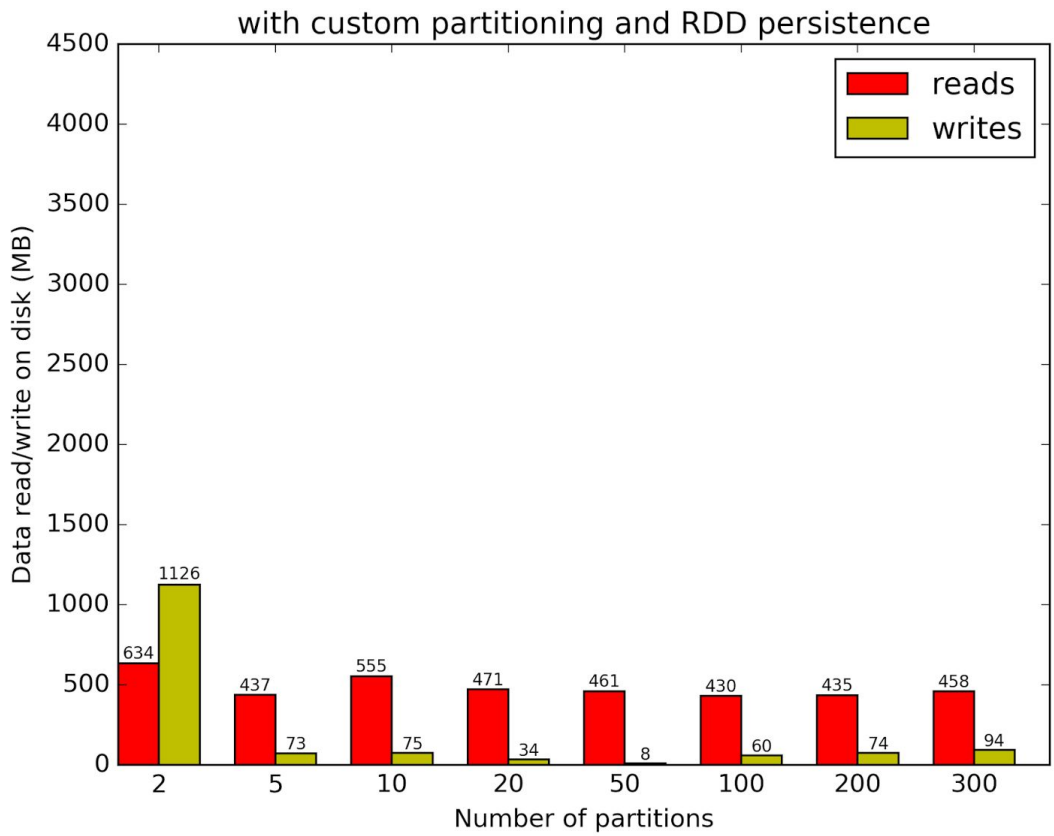



**Figure 3.2**. Impact of number of partitions on the amount of data read/written to disk.

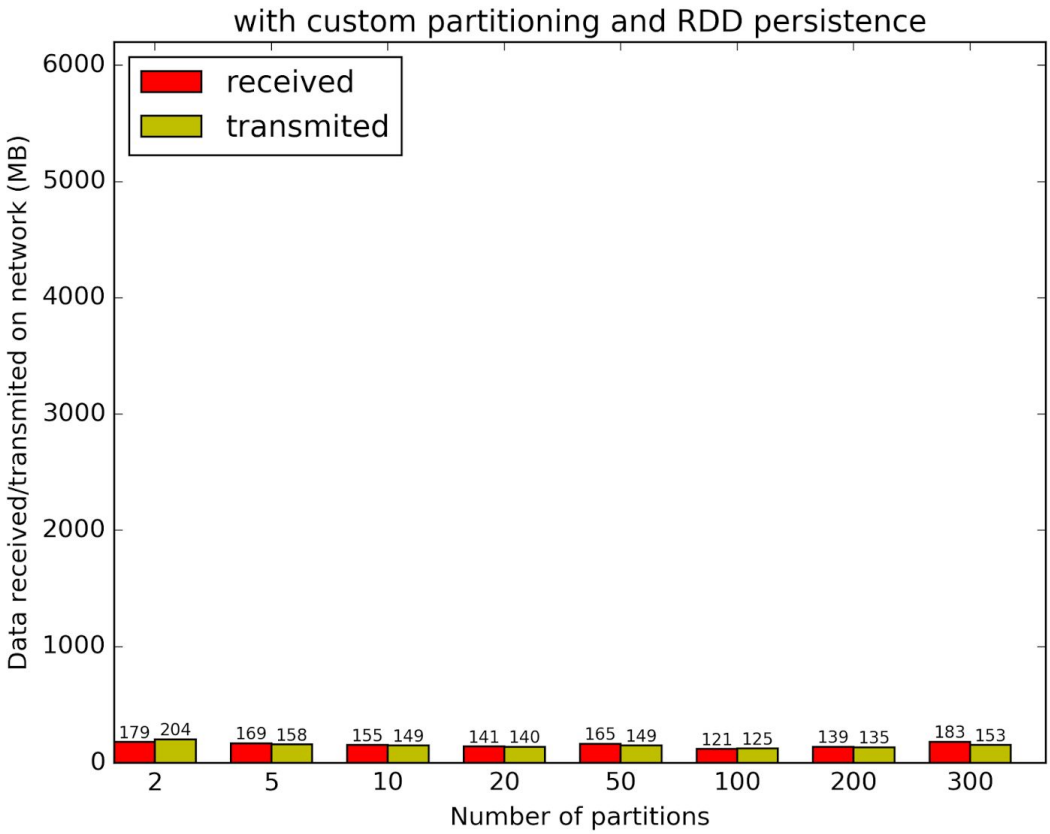


**Figure 3.3**. Impact of number of partitions on the amount of data received/transmitted through network.

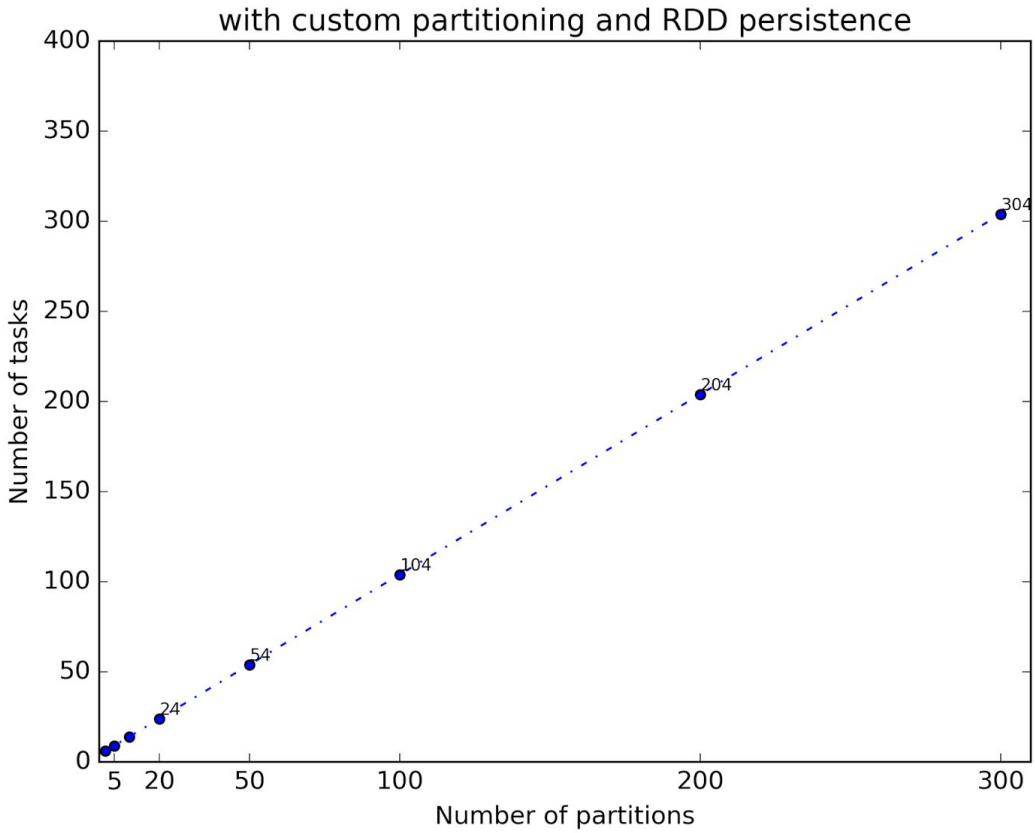



**Figure 3.3**. Impact of number of partitions on the number of tasks.

**Question 4**. Analyze the performance of `CS-838-Assignment2-PartA-Question2` by varying the number of RDD partitions from 2 to 100 to 300. Does increasing the number of RDD partitions always help? If no, could you find a value where it has a negative impact on performance and reason about the same.

**Answer:**
In Question 2 we investigated **the impact of the number of partitions (including 2, 100, 300) of the *links RDD* on the application performance** (with custom partitioning but no RDD persistence). Figure 2.1 through 2.4 show such impact in terms of performance metrics including completion time, amount of data read or written on disk, amount of data received or transmitted through network, and the total number of tasks.

Clearly, increasing the number of RDD partitions does NOT always help with application performance. When the number of partitions is very small, say 2, the performance is very poor. **The performance is best with number of partitions being somewhere around 50 partitions on the links RDD**. Beyond 50 partitions, say 100 or 300 partitions, the performance starts to degrade with increasing number of partitions.

This is mainly because under a very small number of partitions, the tasks division would be quite coarse grained. That means the workload of each tasks would more likely to be imbalanced. The performance of the application will be determined by the slowest task. With a larger number of partitions, there will be a larger number of finer-grain tasks. But the cost of communication between tasks increases. A number of partitions somewhere in the middle would balance these tradeoffs and achieve the best performance.

**Question 5**. Visually plot the lineage graph of the `CS-838-Assignment2-PartA-Question3` application. Is the same lineage graph observed for all the applications developed by you? If yes/no, why? The Spark UI does provide some useful visualizations.

**Answer:**
We have two possible answers to this question.

**(1)** Figure 5.1 shows the lineage graph of the `CS-838-Assignment2-PartA-Question3` application. By definition, lineage graph is a graph showing the RDDs involved in the application and the transformations applied on the RDDs. Applications developed for Q1, Q2, and Q3 essentially follow the same flow of the PageRank algorithm (regardless of custom partitioning or RDD persistence that affect performance of the algorithm). Thus this lineage graph is the same for all the applications, just as the lineage graph presented in the PageRank example in the SPARK paper we read (Figure 3 of the paper). In this respect, the visualizations provided by Spark UI is Spark application DAG, not a clean lineage graph.

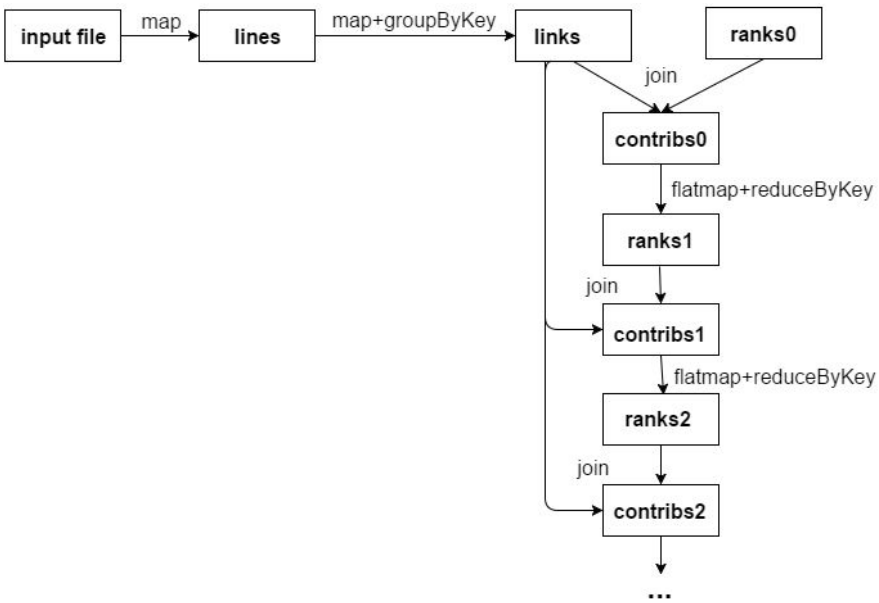
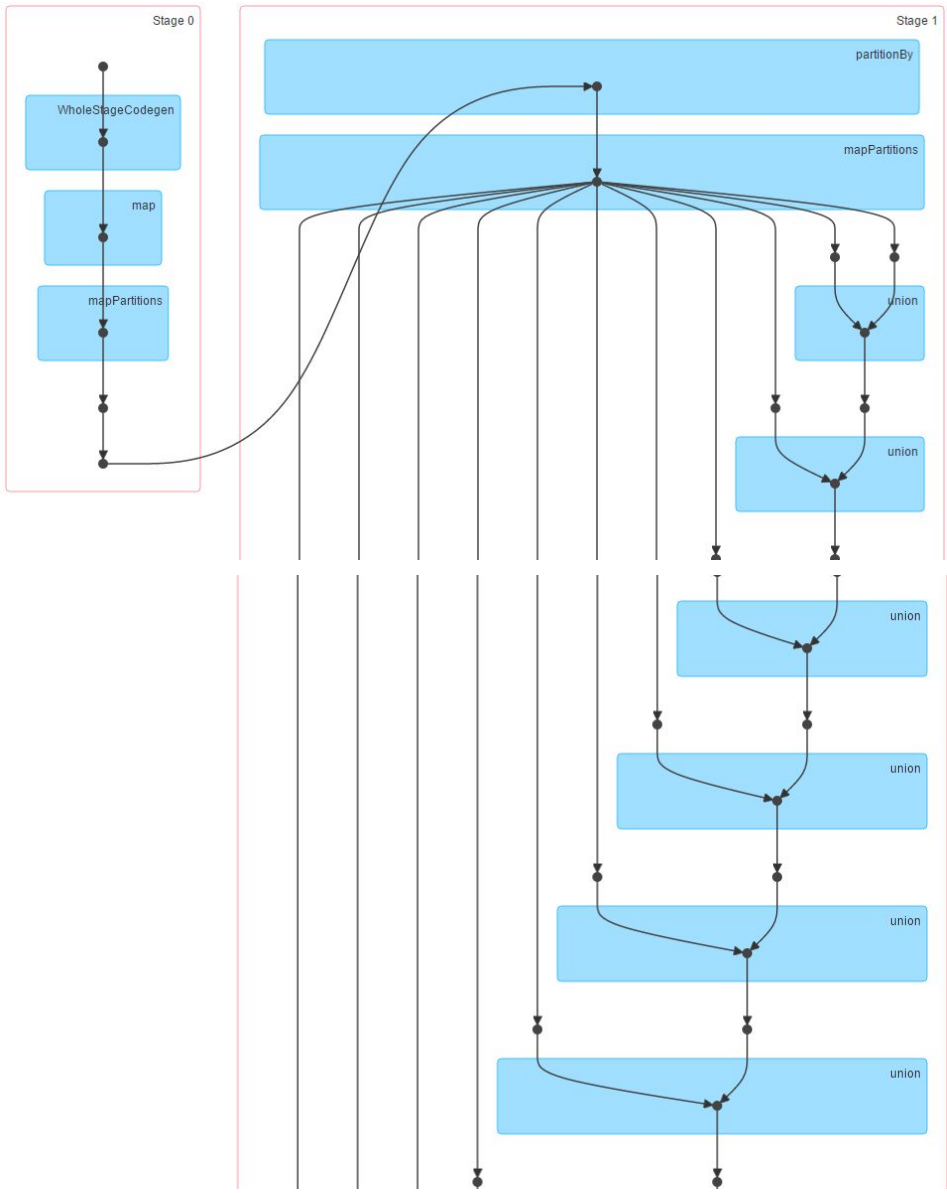



Figure 5.1. Lineage graph of the `CS-838-Assignment2-PartA-Question3` application.

**(2)** However, since lineage graph can be tied to a particular partition of a RDD, it also makes sense to look at the DAG visualizations provided by Spark UI. Figure 5.2 shows the DAG visualization provided by Spark UI for the `CS-838-Assignment2-PartA-Question3` application (black dots are RDDs; blue boxes are operations). This is the same as that for Question 2, but different from that for Question 1. This is because both Question 2 and

Question 3 use the same custom partitioning but Question 1 does not use the same custom partitioning. Due to preserving partitioning, applications in Q2 and Q3 end with fewer stages (only 2 stages) as narrow dependency operations are pipelined in a single stage. Without preserving partitioning, the application in Q1 end up with more stages (22 stages) as stages are divided at the boundary of wide dependencies.



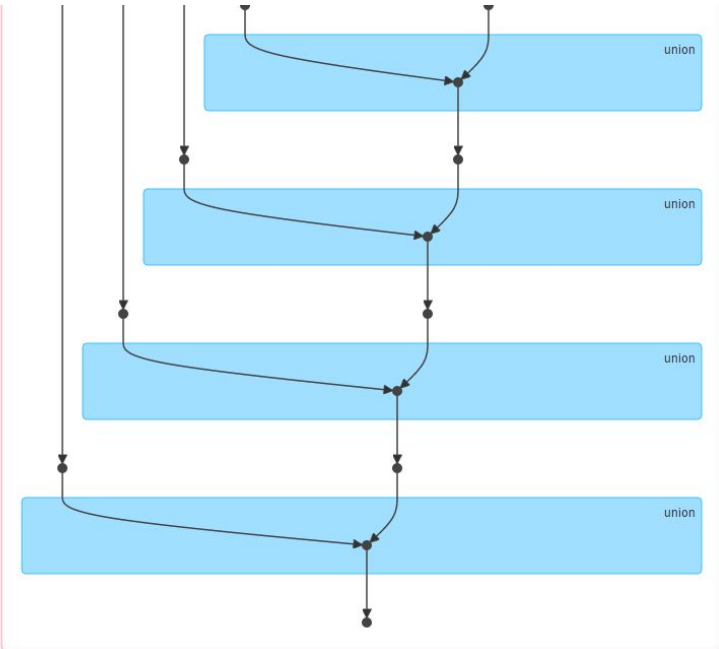



Figure 5.2. Lineage graph of the `CS-838-Assignment2-PartA-Question3` application provided by Spark UI DAG visualization.

**Question 6**. Visually plot the Stage-level Spark Application DAG (with the appropriate dependencies) for all the applications developed by you till the second iteration of PageRank. The Spark UI does provide some useful visualizations. Is it the same for all the applications? Id yes/no, why? What impact does the DAG have on the performance of the application?

**Answer:**
Figures 6.1 through 6.3 show the Stage-level Spark Application DAG for all the applications till the second iteration of PageRank. **It is the same for applications developed in Question 2 and Question 3 but different for the application developed in Question 1**.

The **reason** is that both Question 2 and Question 3 use the same custom partitioning (preserving partitioning) but Question 1 does not use the same custom partitioning. Due to preserving partitioning, applications in Q2 and Q3 end with fewer stages (only 2 stages in the complete job DAG) as narrow dependency operations are pipelined in a single stage. Without preserving partitioning, the application in Q1 end up with more stages (4 stages till the second iteration) as stages are divided at the boundary of wide dependencies.

The DAG has impact on the performance because it implies data dependencies as discussed above. In Spark application, tasks were divided into stages at the boundary of wide dependencies. That means tasks in the same stage have only narrow dependencies and can be pipelined in execution (do not need to wait the upstream tasks to complete to move forward). More stages implies more wide dependencies thus many tasks may end up with waiting for some upstream tasks to complete before they can move forward. The DAG

for Q2 and Q3 are highly pipelined in execution thus the applications for Q2 and Q3 have better performance.

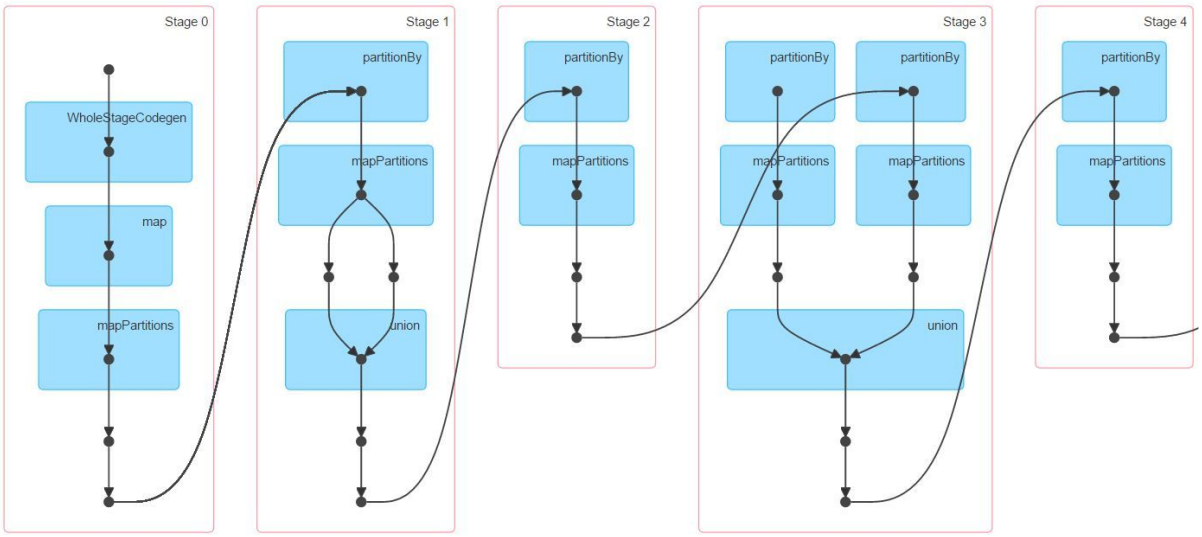



Figure 6.1. Spark Application DAG for Q1 application.

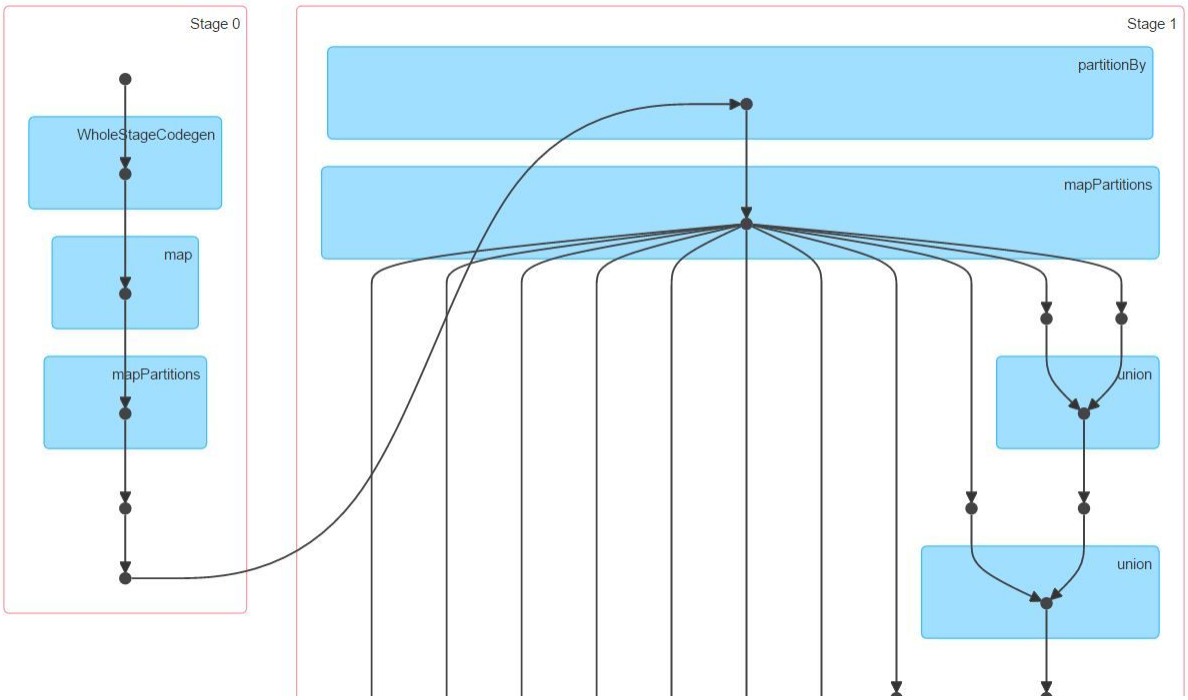



Figure 6.2. Spark Application DAG for Q2 application.

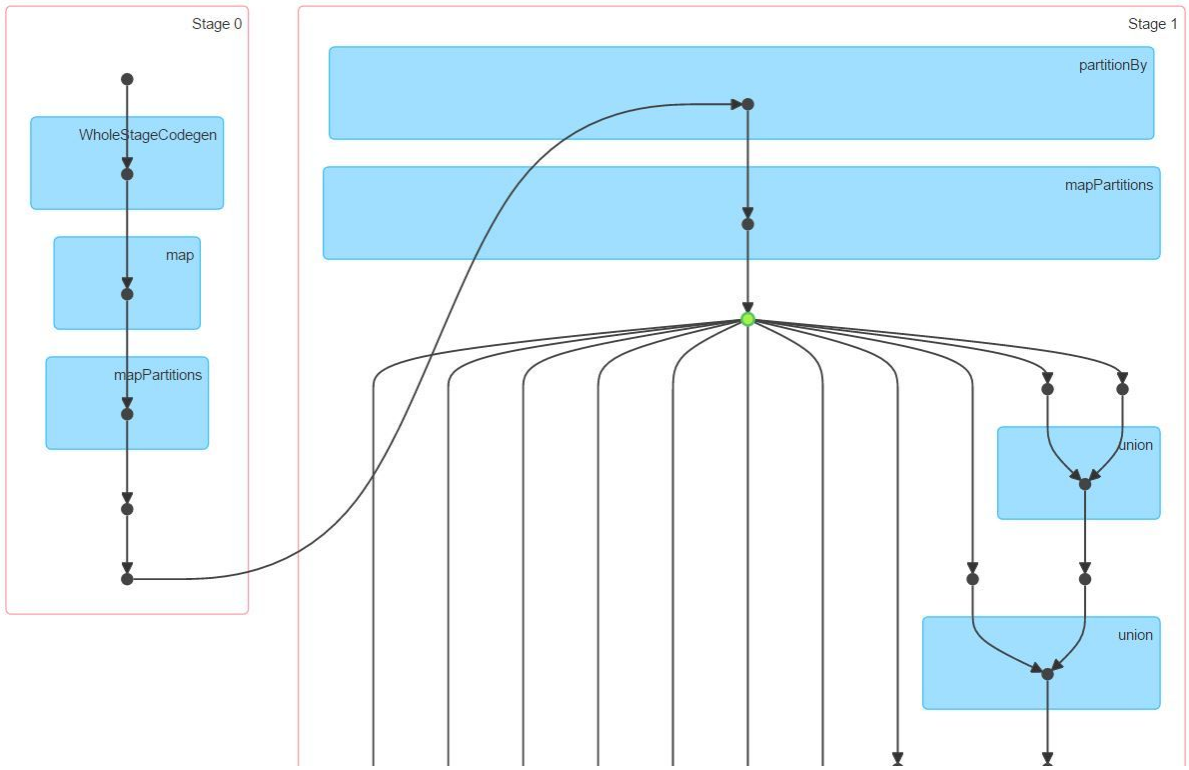



Figure 6.3. Spark Application DAG for Q3 application.

**Question 7**. Analyze the performance of `CS-838-Assignment2-PartA-Question3` and `CS-838-Assignment2-PartA-Question1` in the presence of failures. For each application, you should trigger two types of failures on a desired `Worker` VM when the application reaches 25% and 75% of its lifetime.
The two failure scenarios you should evaluate are the following:

- Clear the memory cache using `sudo sh -c "sync; echo 3 > /proc/sys/vm/drop_caches"`
- Kill the `Worker` process.

You should analyze the variations in application completion time. What type of failures impact performance more and which application is more resilient to failures? Explain your observations.

**Answer:**
The performance (completion time) of the applications in the presence of the two types of failures are presented in Table 7.1 and Table 7.2.

Clearing memory cache on a Worker VM has a more profound impact on the Q3 application (with custom partitioning and RDD persistence) than on the Q1 application (without any custom partitioning and RDD persistence). This make sense because Q1 application does not depend as heavily on memory cache as Q3 application does. For Q1 application, every time a record (in links RDD) is needed, it is read from HDFS, no memory cache is used. For Q3 application, starting from the second time a record (in links RDD) is needed, it will be fetch from the memory and possibly from the cache.

Another observation is that, for Q3 application, cache failure at later point of time over the execution has a more significant negative impact on its performance. This is due to the fact that at later point of time over the execution, the lineage of a RDD partition grows longer. A cache failure thus will trigger a longer sequence of operations to recover the lost RDD partitions.

Table 7.1 The impact of clearing memory cache on a Worker VM on the application performance.

| | Completion time (second) (average of 3 runs) | |
|---|---|---|
| | Question1 (partitions = 20) | Question3 (partitions = 50) |
| No cache failure | 118.7 | 61.9 |
| Clear at 25% | 116.8 | 63.3 |
| Clear at 75% | 118.8 | 68.5 |

Killing the worker process on a Worker VM (tasks were assigned to this Worker prior kill) has a profound impact on BOTH the Q3 application (with custom partitioning and RDD persistence) AND the Q1 application (without any custom partitioning and RDD persistence). This make sense because both applications rely on the Worker to complete computation tasks. As the worker process is killed, the completed tasks, tasks in progress, and RDD partitions on that Worker are lost. The tasks have to be rescheduled to other Workers and the lost RDD partitions have to be recovered on other Workers. This takes time.

Another observation is that, for BOTH applications, killing the worker process at later point of time over the execution has a more significant negative impact on its performance. This is due to the fact that at later point of time over the execution, the lineage of a RDD partition grows longer. A failed worker thus will trigger a longer sequence of operations to recover the lost RDD partitions.

Table 7.2 The impact of killing the Worker process on the application performance.

| | Completion time (second) (average of 3 runs) | |
|---|---|---|
| | Question1 (partitions = 20) | Question3 (partitions = 50) |
| No kill | 118.7 | 61.9 |
| Kill at 25% | 137.9 | 69.2 |
| Kill at 75% | 162.8 | 96.0 |

==========For Part B and C, See the README.txt files in each folders==========

**Part B - Structured Streaming**

**Question 1**. One of the key features of Structured Streaming is support for window operations on event time (as opposed to arrival time). Leveraging the aforementioned feature, you are expected to write a simple application that emits the number of retweets (RT), mention (MT) and reply (RE) for an hourly window that is updated every 30 minutes based on the timestamps of the tweets. You are expected to write the output of your application onto the standard console. You need to take care of choosing the appropriate output mode while developing your application.
In order to emulate streaming data, you are required to write a simple script that would periodically (say every 5 seconds) copy one split file of the Higgs dataset to the HDFS directory your application is listening to. To be more specific, you should do the following -

- Copy the entire split dataset to the HDFS. This would be your staging directory which consists of all your data.
- Create a monitoring directory too on the HDFS that your application listens to. This would be the directory your streaming application is listening to.
- Periodically, move your files from the staging directory to the monitoring directory using the hadoop fs -mv command.

**Question 2**. Structured Streaming offers developers the flexibility is decide how often should the data be processed. Write a simple application that emits the twitter IDs of users that have been mentioned by other users every 10 seconds. You are expected to write the output of your application to HDFS. You need to take care of choosing the appropriate output mode while developing your application. You will have to emulate streaming data as you did in the previous question.
**Question 3**. Another key feature of Structured Streaming is that it allows developers to mix static data and streaming computations. You are expected to write a simple application that takes as input the list of twitter user IDs and every 5 seconds, it emits the number of tweet actions of a user if it is present in the input list. You are expected to write the output of your application to the console. You will have to emulate streaming data as you did in the previous question.
Notes:

- All your applications should take a command line argument - HDFS path to the directory that your application will listen to.
- All the applications will be executed using spark-submit.
- All the applications should use the cluster resources appropriately.
- You may need to ensure that the hive metastore is running in the background before running any of your applications: hive --service metastore &.

## Part C - Apache Storm

**Question 1**. To answer this question, you need to extend the `PrintSampleStream.java` topology to collect tweets which are in English and match certain keywords (at least 10 distinct keywords) at your desire. You should collect at least 500'000 tweets which match the keywords and store them either in your local filesystem or HDFS. Your topology should be runnable both in `local mode` and `cluster mode`.

**Question 2**. To answer this question, you need to create a new topology which provides the following functionality: every 30 seconds, you should collect all the tweets which are in English, have certain hashtags and have the *friendsCount* field satisfying the condition mentioned below. For all the collected tweets in a time interval, you need to find the top 50% most common words which are not stop words (http://www.ranks.nl/stopwords). For every time interval, you should store both the tweets and the top common words in separate files either in your local filesystem or HDFS. Your topology should be runnable both in local mode and cluster mode.
Your topology should have at least three spouts:

- Spout-1: connects to Twitter API and emits tweets.
- Spout-2: It contains a list of predefined hashtags at your desire. At every time interval, it will randomly sample and propagate a subset of hashtags.
- Spout-3: It contains a list of numbers. At every interval, it will randomly sample and pick a number N. If a tweet satisfies friendsCount < N then you keep that tweet, otherwise discard.

Notes:

- Please pick the hashtags list such that every time interval, at least 100 tweets satisfy the filtering conditions and are used for the final processing.
- You may want to increase the incoming rate of tweets which satisfy your conditions, using the same tricks as for Question 1.