Shawn Zhong, Suyan Qu, Sulong Zhou

CS 744: Big Data System

Report for PageRank

Sept. 22, 2019

Experiment Setup

In the cluster, we have one master node and two slave nodes. Each slave node has 5 cores of CPU, 30.2 GB of Memory, 40 GB of disk.

We first wrote a Python Spark application that implements the PageRank algorithm. The code is attached to the end of this report.

We used the following datasets to run our application:

- Berkeley-Stanford web graph from 2002
- enwiki-20180601-pages-articles

During each test, we run only 5 iterations of the PageRank algorithm.

For the cache analysis, we run our application on both datasets with cache and without cache to check the impact of cache.

For the partition analysis, we run our application on the first dataset to measure the effect of the number of partitions to the execution time.

For the fault-tolerant analysis, we run our application on both datasets with default partitioning and no cache. In the first experiment, we clear the memory cache in the middle of execution. In the second experiment, we killed one of the worker when roughly 50% of the job is done.

Lineage Graph With Default Partitioning

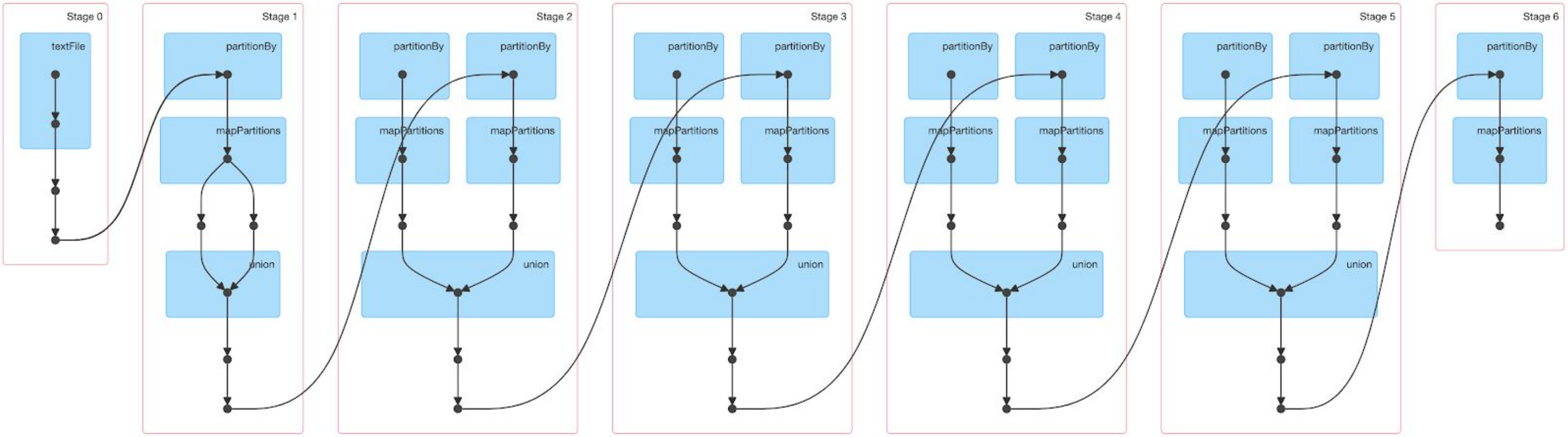


Figure 1.1: Lineage Graph without cache

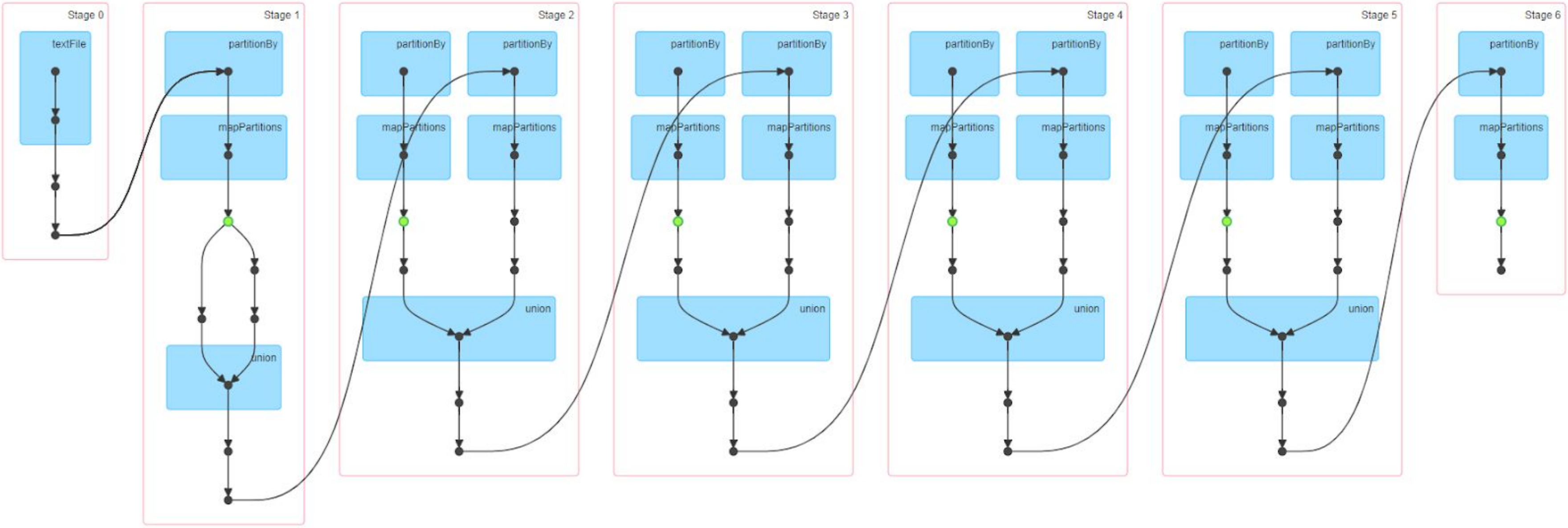


Figure 1.2: Lineage Graph with cache

Figure 1.1 and Figure 1.2 show the lineage graph for running PageRank with default partitioning. Stage 0 refers to the stage that reads from the Hadoop File System, Stage 6 refers to writing the result back to the file system, and Stages 1 - 5 each corresponds to one iteration of the PageRank algorithm.

Cache Analysis

Cache analysis is performed with default partitioning and no machine failure. We comparing running PageRank without caching with running PageRank that caches the RDD consisting of each page and a list of all its neighbors. The ranks of pages are not cached because this RDD is updated every iteration, in which case it would always be inside the memory with or without caching.

	web-BerkStan.txt	**enwiki-pages-articles**
Without Cache	5.3 min	28 min
With Cache	5.2 min	28 min

Table 1: Completion time for web-Berkstan.txt vs. enwiki-pages-articles

We found that cache has little effect on the overall completion time for both datasets.

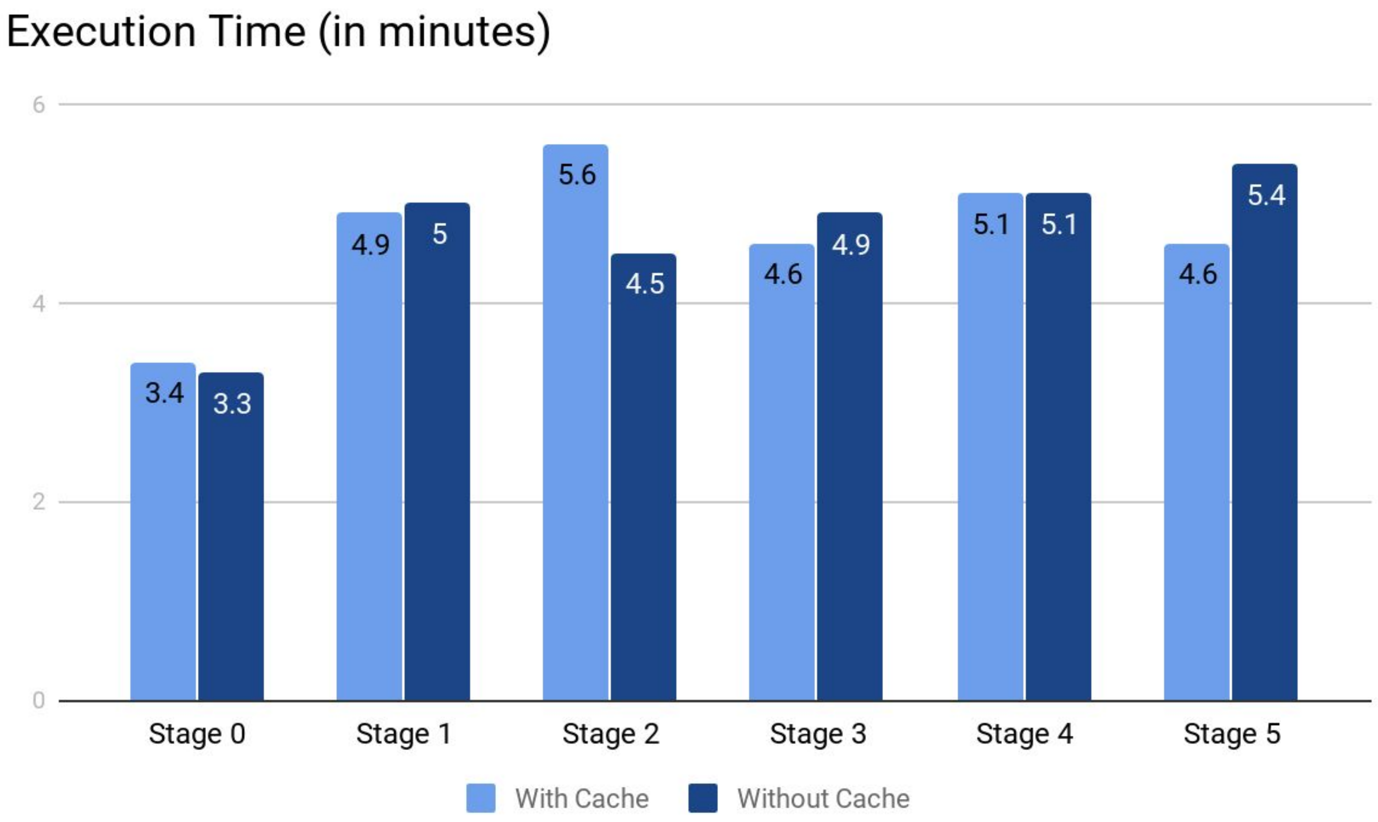


Figure 2.1: Execution time for each stage with and without caching

We execute the algorithm with cache and without cache separately for multiple times and take an average for the processing time. According to Figure 2.1 and Table 1, even though the average accumulated time for all stages are exactly the same, the execution time at each stage is slightly different between algorithm with cache and without cache. Specifically, the algorithm with cache outperforms at stage 1, 3 and 5 while the algorithm without cache outperforms at stage 0, 2, 4. We speculate the reason is probably due to the speed of data read and write. However, the difference in execution time for both each stage and all stages could become significant if the iteration of each process increases.

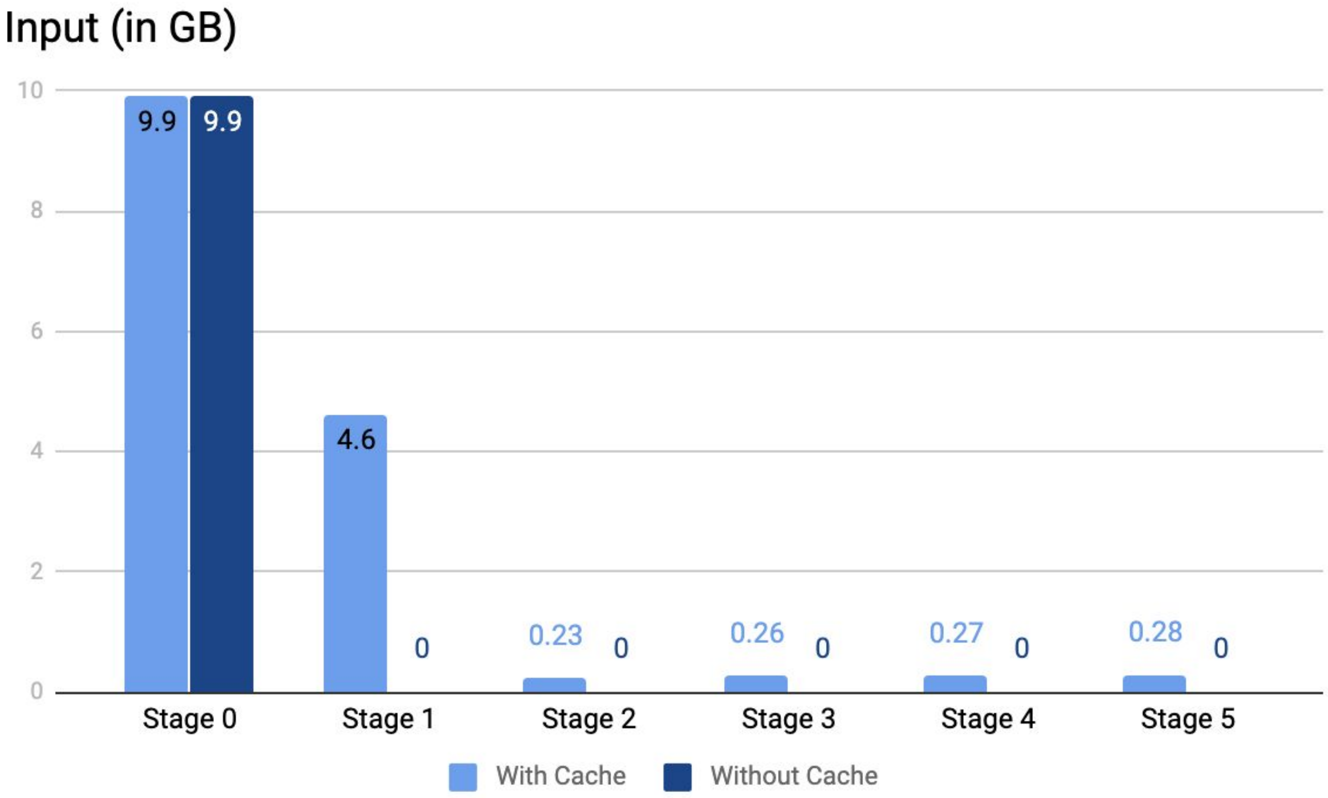


Figure 2.2: Data read during each stage with and without caching

Figure 2.2, on the other hand, explains why the runtime for PageRank with and without cache have similar performances. As shown, PageRank with cache requires more disk I/O than without cache. This is likely because the disk does not have enough memory to cache all the data, which leads to an overhead on each iteration. It is also noticed that the disk I/O is significantly larger during the first iteration than iterations afterward, indicating that running PageRank with cache may have a more significant improvement over running without cache if we run more iterations.

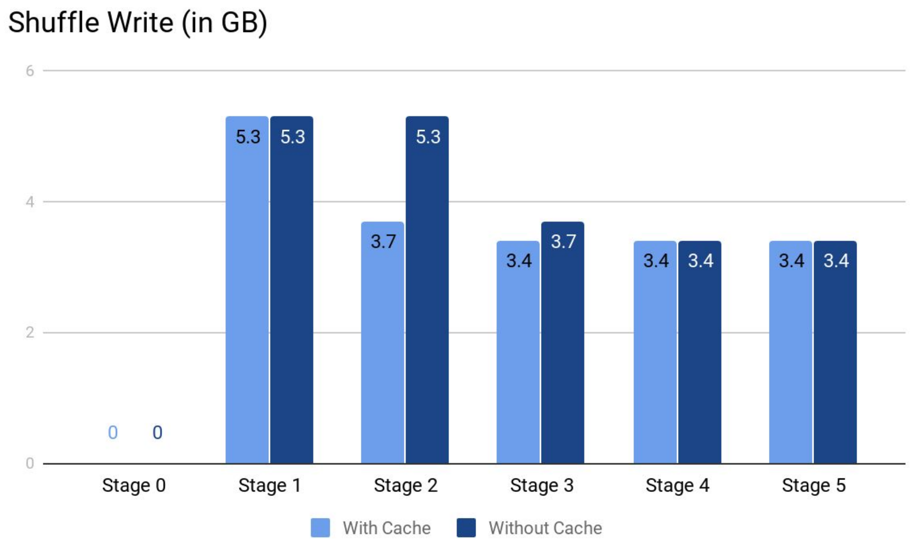


Figure 2.3: Shuffle writes during each iteration

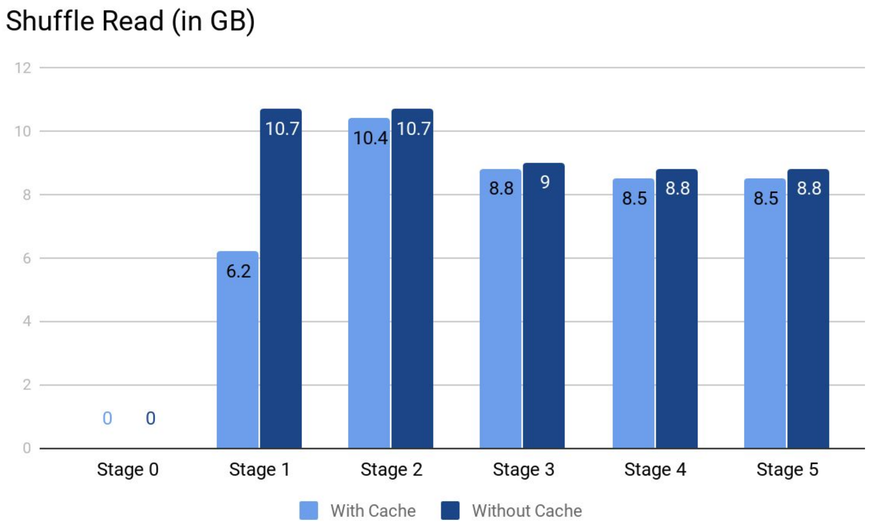


Figure 2.4: Shuffle reads during each iteration

As shown in Figure 2.3 and Figure 2.4, the algorithm with cache always performs better or equal to the algorithm without cache in terms of the amount of data (in GB) for both shuffle read and shuffle write. This is largely due to the fact that the cache will store the data in memory so that the amount of data transmitted from disk is reduced. In addition, both the shuffle read and write starts at stage 0. Since stage 2, both of them gradually decreases. Nevertheless, it is unknown why there is a distinct drop for the shuffle read in stage 1.

Partition Analysis

Since the number of partitions will affect the number of stages, we only analyze the execution time for this part.

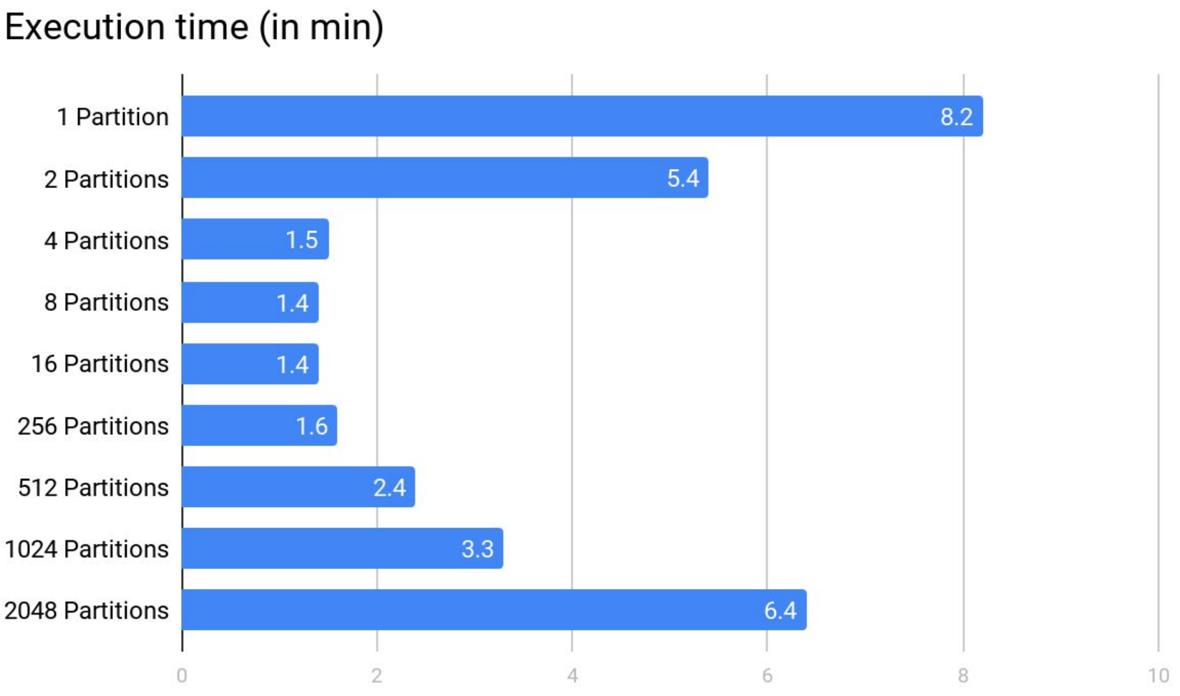


Figure 3.1: Execution time with various number of partitions

As shown in figure 3.1, as the number of partitions increases, the execution time decreases and then increases. For a small number of partitions, data points with the same key may be located far apart or even on different workers when there is only 1 partition, causing low locality and high cost to swap data points around when reducing data by key. As the number of partitions increases, on the other hand, the overhead for partitioning and merging increases to a point where the overhead dominates the increase in locality, and further partitioning will lead to longer execution time. From our experiment, PageRank has the best performance with 8 and 16 partitions in our setup.

Fault Tolerance Analysis

	web-BerkStan.txt	**enwiki-pages-articles**
Normal Execution	5.3 min	28 min
Memory Cache Cleared during Execution	5.4 min	29 min
One worker killed	5.6 min	39 min

When a worker is killed, the following exception is thrown, and 171 tasks failed :

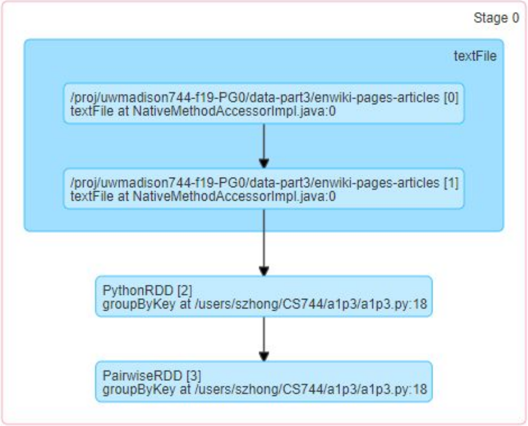
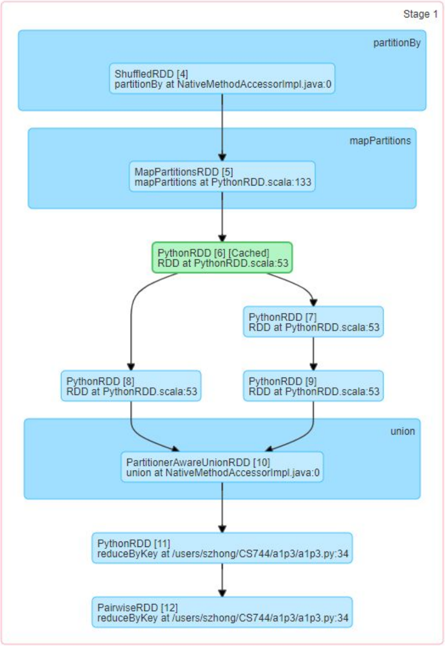
```
org.apache.spark.shuffle.FetchFailedException: Failed to connect to /128.104.223.155:37793
```

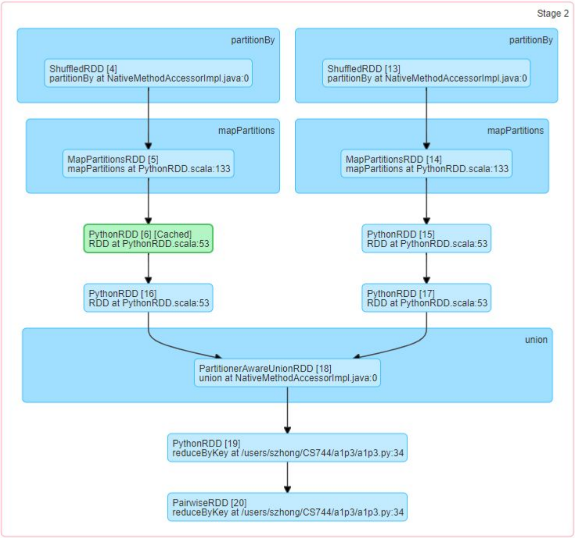
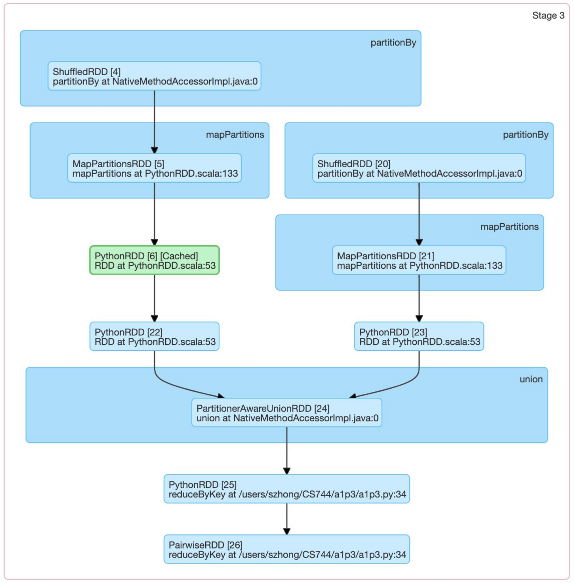
When a worker is killed, the following exception is thrown, and 8 tasks failed :

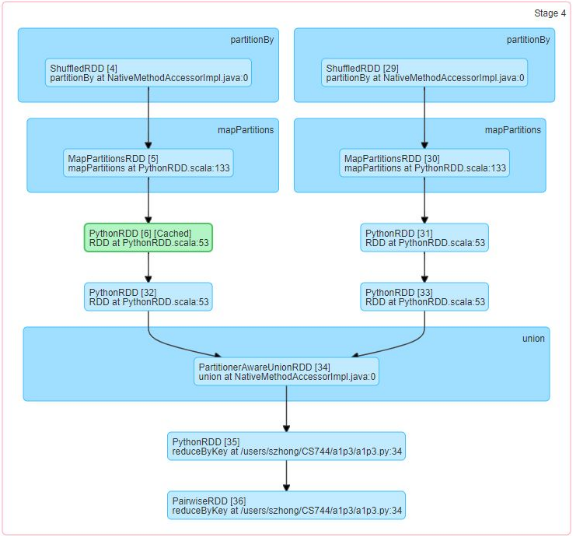
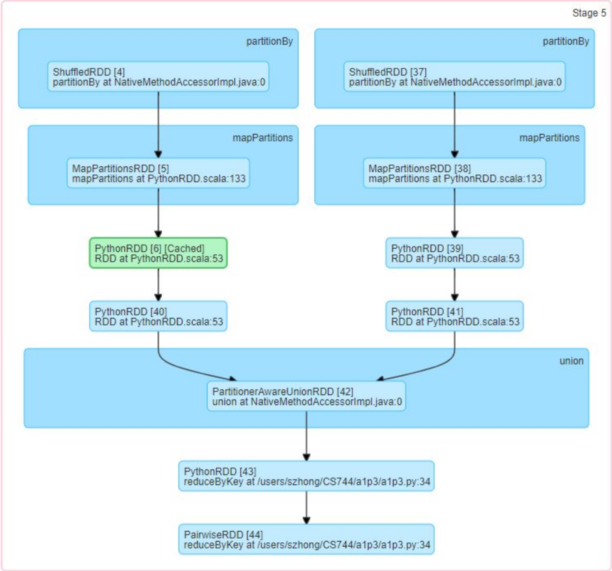
```
org.apache.spark.shuffle.FetchFailedException: Error in reading
FileSegmentManagedBuffer{file=/mnt/data/spark-8eed9f6f-16a3-408e-8506-59b2046ec63a/executor-efa886ba-
4f7b-47ff-9f27-f3707979bf67/blockmgr-9a06a357-8722-44e9-83b2-c207dd040591/19/shuffle_3_2_0.data,
offset=5240080, length=34540}
```

In all the cases, Spark is able to finish all the jobs correctly, since RDDs provide an interface based on coarse-grained transformations that apply the same operation to many data items. This allows them to efficiently provide fault tolerance by logging the transformations used to build its lineage rather than the actual data. For example, if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute. The execution time will be significantly longer with worker malfunction since the worker needs to retry the failed tasks and its precedent tasks. When memory cache is cleared during execution, the execution time will be slightly longer than that of normal execution, since it only needs to reload some data previously cached by the OS.

Case Study: enwiki-pages-articles with cache and default number of partitions

Stage	Observation	Stage Graph
0	Duration: 3.4 min Input: 9.9 GB Shuffle write: 5.3 GB Shuffle read: 0 Number of tasks: 338 The data is parsed and grouped by the from_page in stage 0.	 Stage 0 textFile /proj/uwmadison744-f19-PG0/data-part3/enwiki-pages-articles [0] textFile at NativeMethodAccessorImpl.java:0 /proj/uwmadison744-f19-PG0/data-part3/enwiki-pages-articles [1] textFile at NativeMethodAccessorImpl.java:0 PythonRDD [2] groupByKey at /users/szhong/CS744/a1p3/a1p3.py:18 PairwiseRDD [3] groupByKey at /users/szhong/CS744/a1p3/a1p3.py:18
1	Duration: 4.9 min Input: 4.6 GB Shuffle read: 6.2 GB Shuffle write: 5.3 GB Number of tasks: 338	 Stage 1 partitionBy ShuffledRDD [4] partitionBy at NativeMethodAccessorImpl.java:0 mapPartitions MapPartitionsRDD [5] mapPartitions at PythonRDD.scala:133 PythonRDD [6] [Cached] RDD at PythonRDD.scala:53 PythonRDD [7] RDD at PythonRDD.scala:53 PythonRDD [8] RDD at PythonRDD.scala:53 PythonRDD [9] RDD at PythonRDD.scala:53 union PartitionerAwareUnionRDD [10] union at NativeMethodAccessorImpl.java:0 PythonRDD [11] reduceByKey at /users/szhong/CS744/a1p3/a1p3.py:34 PairwiseRDD [12] reduceByKey at /users/szhong/CS744/a1p3/a1p3.py:34

2	Duration: 5.6 min Input: 287.4 MB Shuffle read: 10.4 GB Shuffle write: 3.7 GB Number of tasks: 338	 Stage 2 partitionBy ShuffledRDD [4] partitionBy at NativeMethodAccessorImpl.java:0 partitionBy ShuffledRDD [13] partitionBy at NativeMethodAccessorImpl.java:0 mapPartitions MapPartitionsRDD [5] mapPartitions at PythonRDD.scala:133 mapPartitions MapPartitionsRDD [14] mapPartitions at PythonRDD.scala:133 PythonRDD [6] [Cached] RDD at PythonRDD.scala:53 PythonRDD [15] RDD at PythonRDD.scala:53 PythonRDD [16] RDD at PythonRDD.scala:53 PythonRDD [17] RDD at PythonRDD.scala:53 union PartitionerAwareUnionRDD [18] union at NativeMethodAccessorImpl.java:0 PythonRDD [19] reduceByKey at /users/szhong/CS744/a1p3/a1p3.py:34 PairwiseRDD [20] reduceByKey at /users/szhong/CS744/a1p3/a1p3.py:34
3	Duration: 4.6 min Input: 264.4 MB Shuffle read: 8.8 GB Shuffle write: 3.4 GB Number of tasks: 338	 Stage 3 partitionBy ShuffledRDD [4] partitionBy at NativeMethodAccessorImpl.java:0 mapPartitions MapPartitionsRDD [5] mapPartitions at PythonRDD.scala:133 partitionBy ShuffledRDD [20] partitionBy at NativeMethodAccessorImpl.java:0 mapPartitions MapPartitionsRDD [21] mapPartitions at PythonRDD.scala:133 PythonRDD [6] [Cached] RDD at PythonRDD.scala:53 PythonRDD [22] RDD at PythonRDD.scala:53 PythonRDD [23] RDD at PythonRDD.scala:53 union PartitionerAwareUnionRDD [24] union at NativeMethodAccessorImpl.java:0 PythonRDD [25] reduceByKey at /users/szhong/CS744/a1p3/a1p3.py:34 PairwiseRDD [26] reduceByKey at /users/szhong/CS744/a1p3/a1p3.py:34

4	Duration: 5.1 min Input: 279.3 MB Shuffle read: 8.5 GB Shuffle write: 3.4 GB Number of tasks: 338	 Stage 4 partitionBy ShuffledRDD [4] partitionBy at NativeMethodAccessorImpl.java:0 partitionBy ShuffledRDD [29] partitionBy at NativeMethodAccessorImpl.java:0 mapPartitions MapPartitionsRDD [5] mapPartitions at PythonRDD.scala:133 mapPartitions MapPartitionsRDD [30] mapPartitions at PythonRDD.scala:133 PythonRDD [6] [Cached] RDD at PythonRDD.scala:53 PythonRDD [31] RDD at PythonRDD.scala:53 PythonRDD [32] RDD at PythonRDD.scala:53 PythonRDD [33] RDD at PythonRDD.scala:53 union PartitionerAwareUnionRDD [34] union at NativeMethodAccessorImpl.java:0 PythonRDD [35] reduceByKey at /users/szhong/CS744/a1p3/a1p3.py:34 PairwiseRDD [36] reduceByKey at /users/szhong/CS744/a1p3/a1p3.py:34
5	Duration: 4.6 min Input: 281.6 MB Shuffle read: 8.5 GB Shuffle write: 3.4 GB Number of tasks: 338	 Stage 5 partitionBy ShuffledRDD [4] partitionBy at NativeMethodAccessorImpl.java:0 partitionBy ShuffledRDD [37] partitionBy at NativeMethodAccessorImpl.java:0 mapPartitions MapPartitionsRDD [5] mapPartitions at PythonRDD.scala:133 mapPartitions MapPartitionsRDD [38] mapPartitions at PythonRDD.scala:133 PythonRDD [6] [Cached] RDD at PythonRDD.scala:53 PythonRDD [39] RDD at PythonRDD.scala:53 PythonRDD [40] RDD at PythonRDD.scala:53 PythonRDD [41] RDD at PythonRDD.scala:53 union PartitionerAwareUnionRDD [42] union at NativeMethodAccessorImpl.java:0 PythonRDD [43] reduceByKey at /users/szhong/CS744/a1p3/a1p3.py:34 PairwiseRDD [44] reduceByKey at /users/szhong/CS744/a1p3/a1p3.py:34
6	Since this stage was completed within virtually no time, the data for Stage 6 is not available from Spark.	

Code

```python
from pyspark import SparkContext, SparkConf

input_path = "/proj/uwmadison744-f19-PG0/data-part3/enwiki-pages-articles"

# set SparkConf and SparkContext
conf = SparkConf().setAppName("a1p3").setMaster("spark://c220g1-030823vm-1.wisc.cloudlab.us:7077")
sc = SparkContext(conf=conf)

# Read the file as RDD
lines = sc.textFile(input_path)

# Parse each line to a pair (from_node_id, to_node_id)
pairs = lines.map(lambda l: l.split("\t"))\
             .filter(lambda l: len(l) == 2)

# buckets is a list of tuples (from_id, [to_id_1, to_id_2, to_id_3, ...])
buckets = pairs.groupByKey()

# Set initial rank of each page to be 1.
# rank is a list of tuples (from_id, rank)
rank = buckets.mapValues(lambda e: 1.0)

# On each iteration, each page contributes to its neighbors by rank(p)/ # of neighbors.
def compute_contribution(to_id_list, rank):
    return [(to_id, rank / len(to_id_list)) for to_id in to_id_list]

for i in range(5):
    # out_edge_ranks is a list of tuples (from_id, ([to_id_1, to_id_2, to_id_3, ...], rank_for_from_id))
    out_edge_ranks = buckets.join(rank)

    # contrib is a list of tuples (to_id, contrib)
    contrib = out_edge_ranks.flatMap(lambda e: compute_contribution(e[1][0], e[1][1]))\
                            .reduceByKey(lambda e1, e2: e1 + e2)

    # Update each page's rank to be 0.15 + 0.85 * (sum of contributions).
    rank = contrib.mapValues(lambda l: 0.15 + 0.85 * l)

print(rank.take(10))
```