CS744 Assignment-1 Report

Mohammed Danish Shaikh, Mohan Rao Divate Kodandarama, Shreeshrita Patnaik
{mshaikh4, divatekodand, spatnaik2}@wisc.edu
University of Wisconsin, Madison

1 Goals

1. To understand deployment and configuration of Apache Spark and HDFS.

2. To understand how Apache Spark [4] and HDFS [3] work and interact with each other by writing simple Spark applications and running them on the cluster.

2 Experiment

2.1 Pagerank

2.1.1 Algorithm overview

We've implemented the pagerank algorithm as follows:

1. Read the file from HDFS.

2. Pre-process the file to filter out lines missing values, convert the text to lowercase and additional operations as required by the assignment.

3. Create *Ranks RDD* by assigning *1* to all distinct links in the file.

4. Create a *Links RDD* by splitting the lines into key, value pairs on basis of tab and grouping them by key.

5. Calculate a *Contributions RDD* by first performing a join on *Ranks and Links RDDs* and then mapping the URLs to the *Contributions RDD*.

6. Obtain the new *Ranks RDD* from the *Contributions RDD*.

7. Repeat steps 5 and 6 for 10 iterations.

2.1.2 Custom Partitioning

We implemented this by utilizing the HashPartitioner provided with Spark [1]. The idea was to partition ranks and links based on the hash values of the URLs (keys). This way, the URL's having hash values in a given range would all go to a single partition. A join operation could benefit from this in that only the corresponding partitions would require to be joined as compared to all-to-all join expected in a naive case.

2.1.3 Caching

It made sense to cache the *Links RDD* since it was getting utilized by the join operation in every iteration. It would also had been a good idea to cache the *Ranks RDD* after every X iterations (for some value of X). However, given that the data set and the number of iterations were not big, we thought the effect of caching would not be that much pronounced, i.e. recomputing might be a better idea.

2.1.4 Killing

We let the experiment run to 50% of it's completion, i.e. half of the total number of stages. We then killed one of the worker processes manually and cleared the machine cache.

2.2 Sorting

The Spark application sorts an input CSV file, according to the *country code* and then the *timestamp*. In the application, we load the file, separated by delimiters into *dataframe*. The *dataframe* is then sorted and written back as a CSV file.

2.3 Environment

HDFS and Spark were setup in a cluster consisting of three nodes containing *5 cores* and *32 GB* RAM each. The pagerank algorithm was executed over an input data size of *1.5 GB*.
All our code and experimental results can be found in the repository.

3 Observations and Analysis

3.1 Pagerank

The experiments conducted running PageRank with variations of the number of partitions, with or without persisting the RDDs in memory and finally on killing the worker midway in the job, leads to some salient observations.

3.1.1 Runtime

On adding a Custom Partition, based on Hash Partitioning, we notice a significant reduction in Runtime, with an increase in the number of partitions (Figure 1). We attribute this to the Shuffle Read per concurrent task running on a core, which decreases with increase in number of partitions (Figure 2). We also see a very good runtime and performance with the default partitioning of 16, comparative to Hash Partitioning

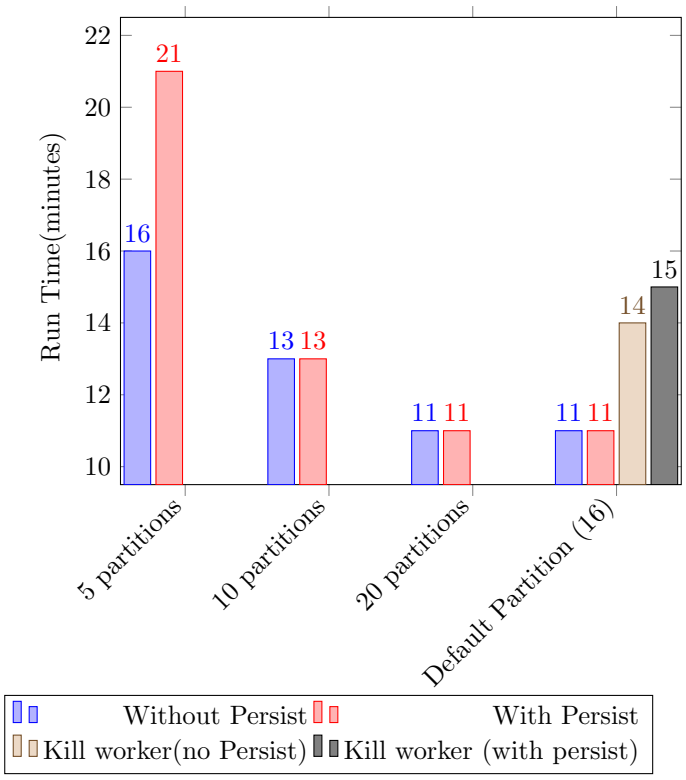

Figure 1: Run Time for Different types of Runs.

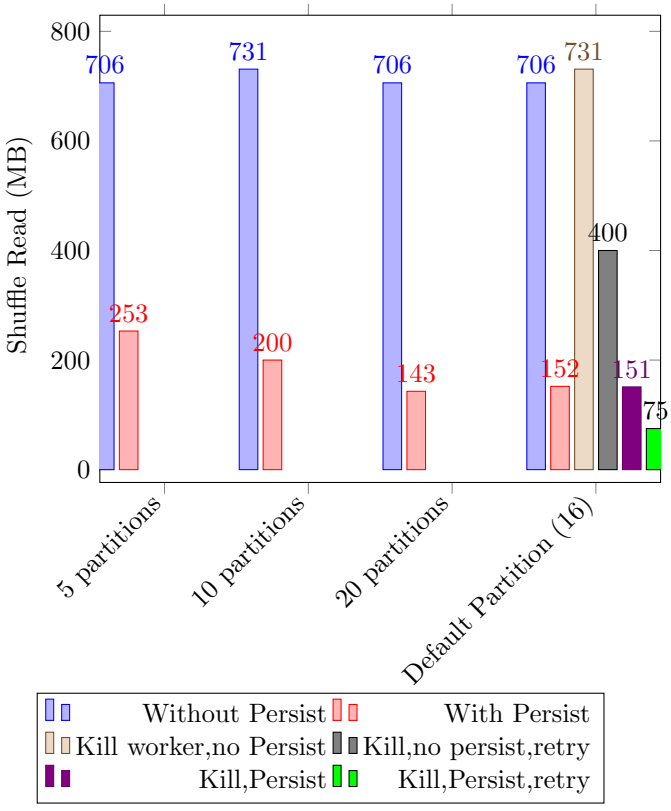

Figure 2: Shuffle Read for Different types of Runs

of 20 (Figure 1). We attribute this to the fact that operations like *distinct* and *groupByKey* (Figure 4), in lieu of sorting the data, would be creating a range like partition. Furthermore, the runtime doesn't seem to significantly change across any of the cases. We speculate the reasoning behind this to be the data size. We did perform an experiment on entire data (10GB) set while running spark on *local mode* with *4 cores*. It was observed that the runtime improved from 1.7 hours in no custom partitioning case to 53 minutes in case of creating 10 custom partitions based on hash value of links. None of the RDD'S were persisted in both cases.

3.1.2 Shuffle Read and Shuffle Write

We observe that the Shuffle Read (Figure 2) decreases when we custom partition the RDD's based on the hash values of links. This makes sense because the join would now consist of only joining the corresponding hash partitioned partitions. We also note that there is no significant difference in Shuffle Write in either of the cases considered. We also note from Figure 2 that Shuffle Read is significantly less in cases wherein the *Links RDD* is persisted. This is because the corresponding data is obtained from the memory of the worker directly.

3.1.3 Input and Output

We observe that the cached data is used as input for joining in each iteration. The input cached data reduces in cases wherein the executor is killed since the new executor only needs to read the data for partitions that need to be recomputed.

3.1.4 Garbage Collection Time

	5 Partitions	10 Partitions	20 Partitions
% of Time spend on GC	13.18	16.7	15.6

Table 1: Table showing the effect of number of partitions on Garbage Collection

Tables 1, 2 and 3 summarize the effect of partitioning, persisting the RDD's and Killing the executor process on the fraction of time spent on Garbage Collection.

	Persist	Without Persist
% of Time spend on GC	8.8	13.1

Table 2: Table showing the effect of persisting the RDD's on Garbage Collection

As shown in Table 1, fraction of time spent on Garbage Collection (GC) is significantly large for computations with 10 and 20 partitions. Table 2 captures the effect of persisting RDD's on GC.Finally, Table 3 shows the effect of persisting the RDD's when one of the executor is killed halfway through the computation. From Table 2 and 3, we note that the fraction of time spend on GC is improved when the RDD's are persisted.

	Persist	Without Persist
% of Time spend on GC	8.5	13.89

Table 3: Table showing the effect of persisting the RDD's when one of the executor is killed on Garbage Collection

3.1.5 Fault Tolerance

We evaluated this across the following failure cases:

1. Killing the executor process.
2. Killing the worker process.

As soon as we kill an executor process, a new executor is spawned which recomputes only the required partitions (Figure 3). Furthermore, cached partitions are utilized for recomputations if caching is done. This is confirmed by the following observations combined:

1. The stages that had already completed are retried.
2. The Shuffle Reads and Writes for retried stages are significantly lesser as compared to a normal case.
3. The Input for retried stages with cached data is significantly lesser as compared to a normal case.

As soon as we kill a worker process, the number of executors goes down by one. The tasks that were being done by the dead executor are now retried by other alive executors. If the worker process being killed happens to be the last alive worker in the cluster, the job goes to a *WAITING* state until a worker (existing or new) enters the cluster. The worker then spawns an executor that executes tasks of the job.

3.1.6 DAG

We make the following observations from the DAG's in Figure 4:

1. In case of custom partitioning, a wide dependency is introduced at the point of partitioning.
2. The DAG shows the RDD's being cached.
3. The DAG separates stages at wide dependencies. This enables pipeline optimization within the stages.

3.2 Sorting

The total runtime of the Sorting Application, applied on a dataset of 110kB was approximately 25 seconds.

4 Conclusion

Our learnings from the project, helped reaffirm the objective goals of the project. Through the project, we were able to set up *Apache Spark* and the *Hadoop Distributed File System*, and correctly configure the environment. We were able to successfully understand and use the *spark-shell* and the *spark-submit* processes, along with building and running Spark applications in Scala [2]. We were able to observe, diagnose and troubleshoot the components and the interaction between the said components.

5 Discussion

Mentioned below are some aspects not directly related to our analysis, but something that we think are worth mentioning:

5.1 Deadlock

We submitted four jobs to Spark consecutively, each configured with *executor cores = 5, driver cores = 1*. One job started *RUNNING*, two jobs were assigned drivers and were in *WAITING* state, whereas one job was *UNASSIGNED*. As soon as the first job finished executing, the *UNASSIGNED* job was assigned a driver and went into *WAITING* state. This meant that a driver was running on each machine in the cluster, thereby setting free only 4 cores per machine. Since the executor required 5 cores to execute, all three jobs were in the *WAITING* state indefinitely. We had to manually kill one of the jobs to break this deadlock.

5.2 Memory Spill

During our initial runs, we were not setting the configuration option of *executor memory* to *8g*. This lead to memory spills into disk since the default *executor memory* is set to *1g*. After realizing the issue, we changed the configuration to *8g* and we didn't see any memory spills in the logs anymore.

6 Acknowledgements

We would like to thank Prof. Akella for the interesting assignment, which served as a self-directed learning of Spark and Hadoop. We would also like to thank the TAs and our batchmates, for helping us with our queries and raising important observations.

References

[1] Spark Contibutors. *Spark*. https://spark.apache.org/. Accessed 2019-02-15.

[2] Scala-Sbt contributors. *Scala-Sbt*. https://www.scala-sbt.org/. Accessed 2019-02-15.

[3] Konstantin Shvachko et al. "The Hadoop Distributed File System". In: *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. ISBN: 978-1-4244-7152-2. DOI: 10.1109/MSST.2010.5496972. URL: http://dx.doi.org/10.1109/MSST.2010.5496972.

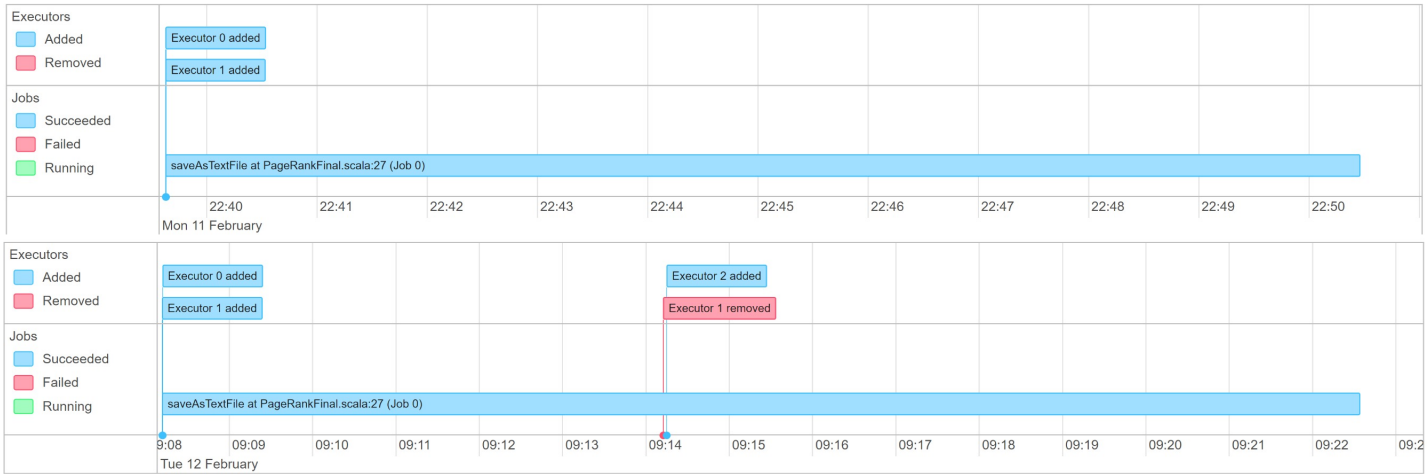

Figure 3: Event timeline for one of the cases (Normal Execution v/s Killed Execution)

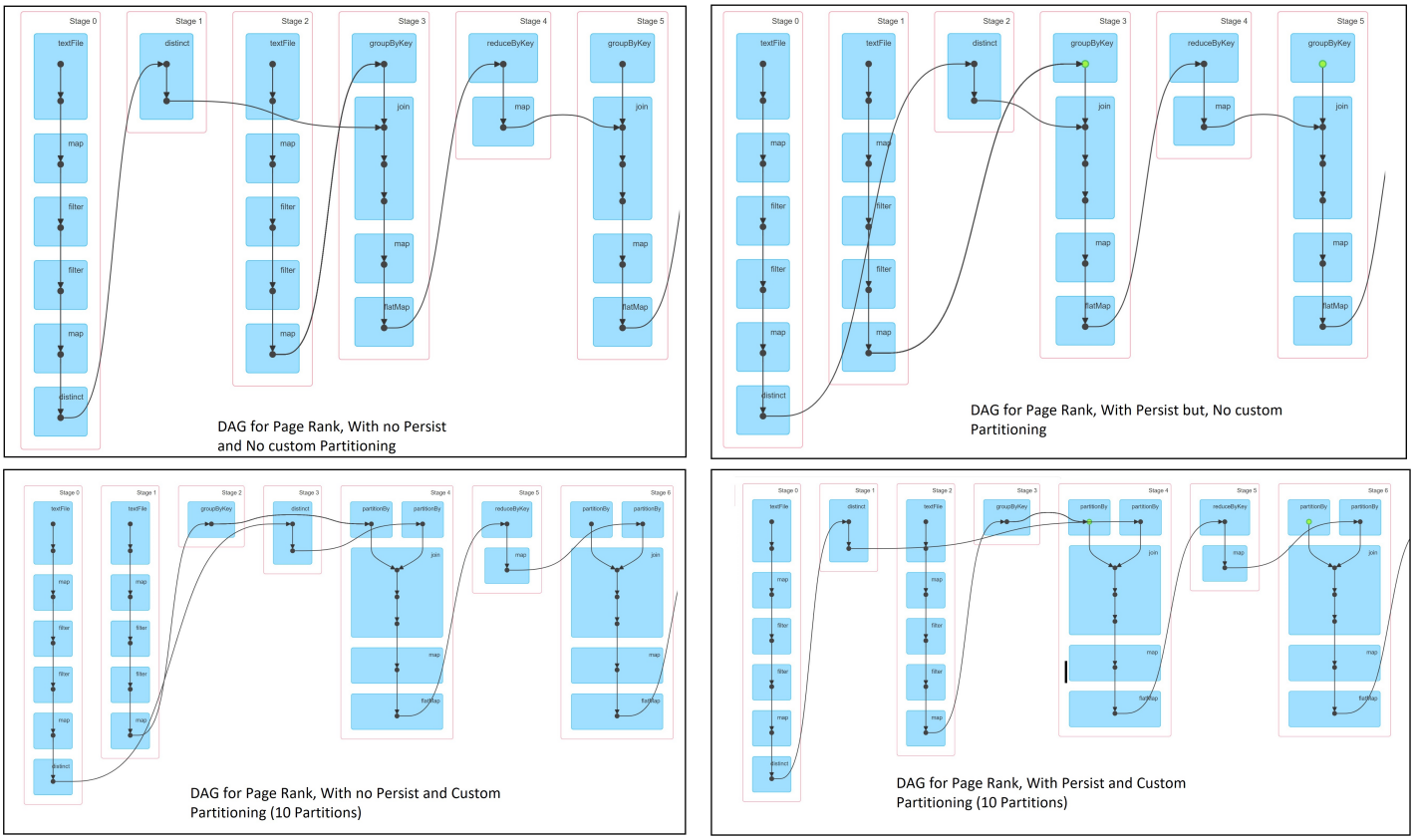

Figure 4: DAG for PageRank with different Parameter settings

[4] Matei Zaharia et al. “Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing”. In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI’12. San Jose, CA: USENIX Association, 2012, pp. 2–2. URL: http://dl.acm.org/citation.cfm?id=2228298.2228301.