CS744 - Big Data Systems

Assignment 1

Akshata Bhat
akshatabhat@cs.wisc.edu

Kyle Klassy
klassy@cs.wisc.edu

Rohit Kumar Sharma
rsharma@cs.wisc.edu

September 25, 2019

1 Goal

- Deploy and configure Apache Spark and HDFS.
- Understand how Apache Spark and HDFS work, and interact with each other.
- Implement Sorting and PageRank algorithm in Apache Spark.

2 Environment

In part 1 of the assignment, HDFS and Spark were setup in a cluster in a 3-node cluster with Ubuntu 16 installed on each machine. Spark was configured with driver memory of 29 GB, executor memory of 29 GB, 5 executor cores and 1 cpu per task.

3 Sorting with Apache Spark

In part 2 of the assignment, we implemented a simple Spark application to sort an IoT dataset with 1000 records first by the country code alphabetically and then by timestamp to break ties in country code.

The input file was loaded into HDFS and read as an RDD. The sorted output was written to HDFS. The program takes about 1 *sec* to complete.

4 PageRank with Apache Spark

In part 3 of the assignment, we implemented the PageRank algorithm using Apache Spark. PageRank algorithm is used by Google to assign ranks to web pages based on the number of pages that outlink to it.

We used two datasets to run the PageRank algorithm: a smaller Berkeley-Stanford web graph dataset and a larger dataset based on Wikipedia article links. We used the larger dataset to understand the performance of Spark by analyzing various scenarios as described below. We run the PageRank algorithm for 10 iterations for the smaller dataset and 5 iterations for the larger one. The small dataset takes about 2 *min* to get the page ranks using 10 partitions. We use the larger dataset to perform further analysis and have presented the results in the following sections.

4.1 PageRank with no Custom Partitioning and no Persistence

In this task, we wrote a PySpark application implementing PageRank algorithm. In this implementation, we did not use any custom partitioning nor did we persist any RDDs in memory.

The total time it took to compute ranks for all the article links in the large dataset with 5 iterations and 337 partitions (default) is 30 *min*.

4.2 PageRank with Custom Partitioner

We extended the above implementation to use a custom partition function to partition the RDDs. Having a custom partitioner will make shuffle operations more efficient. We partition both links and ranks based on hash values of the URLs(keys) and URLs having hash values in a given range would go into the same partition. In the *join* operation since the corresponding URLs are in the same partition, Spark doesn't have to fetch the information from multiple partitions.

We used the inbuilt *partitionBy*() function which uses a *Hash Partitioner* by default. We also monitored the disk interface on which intermediate data is written and network interface during the experiment. It was observed that the *network* and *disk usage* was less when using the custom partitioner as we expected.

Impact of Number of RDD Partitions

We further analyzed the impact of using different number of RDD partitions while using a custom partitioner. We've found that the optimal number of partitions is around 200, and we've used this number for the remainder of the analyses below. The results are shown in figure 1.

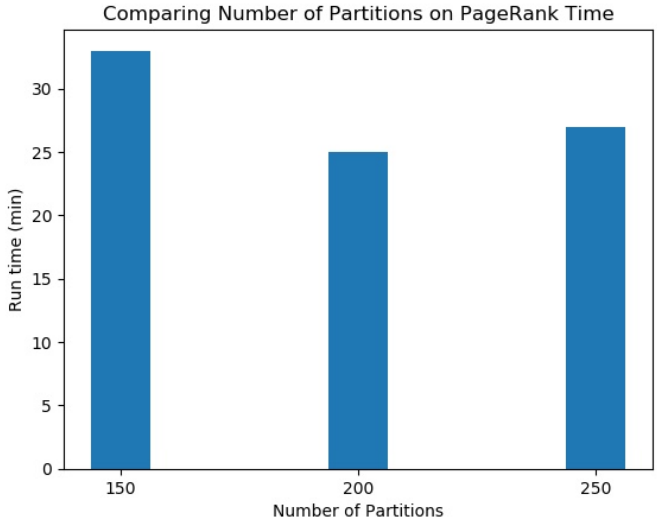

Figure 1: Impact of the number of RDD partitions

4.3 PageRank with Persisting RDDs in Memory

In this section, we analyzed the impact of persisting *links* RDD in memory. *links* RDD is used to update the page ranks in each iteration of the RDD algorithm. Figure 2 shows the persisted RDD being cached in the lineage graph during execution.

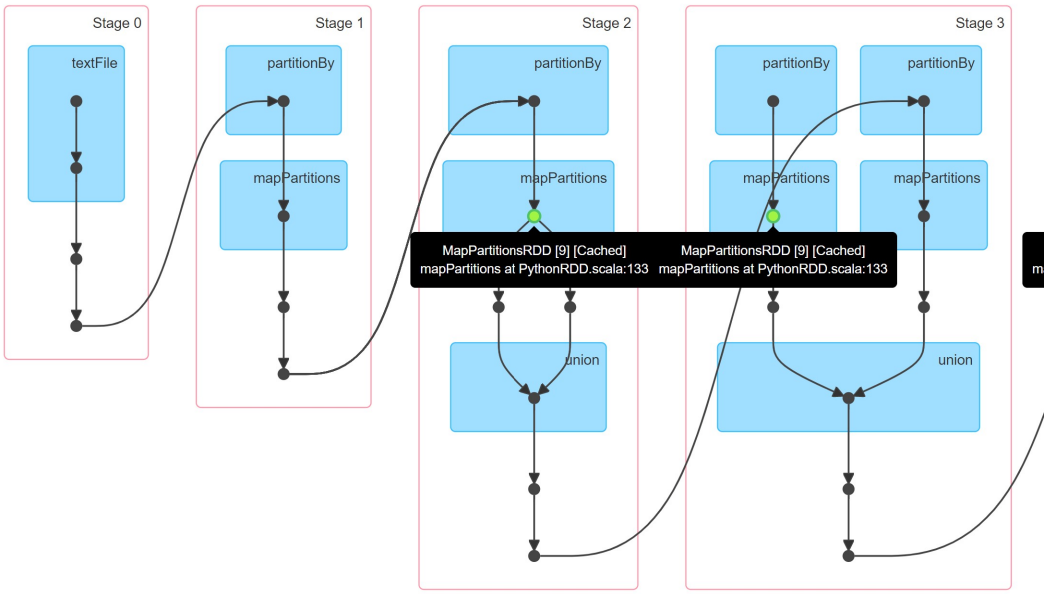

Figure 2: Lineage graph showing cached RDD

We expect the performance to improve by using persist. The results we obtained using persist are depicted in figure 3. We observed that there is not much impact of persisting the RDD in memory. This may be due to the fact that since the amount of memory available is more than the size of the RDD, even without specifically persisting it, Spark would have persisted it automatically.

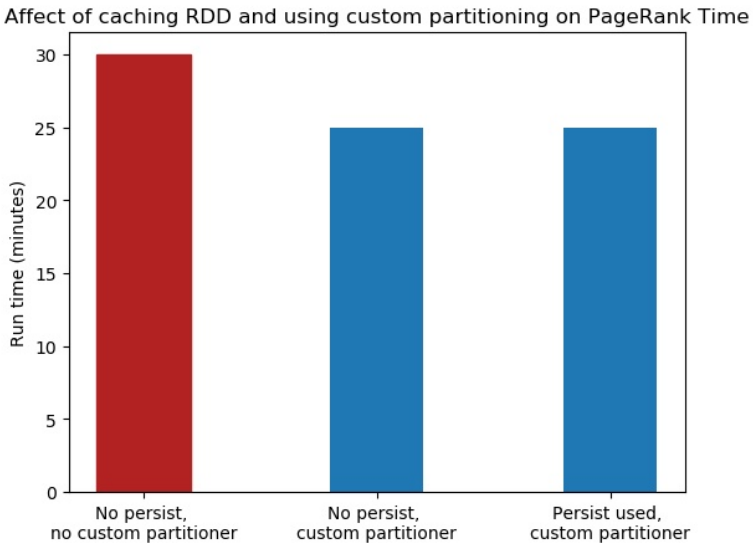

Figure 3: Comparing performance of using Custom Partitioner and Persisting RDD

4.4 Impact of Worker Failure

Here, we manually clear the memory cache and kill a Spark Worker midway during task execution. In this scenario, we observed that the killed tasks were being re-executed on the other available worker. This also leads to a reduction in the degree of parallelism. Therefore, the total runtime is increased. Spark also tries to optimize the execution of failed stages by not executing all the tasks again.

We observed that the total runtime is more when we killed the worker at approximately 75% of total time

compared to 25%. This might be because failure of a worker at an earlier stage leads to re-computation in a smaller lineage graph. Therefore, late failures have a worse effect on the completion time due to bigger lineage graphs. Figure 4 depicts this behavior.

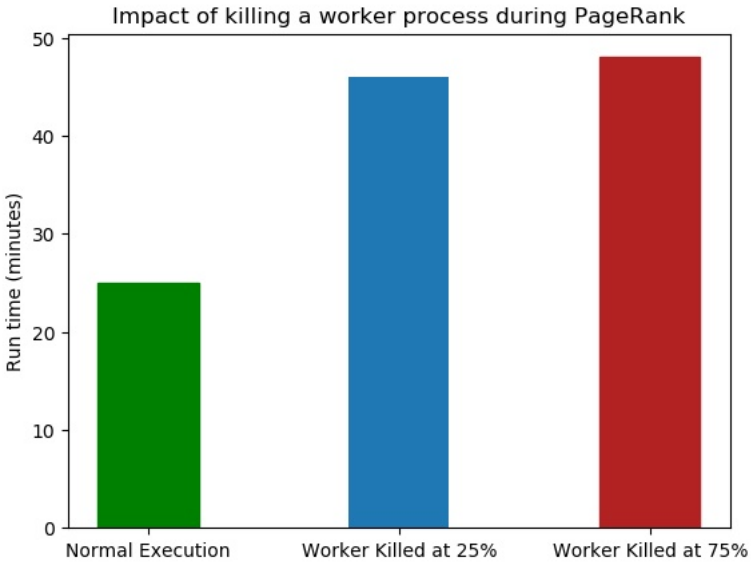

Figure 4: Impact of Killing a Worker Process